Open Systems Design Using Agent Interactions

by

Simon Miles

A thesis submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy in Computer Science

University of Warwick, Department of Computer Science

September 2002

Contents

List of Figures

List of Tables

Acknowledgments

I would like to start off by thanking Mike Joy, who has been an accessible and kind supervisor as well as organised and knowledgable. Next, I would like to thank Mike Luck, who has given up plenty of reading time to pour over many drafts of this thesis and has been ready to help out whenever I've needed it. Nathan Griffiths should also be thanked for being a friendly face to start my PhD life, and always good for a chat. I would also like to thank the Department of Computer Science in general for providing the opportunity, including funding, to conduct this work.

I've also enjoyed the encouragement and support of many friends at Warwick over the four years, as well as my parents and sister. I'm grateful to all these people for providing the necessary balance for a happy PhD.

Finally, I would like to thank Jenny Bunker who has been all of the above and more. Not only has she has put her philosophical mind to dissecting my thesis drafts but has provided plenty of sparkle to my life.

Declarations

I declare that all the work presented here is my own and has not been submitted for a degree at another university. As part of my PhD research, I have had two papers accepted for publication in conference proceedings including portions of this work [76, 77].

Abstract

As software requirements grow increasingly complex, the need to connect to and *re-use* existing, tested software, grows with it. Open systems, such as the Internet, aid this process by connecting together software services provided by a range of organisations, and the distributed nature of the system allows the services to be regularly updated and improved. Applications can be deployed within the open systems that *opportunistically* attempt to make use of the best functionality available at any one time. Agent-based systems have been proposed as an ideal way to implement such applications, due to their flexibility and distributed control. However, a balance must be kept between acting opportunistically and ensuring that each application operates to the standards demanded by the application requirements. Determining whether an application will perform to its requirements necessitates *justifying* the design decisions made in creating it. Our goal is to provide application developers with the means to create justified designs for opportunistic applications. The main contribution of this thesis is a software engineering methodology, *agent interaction analysis*, based on a set of independently valuable techniques we have developed. The first of these is a novel approach to modelling applications as being instantiated by a set of agent interactions, allowing such applications to be described with minimal restrictions on their implemented structure. Second, we provide a technique, based on design patterns, for comparing mechanisms for instantiating parts of multi-agent system. Finally, we provide an approach to more detailed analysis and comparison of coordination mechanisms,

Chapter 1

Introduction

As software grows more and more complex, application developers aim to *re-use* as much of the existing software as possible. Fundamentally, this thesis is concerned with how this re-use can be achieved in a reliable way.

A set of software services and applications managed by a defined set of users, e.g. the software on a personal computer or on an organisation's network, can be called a *software environment*. Connecting software environments together provides the potential for applications to re-use the most up-to-date functionality in any of those environments at any time, and for that functionality to be continually extended by adding more environments and adding new services to the existing environments. Such a connected *system* is said to be *dynamic*, as the environments' content may change at any time, and *open*, as environments may be added while an application is running on the system. A popular example of a dynamic open system is the Internet.

An application developer wishing to make use of the changing functionality of an open system must add software processes to the system that utilise the available services so that the application as a whole comes into existence. For example, an application that makes travel arrangements for a holiday on request may be realised by a process that sequentially uses flight booking, hotel booking, currency changing etc. services on the Internet. To achieve more complex applications, developers may need to deploy their own services into the open system, e.g. a service to search for tourist attraction websites could be added to the former application. For most effective re-use, the addition of other services, such as new

hotels, to the system should cause the application to decide whether the new services are better than currently used services and utilise them if so.

This chapter introduces the specific problem we are looking at and the broad approach taken to solve it. In Section 1.1 we examine the demands on applications that wish to run within an open system. We examine the most significant problems in developing open system applications, i.e. meeting the application requirements while also making use of future open system services, in Section 1.2. Section 1.3 examines how an approach based on autonomous, decision making entities, *agents*, is ideal for meeting these demands. Section 1.4 examines how a solution to the development of one open system application could be generalised over many applications. We define the specific aims of this thesis in Section 1.5. In Section 1.6 we introduce the case study application used as an illustrative example throughout this thesis. We summarise the problem and describe the structure of this thesis in Section 1.7.

1.1 Open Systems

Open systems are made up of an unlimited and varying number of connected sub-systems [52]. Sub-systems will change, be added and be removed during execution of an application. Open systems allow different (human) organisations or parts of a single organisation to share computational services and resources. Each subsystem can still be maintained separately to ensure the security and integrity of its stored data.

Due to the dynamic nature of the elements of an open system, the behaviour of the system as a whole will be unpredictable. In order to perform consistently, an application within an open system must take account of the following.

- The application should be *robust* so that those changes in the open system forseeable by the application designer do not break the application.
- The application itself should *compensate* for failures in the system by not failing, or gracefully failing (exhibiting *graceful degradation*).
- The application should be *easy to maintain*, so that it can be modified to cope with *radical changes* in the system (where this would be possible). By *radical changes* we mean changes that the application could not have initially been designed to take

account of.

- The application should be *easy to extend*, so that it can be modified to take advantage of useful radical changes.

1.2 Justification and Opportunism

As illustrated in the introduction, the development of applications for open systems is equivalent to asking the question: what should the designer add to the open system in order to realise the application?

An application yet to be developed is described in terms of *requirements*, which are the desired and expected abilities of the application presented to the application designer as a written document, through repeated interviews with the potential users and/or through any other means of requirements capture. In order to ensure an application meets the application requirements, and to satisfy users that this is the case, the application design should be clearly *justified* by the application requirements. The problem becomes more challenging in an open system as the future form of the system will not be known at design time. The clear connection between a design and its requirements is made even more important for applications running in constantly changing environments by the difficulty with producing and executing meaningful empirical tests in those environments [53, 109]. Therefore, *justification* of an application design, by reference to the requirements, is of primary importance in our solution.

To allow an application to re-use the best of existing functionality in the open system, and to allow it to take advantage of improved functionality as it appears, it should be *opportunistic*. Opportunistic applications are flexible enough to decide what functionality to use from that known to be available at any time.

There is an obvious conflict between justification and opportunism in that the former requires constraints while the latter requires flexibility. The primary problem that this thesis considers is how best to resolve the conflict and so solve the problem of what to add to an open system to create a justified, opportunistic application.

1.3 Agent-Based Systems

In order to cope with the unpredictability of open systems described above, we propose designing open system applications in terms of autonomous, flexible, decision making software entities, or *agents*. All the beneficial characteristics that an agent-based approach supplies are required by applications for dynamic open systems [11, 24]. In [58], Jennings states that agents are clearly a suitable approach for decomposable complex systems, including open systems, as agent decompositions are a natural way of analysing complex systems, and agent-oriented abstractions provide suitable models of complex systems.

Following on from Wooldridge and Jennings' definition [60], we consider agents to be *flexible*, *social* and *autonomous* entities. There is significant robustness offered by entities that can perform actions based on the current context of the system, rather than solely on the demands of other entities, i.e. are autonomous rather than passive objects. Aside from this benefit, the characteristics of flexibility, social behaviour and autonomy are particularly useful in allowing an application to be opportunistic. We consider each of these characteristics in turn below.

First, flexibility is an important characteristic in unpredictable environments as it potentially allows agents to take advantage of a wide range of situations. Such behaviour requires agents to *pro-actively* seek to achieve *goals*, i.e. desired states of the environment, while reacting to changes in their environment. It allows the agents, therefore, to be opportunistic in taking account of the current system state.

Second, social behaviour is a very important in being opportunistic. The entities within the environment that might most usefully be taken advantage of are those that are also flexible and social, i.e. other agents. The cooperation of several opportunistic agents allows all stages of an activity to be opportunistic. To get the most from social interaction, agents will *coordinate* their activities, ensuring they mutually fulfill their goals as best as possible, judged by whatever criteria the agents employ.

Finally, autonomy is a necessary characteristic for opportunism, as the agents may have to react immediately to take advantage of the current situation without deferring decisions to another entity. Pro-actively attempting to seek goals also implies some independent continuous activity.

1.4 Generality and Consistency

Creating a *justified, opportunistic design* (by which we mean a justified design of an opportunistic application) for a single application may be useful for users of that application. However, it would be preferable to create a more general approach that could be used to develop justified, opportunistic designs for many applications. The problem of creating a general design approach is within the domain of *software engineering.* Such encoded techniques for development are called *methodologies.*

Aside from generality, methodologies also provide *consistency.* The effort to consider possible problems, opportunities and effects of a design should be placed on the methodology rather than the designer. A standard method of description has the additional benefit that a designer will be more able to interpret the reasoning used in creating the designs of other applications, because the reasoning will be presented in the same form.

We therefore believe that the most useful way of solving the problem of creating justified, opportunistic designs is to produce a methodology for doing so. The principles of software engineering that ensure generality and consistency must be observed in our methodology in order to resolve the above issues. These principles are examined further in Chapter 2.

1.5 Aims

When developing an application that will opportunistically re-use the resources in an open system, and especially when using appropriate but largely unproven technology such as multi-agent systems, it is very important that the developers can justify that their design will meet the application requirements over the course of the application lifetime. In this thesis, we aim to provide a means for designing such appliations which forces designers to justify their design decisions throughout the process. The specific aims of the thesis are as follows.

- We will create a widely applicable methodology for designing open system applications.
- Use of this methodology will be illustrated with a case study.
- Techniques will be demonstrated for comparing the known mechanisms that could instantiate each part of an agent-based system.

- We will show how to check justification of design.
- We will show how to check flexibility for opportunism in design.

1.6 Case Study

In this section, we present the requirements of an example application in order to illustrate important aspects of creating justified designs. When mentioning *the user* any time in the application requirements below, we refer to any person using the application.

1.6.1 Requirements

We require a collaborative weather mapping application. Using the application, a global weather map giving the current state is accessed and edited by various collaborating organisations. Contributors can add data they have gathered locally to the map in authorised locations. For example, one contributor organisation may be authorised to add data to a small local area, while another can add to any location within a country. Authorisation is enforced to prevent accidental changes. Contributors and other (paying) organisations can access the weather data and be provided with predictions of weather at specified locations in the future.

For speed and robustness in a system where organisations' software services may become accessible at any time, and later stop being so, the weather data is distributed among the contributors. As different organisations require access to different local areas, the data should be distributed to try to ensure each organisation has as rapid access as possible to the data they need. This distribution should be updated regularly to reflect the current demands.

Several services offer predictions based on the data, and the number of predicters available at any one time may vary, partly due to the load they each have on them. The predicter services vary in speed and in accuracy. They must have access to the weather data to make the prediction, either by moving onto the system on which relevant data is stored or by repeated requests to the relevant sources.

Prediction requests specify location, time (in the future) and whether speed or accuracy should be the priority in producing results. A prediction command, from a user, that prioritises speed of completion is called a *speed view* and one prioritising accuracy of

prediction is called an *accuracy view*. On a speed or accuracy view, the application should always report back to the user within 10 seconds either with the prediction or with a 'time out' warning. All currently known predicters offer a time out service whereby the prediction is halted and a warning returned after a specified interval.

As any software service may become inaccessible to the rest of the system at any time, the functionality of the application should be available locally on the computer system of each contributor organisation, and preferably the same software will be loaded at each node for ease of deployment.

1.6.2 Variations

To illustrate the effects of different application priorities, we give two variations on the application requirements. The preferences below are ones which could have been given by the people drafting the requirements in addition to the text above.

Speed Variation After the priorities of the particular operations of the application, e.g. accuracy in the predictions produced from an accuracy view command, speed should be the most important factor in considering how the application is designed and implemented.

Interoperability Variation After the priorities of the particular operations of the application, the ability of the application to interoperate with and take advantage of many services in the open system should be the most important factor in considering how the application is designed and implemented.

1.6.3 Concerns to be Addressed

With regards to the problems expressed earlier in this chapter, we can see how these requirements would not be best fulfilled by an application that was not opportunistic and justified. Without the application acting opportunistically, the best services available in the open system at any one time would not be used. This would most obviously include the fastest, most accurate predicters, but could also miss the use of more complex services such as those handling the entire process of contribution of weather data. Such a service may provide better functionality than others of the same type by being faster, transmitting

information more securely, storing the new data where it would be more often used locally etc.

If the design was not justified, i.e. not all design decisions were explicitly matched to the reason for them in the requirements, then the user could not know it was fulfilling the commands given to it as quickly, accurately etc. as it could. Also, without justification, it cannot be stated that the application is acting as opportunistically as it could. Finally, without justification for the design decisions made, the parts of the design developed within this application could not be re-used for others.

1.7 Summary

We would like applications deployed in an open system to be able to *re-use* as much of the functionality available in that system as possible in fulfilling their requirements. In order to achieve this, our main goal is to create a methodology for developing justified designs for opportunistic applications. Alongside this we aim to ensure that the designs produced by our methodology are themselves re-usable wherever appropriate.

Chapter 2 provides a background on agents in software engineering and in applications for dynamic open systems. We present the main theory behind our approach in Chapter 3. Our methodology, *agent interaction analysis*, is described in Chapters 4, 5 and 6 on analysis, design and agent coordination issues respectively. Chapters 7 discusses the implications of and conclusions drawn from our work. Throughout the thesis, we illustrate our approach with the case study given above.

Chapter 2

Background on Agent-Oriented Software Engineering

In this chapter, we review the current state of the art in software development using agents as a primary design abstraction (*agent-oriented software engineering*). The aim is to reach a point of understanding from which we can judge the adequacy of existing techniques in the consistent development of justified opportunistic designs. We must first understand the demands of software engineering in general (Section 2.1) to judge the worth of methodologies, and why agent-oriented software engineering have been suggested as an improvement on standard approaches for certain classes of application (Section 2.2). We then look at the necessary concerns in developing agent-based systems, including how the application functionality is divided between that which an agent may choose to use and that which supports agents in all their activities (Section 2.3). In judging whether an approach is going to to produce opportunistic designs, we must understand how one part of an open system can make use of functionality elsewhere in the system. This is called 'coordination' and it is introduced in Section 2.4 with different approaches examined in Sections 2.5 (centralised coordination), 2.6 (distributed coordination) and 2.7 (organised coordination). Finally, we describe the existing agent-oriented software engineering techniques themselves in Section 2.8 for evaluation in the next chapter. We conclude our review in Section 2.9.

2.1 Software Engineering

Software engineering methodologies are used to produce designs in a consistent, generalised and justified way. In this section we examine general qualities proposed as requirements for good software engineering methodology (though these qualities are not entirely uncontroversial in software engineering literature). This will provide us with a background for judging whether existing agent-oriented methodologies are adequate for meeting the aims discussed in Chapter 1.

The approaches described assume that there is a clear distinction between the requirements and domain set out for an application and the design produced to produce the application within the domain. Alternative methods of development could generate implementation based on the interaction between users and a system, for instance.

2.1.1 Problems of Software Engineering

The *problems* with the development of good software have been recognised and have informed the creation of recent methodologies. They include the following [22].

- How can we predict the amount of time and effort required in developing a product (an implemented design)?
- How can we ensure that the developed system is correct with regards to the original requirements?
- How can we lower the cost of maintaining and updating the developed system?
- How can we avoid wasting time on developing something several times over?

2.1.2 Desirable Characteristics

The problems above can be tackled by improving either the tools or the processes of development. In the latter case, several *characteristics* have been found to be desirable in a development methodology, including the following [93]:

Understandability The methodology should produce designs that are explicitly defined and easily interpreted.

Visibility The methodology should produce clear results within the course of the design process to show its progress.

Supportability Use of the methodology can be eased by automation of parts of the process, and the structure of the methodology should allow this.

Acceptability The methodology should seen as useful to software engineers. This is often achieved in new methodologies by re-using concepts familiar from already established methodologies.

Reliability Errors in the design produced while using the methodology should be detected and removed without affecting the final product.

Robustness The methodology should be resilient to unexpected problems.

Maintainability The methodology should be suitably flexible to cope with changes in requirements or improvements in method.

Rapidity The methodology should quickly deliver a complete product.

While these characteristics are all required to an extent to make a process useful, it is not always possible to optimise them all as some characteristics may negatively affect each other.

2.1.3 Engineering Principles

Recognising the need for these characteristics, certain *principles* have been identified which the methodologies should abide by. Several are identified in [44] and described below.

Rigorous A methodology is *rigorous* if it has certain qualities such as being systematic, repeatable, justified by the requirements etc. The aim of being rigorous is to cover all areas of relevance to the problem. A stricter form of rigour is *formality*, which, in this context, implies mathematical proofs of validity.

It is noted [44] that it may be best to apply more rigour in some parts of the process than others, e.g. in balancing reliability with rapidity (see the characteristics listed above). It would be useful for the methodology to highlight areas which may require more detailed analysis to ensure rigour of the design as a whole.

Separation of Concerns It is often possible to divide problems in various ways which allow developers to focus on smaller, less complex goals. There are various ways of separating concerns such as separation by time, by desired product qualities, by views of the system under development etc. In all cases, only the significant relevant factors are given attention. Sometimes it may be necessary to make some design decisions before a separation can occur.

Modularity One specific way of dividing the system (separating concerns) is by the structures implementing different functionality, i.e. dividing it into *modules*. This allows the designer to first focus on the individual modules and then the interaction of those modules. Ideally, modules should display *high cohesion* (component elements are strongly related in functionality) and *low coupling* (modules are reasonably independent of each other).

Abstraction Abstraction is the principle of ignoring the details of a problem until the higher level decisions have been made. It applies both to the product under development and the methodology by which it is developed.

Anticipation of Change A system should be easy to extend or alter in part without causing problems in the rest of the system. The possibility of extension or re-use should be recognised in the design of the original system and should be promoted by the methodology. The software should easily remain consistent with the design under minor changes (which may not be intentional).

Generality It is often useful to look at a more general case than the problem at hand. This may be useful because the general problem is simpler, because it encompasses more cases (making it more anticipatory of change) or because it could be useful in solving future problems including those unrelated to the system currently under development.

Incrementality It is useful to get feedback during the process of development. This allows designers to get a better understanding of the requirements and anticipate changes. The difficulty is that a system cannot be tested nor a design verified until a certain proportion of it exists. The idea of incrementality is to make successively closer approximations of the system so that it can be repeatedly analysed.

One particular use of these principles is the division of the methodology into *phases*. The division can encompass all the principles above and help to ensure that the overall process fulfils a lot of the characteristics mentioned earlier. In Fusion, for example, [22] there is an *analysis phase*, for deriving detailed and tightly structured information from the requirements; and a *design phase* in which design decisions are made based on the analysis phase results.

Differences of opinion exist in software engineering research, but we assume that methodologies that incorporate the above principles to a greater extent are likely to be more productive than those that do not. The benefits such methodologies bring about include the control of errors, verification of the requirements, understanding of the problem, reduction of changes required to the design after implementation begins, easier separation of work load in time or between people, standard notations for communication between developers and easier maintenance [22].

The above problems, characteristics and principles are a useful guide to assessing the effectiveness of a methodology. They can highlight the areas in which a method will not work particularly well. In the rest of this section we discuss some of the techniques used to achieve these desirable characteristics in object-oriented development, which is an obvious starting point for achieving the same in an agent-oriented approach.

2.1.4 Re-use, Generality and Consistency

We wish to aim for re-use of other products, such as design parts and services in an open system. It is therefore worth looking at standard contemporary software engineering approaches to re-use, generality and consistency, including *design patterns*, *software architectures* and *requirements engineering*.

Design patterns are a method of abstracting away from the structures in a design so that the structures can be used in comparable contexts in other designs [39]. They are an important development in the *re-use* of design parts. For example, the practice of using one object (a Factory) to create other objects of a give type (Products) and thereby hiding the creation process from the client requiring the Products, is encoded in the *Factory* design pattern [40].

Design patterns are described using a schema called a *pattern language*. There are several different pattern languages in use. One important part of a pattern's description is

information on when it should be applied. This information should be expressed in such a way so as to allow comparison between possible patterns. In [89], Rising gives an overview of several pattern languages. Examples of the criteria concerned with pattern selection, used in different languages, are given below.

Consequences The consequences of a pattern are the trade-offs and results of using that pattern.

Applicability/Context The applicability of a pattern is the situation or situations in which the pattern would be useful

Forces/Constraints The constraints on a pattern are the considerations, possibly contradictory, affecting choice of a pattern. The pattern may optimise some forces while ignoring others, or have priorities regarding different constraints.

Preconditions The preconditions of a pattern are those design decisions that should be considered and/or made before this one.

Using a pattern language to describe possible *models* for parts of a design aids selection. By applying the same criteria for all models we have a sound basis for comparison. Models relating to a particular choice can be expressed in a standard form to provide useful comparison. For example, deciding between using the Factory pattern mentioned above and a different object creation pattern which involves the user more in the creation process (e.g. the Builder pattern), is eased by being able to compare their applicability using entries in a single pattern language schema.

Describing agent-based systems as patterns is an idea put forward by Kendall in [63]. Object-oriented role models, consisting of object roles, are extended to collections of agents comprising agent roles. Such collections found to be useful in particular applications can be added to a pattern library for re-use. Re-use of patterns is much more productive, and likely to produce better outcomes, than deriving every low-level part of the design from first principles from the requirements. Another application of design patterns to agent-based systems is given by Hayden et al. [51] where system-wide coordination mechanisms are described by design patterns. Aridor also proposes using design patterns for agent-based systems [3], though in this case the design patterns are in exactly the same form as for

object-oriented designs, so there is little account of agent-specific concerns in the pattern language used.

Extending the design pattern idea, *software architectures* are abstract design structures defining the whole of an application at a high level, and their benefit and use, can also be explained using pattern language, though they may be developed through analysing theories rather than purely from experience of implementation. Software architecture research examines how architectures can be assessed and compared. At a more detailed level of design, a standard language is also useful. A graphical notation is commonly used for making the design specifications produced by methodologies useful for communication and generally *understandable*. The most prolific graphical design specification language in object-oriented design is the Unified Modelling Language (UML) [4].

One part of the development process not strongly supported in UML is the initial capture, analysis and transformation of application requirements. This broad stage is called *requirements engineering*. In [96], Lamsweerde and Willemet develop a technique for deriving goals from *scenarios* (and vice versa). Scenarios are descriptions of users interacting with objects in the application domain and are useful for capturing required functionality. Scenarios can be represented in the form of *event trace diagrams*, which are "well-known, very simple and widely used" [96]. An example event trace diagram is given in Figure 2.1. It shows the interactions between user and application (messages containing the information that needs to be passed between them) in our case study application. The scenario is one where the user wishes to view a weather prediction of a particular location and time. It is only one possible interaction scenario between the user and application and so there is no restriction that this scenario must take place or the events must occur in the order shown, only that it is possible. Requirements engineering has also been applied to agent-oriented software engineering by Yu [105], who proposes that the modelled dependencies between goals be used as the starting point for a development process.

In order to understand how all these software engineering techniques apply to agent-oriented development, we examine the essential characteristics of agent-oriented software engineering in the next section.

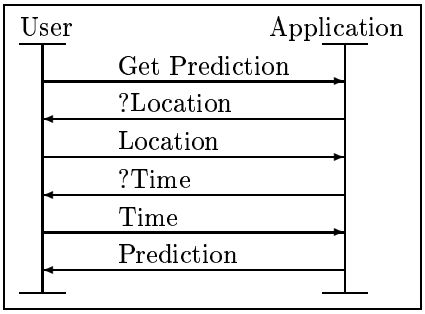


Figure 2.1: Event trace for choosing a map location prediction to speed view

2.2 Agent-Oriented Software Engineering

The agent paradigm has been used in research on all aspects of constructing applications requiring distributed autonomous activity. Recently, it has been applied to software engineering as a design abstraction [54, 100]. The approach is promoted as introducing intuitive methods for the creation of applications implemented as multi-agent systems, but it is also a useful software analysis technique in its own right (i.e. regardless of whether the implementation is perceived as agent-based). Agent-oriented software engineering (AOSE) is particularly aimed at complex, dynamic systems, such as *open* systems, where the capabilities of multi-agent systems are most beneficial [58]. This section justifies the need for an agent-oriented methodology. We describe particular existing AOSE methodologies in Section 2.8 after all the underlying concepts have been explained.

2.2.1 Agent-based Applications for Open Systems

As discussed in Chapter 1, multi-agent systems are a highly appropriate technology for applications that will run in dynamic open systems as they can, if well designed, be opportunistic, i.e. take advantage of the best functionality available at the current time. However, *designing* a multi-agent system that opportunistically provides functionality for a particular application is still a considerable problem.

Without structured methodologies, the task of constructing agent-based systems can be more difficult, more liable to fail and the eventual implementation can be less justifiable, with respect to the requirements, and less easy to maintain [29]. Therefore a structured methodology is required for developing applications for dynamic open systems.

2.2.2 Agents and Objects

Methodologies based on other technologies are not suitable for developing agent-based systems without modification [61]. In particular, agents may appear to be merely complex objects, in which case object-oriented techniques could be applied. However, the strongest beneficial property of the agent-oriented approach is the *agent metaphor* [98], which means that anyone can view the entities making up a system (design) in a comparable way as they view actors in the physical world. Objects do not fit this view without extension of the concept.

2.2.3 Formal and Informal Methods

Some work on agent-oriented methodologies has concentrated on formalising the analysis process and creating formal methods (e.g. [38, 70, 72]). In our approach, and in the review below, we concentrate on informal, structured processes for several reasons.

1. In object-oriented development, informal, structured processes are more common (and so, more *acceptable* to developers), e.g. Fusion [22]. It has been argued, e.g. by Wooldridge and Ciancarini in [101], that for wide adoption of agent technology informal development methods are needed.

2. Object-oriented notation based on the informal methods, such as UML, is more appropriate for communication of concepts to customers, and this is no less true for agents [28]. Communication is essential for ensuring customers receive what they expected and for the design process to be *visible* and *understandable*.

3. It is not possible to give a complete formal definition of behaviour in an open system, as sub-systems will be wholly or partially unknown at design-time.

Formal specification is very useful in representing and verifying applications, but formalising a solution to our problem is beyond the scope of this thesis. Information on formal methodologies is given in the review in [101].

2.3 Application Structure

Before we review the existing AOSE methodologies we have to provide the basic concepts upon which they are built. This section examines the systems in which agents operate while subsequent sections look at the behaviour of the agents themselves.

2.3.1 Domain Elements

A methodology may be wide or narrow in the scope of domains it attempts to be useful for. As discussed in Section 2.1.3, *generality* is a principle to be aimed for, and processes such as Fusion [22] try to be widely applicable. However, most important domains will have techniques tailored to designing suitable applications on top of the generic ones. For example, in a manufacturing domain, manufacturing machines may have a standard representation for ease of development and communication of designs.

In defining an initial design structure, the designer may decide to mimic the organisation of the application's domain. However, Zambonelli et al. [107] argue that this is not always appropriate for several reasons.

- The real-world organisation itself may be badly structured.
- The existence of the application may alter the way the real-world organisation works.
- The efficiency and other issues which cause a preference of one organisation over another may be different for the real-world organisation and the application.

Because of this, several alternative ways of structuring applications have been developed, and are discussed over the next few sections.

2.3.2 Shehory: Software Architectures

Software architectures are abstract structures defining (at a very high level) the whole of an application. This approach has been applied to agent-based systems by Shehory [91], where three multi-agent system designs are examined to establish important attributes for comparison. The following attributes are identified in [91] by Shehory, and used to compare agent-based system architectures.

Agent Internal Architecture is the structure of the agents within the system, i.e. how each individual agent operates internally.

Organisation is the structural form of how agents relate to and exercise control over each other. We will examine this further in Section 2.7.

System Openness is the ability of the infrastructure to operate in dynamic open environments.

Infrastructure Services are the functionality provided by the system for identifying agents, locating agents, providing security etc.

System Robustness is the ability of the system as a whole to address failures by parts of the system.

Code Reusability is a the ability for parts of the system to to be replicated and extended within other systems with minimum modification.

Being able to explicitly specify how an architecture should instantiate the above properties (and others) in a design process gives designers more scope for making justified design decisions.

The role of the infrastructure in a multi-agent system is to permit the collaborative activity of the agents to take place regardless of how the agents change over time [66]. It will be useful in the rest of this chapter to have a working definition of the infrastructure of agent-based systems (corresponding to the Infrastructure Services, Organisation and the Agent Internal Architecture above) separate from the other software architecture aspects which describe the use or user-perceived properties of that infrastructure.

Multi-Agent System Infrastructure A multi-agent system (MAS) *infrastructure* is the combined implemented elements which comprise and support agents in a multi-agent application.

As discussed earlier in this chapter (Section 2.1.3) it is, for many reasons, important to *separate concerns* when developing an application, i.e. divide the task along meaningful lines. For instance, this makes design decisions easier, allows development to be distributed over a group of designers and aids correction in the design process by reducing the dependencies between parts of the application. Agent-based infrastructures can be divided

into separate parts, often represented as layers. Work on the modularisation of agent-based infrastructures is discussed below.

2.3.3 Gasser: MAS Infrastructure Needs

Gasser [41] makes a thorough analysis of the elements, services, capabilities and attributes that are essential or desirable for multi-agent system (MAS) infrastructures. Thirty-seven "MAS infrastructure needs" are identified and divided into five categories.

System Elements Developers need tools for creating multi-agent systems. These include components such as methodologies, development environments, infrastructure frameworks (for example, RETSINA [94] or SoFAR [78]) and agent communication languages. Agent communication languages are structures that contain content which one agent wishes to pass to another and are discussed further in section 2.4.1.

Active Services Gasser argues that there is much need for independent on-line services used by multi-agent systems on demand, such as services for certification of systems for security, inter-system resource discovery etc.

Capabilities For continued reliability, management services are needed for assessing the running of systems, experimenting on it and reporting on its activities.

Attributes of Elements/Services/Capabilities Useful attributes of multi-agent systems are factors such as robustness, scalability, usability and visibility, as discussed in Section 2.1.3.

Other Community support, including open source projects and user groups, is also identified as desirable.

The areas of community support and active services are of significance to particular multi-agent system infrastructures and develop from (after) the designs of those infrastructures. The other three categories (system elements, capabilities and attributes) need to be considered *during* the design of each infrastructure or application and therefore are necessary considerations for a methodology. The system elements and capabilities are implemented for execution in known operating environments of computer systems. They can often be described as layers built on top of the operating environments. This is discussed in the next section.

2.3.4 Sycara et al.: Infrastructure Layers

Extending the modularisation of infrastructure *system elements* and *capabilities* discussed above, Sycara et al. [94] use their experience with the implemented RETSINA infrastructure to identify nine infrastructure layers that they consider necessary in a fully functioning application. The layers are applicable on the level of the whole application as well as for individual agents, e.g. *management services* are required to analyse the performance of the application as a whole and to discover the behaviour of individual agents. The full set of layers, from furthest from the operating environment to nearest, are as follows.

Interoperation Agents and multi-agent systems must be able to interact and correctly interpret messages sent between each other.

Capability to Agent Mapping There must be implemented methods to match those agents requiring services (abilities to achieve goals or perform tasks) to those providing them.

Name to Location Mapping Facilities must exist to discover the presence of agents in the system (regardless of the services they provide).

Security Services should exist to allow safe, reliable access between multi-agent systems in an open system and to prevent misuse of resources.

Performance Services Monitoring of system performance allows for reaction by the system to improve subsequent performance.

Management Services Services should be implemented to analyse the performance of the application as a whole and to discover the behaviour of individual agents

Agent Communication Language Infrastructure A structured language in which agents can communicate is required, along with the implementation of mechanisms to construct messages in that language.

Communication Modules Communication modules provide functionality, independent of particular agent communication languages, to allow communication of messages between agents.

Operating Environment The infrastructure is based upon an underlying set of functions for interacting with the computer system.

All of these layers should be addressed in the methodology. The lowest layer, operating environment, is, however, likely to be constrained by the application domain.

2.3.5 Logan: Classifying Agent Systems

Once modularised, the designer may be able to instantiate the parts of the infrastructure by several means. For example, there are different agent communication languages to choose between and different ranges of management services which could be supplied. In [68], Logan develops a classification of agent systems. The example applications of the classification are system designs where the agents are already identified, such as agents controlling robots or agents retrieving and notifying users about email, rather than those applications where the agents are to be identified from the requirements. The classification is proposed as an aid to choosing tools for implementing the designs. The factors are grouped into four categories.

1. Properties of the *environment* (domain) that the system acts within.
2. Properties and types of *actions* that the agents can perform.
3. The form of *goals* that agents may possess and how they appear in the system (are generated).
4. Properties of the *beliefs* (knowledge) that agents may possess.

Each of these is subdivided into several different factors which have qualitative or quantitative values for each different system (or the proposed model for each system). For example, the environment properties identified are classified by the following factors.

Observable An environment is *observable* if it is possible in principle for the complete system state to be determinable by the agents at any time.

Dynamic An environment is *dynamic* if the environment is not fully controlled by one agent.

Deterministic An environment is *deterministic* if the future state can, in principle, be predicted from its current state.

Discrete An environment is *discrete* if the information agents receive and the changes they make are distinct and clearly defined.

Multi-Agent An environment is *multi-agent* if there is more than one agent in the system.

Our problem domain, for instance, is an open, dynamic multi-agent software system. Therefore it would be classified as *dynamic*, *discrete* (because it is a software system) and *multi-agent* but only *partially observable* and *non-deterministic* (because the system is open and dynamic). Regardless of the completeness of the classification, the idea of most significance in this approach is to be able to select between comparable *models* for infrastructure modularised in the ways described above.

2.4 The Coordination Problem

While the decompositions of multi-agent system infrastructure described above are essential for structuring and separating concerns within a design, they do not allow designers to describe how agents will interoperate at run time. In order for an application to be opportunistic it must interact with those services it wishes to take advantage of. Agents are, by definition, *social* and so are specifically designed to interact. For agent interactions to best meet the needs of the users, i.e. meet the application requirements, the agents need some control over how the interactions take place. This is the problem of *coordination* and the infrastructure parts concerned with agents' control over interactions are called *coordination mechanisms*. In order to discuss coordination mechanisms further, we give an informal definition. The definition is deliberately broad, to allow a range of approaches to be included.

Coordination Mechanism A coordination mechanism is part of a multi-agent system infrastructure which enables agents to cooperate in achieving the application requirements. A coordination mechanism assumes that a suitable means of communication between agents exists.

If a designer wishes to compare two or more coordination mechanisms, it will be easier if they are expressed in the same form. To generalise over multiple mechanisms, the designer needs more abstract forms of description than the description used to define an individual mechanism. Research has been done in the area of comparison of coordination mechanisms and this is described later, but we first examine the underlying communication between agents that allows coordination to take place.

2.4.1 Agent Communication Languages

Underlying all deliberate coordination between agents is some form of communication. Typically, agent communication is in the form of asynchronous messages sent over some transport mechanism. In order for one agent to be able to interpret the communications from others, the agents utilise a mutually interpretable *agent communication language* [67, 92]. Petrie [87] has argued that these languages are one of the fundamental defining features of agent based systems. We define an agent communication language as follows.

Agent Communication Language An *agent communication language* (ACL) is a structured, pre-specified language used to encode messages between agents. Agent communication languages are often divided into an outer language for communicating the *pragmatics* of messages such as the intended receiver, and an inner, content language for communicating *semantic* information.

Agent communication may be specific to the application being developed. For example, in an application controlling the manufacturing of toys, the language may be composed of simple messages such as *MoveToyOntoConveyorBelt (X)* or *StickArmsOntoDoll (Y)*. In such languages, the intended effect of the message, and possibly the recipient too, are implicit in the particular message itself. In agent-based research, more general languages have been proposed. General agent communication languages allow far more expressiveness and scope for extension than domain-specific ones. Furthermore, they can be reasoned about without reference to the domain, making them important for the analysis of communication [30]. The most prominent of the general agent communication languages are KQML [36] and FIPA-ACL [81], the former due to its wide use in agent research projects and the latter because it is an explicit attempt at a standard agent communication language for general use. These are 'outer' languages, as mentioned in the definition above, as they are used for explicitly stating the intended effect and message recipient and not the information content of the message.

KQML

KQML (Knowledge Query and Manipulation Language) is an extensible LISP-based language [36, 37, 75]. In KQML messages are defined by one of the pre-defined *performatives* that define the message effect and several other elements (parameters) including the content

of the message. For example, a simple message representing a query about the location of an airport could encoded as follows (taken from [36]).

```
(ask-one    :content (geoloc lax (?long ?lat))
            :ontology geo-model3)
```

This message is asking for one reply to a query (the performative 'ask-one') with the query expressed in a particular content language asking for two pieces of information. The 'ontology' element specifies the set of semantics which an agent needs to be able to interpret in order to use the message, in this case the ontology is called 'geo-model3' and will include an interpretation of 'geoloc lax (?long ?lat)'. Other KQML performatives define messages as requesting information in a variety of forms ('evaluate', 'ask-one', 'ask-all' etc.), responding ('reply', 'sorry'), knowledge exchange ('tell', 'untell' etc.) and performatives allowing agents to communicate with an infrastructure associated with KQML.

FIPA-ACL

FIPA-ACL (the latest specifications are available from [81]) is intended to improve on KQML and provide a standard message structure so that multi-agent systems created by different groups can easily connect and communicate with each other. This simplifies the interoperation in open system infrastructures. FIPA-ACL gives a definite list of possible parameters (elements) making up a message such as the example above.

performative The type of the message, suggesting the intention of the message, e.g. ask-one in the example above.

sender The agent sending the message.

receiver The agents that the sender intends to be recipients of the message.

reply-to The agent that replies should be sent to.

content The informational content of the message.

language The content language used for the informational content.

encoding The particular encoding of the content language used.

ontology The ontology specifying the interpretation of the content.

protocol The interaction protocol that the message is being sent as a part of. For instance, if agents are exchanging contracts which bind them to performing given tasks then the protocol element may be 'contract-net'.

conversation-id An identifying code that allows agents to keep track of the messages in a conversation between two (or more) agents on a single topic.

reply-with An identifying code that the receiver is instructed to use in replying so that the sender can identify its topic.

in-reply-to An identifying code sent in response to the value of a previous *reply-with* field.

reply-by A specification of the time by which the sender must receive a reply.

There is no requirement that all of these elements must be used in every FIPA-ACL message. There are a large number of FIPA-ACL performatives specified [81], though a particular application would only use some subset of them. The syntax is very similar to KQML. For example, the message below (taken from [81]).

```
(accept-proposal   :sender (agent-identifier :name i)
                   :receiver (set (agent-identifier :name j))
                   :in-reply-to bid089
                   :content ((action (agent-identifier :name j))
                   (stream-content movie1234 19)
                   (B (agent-identifier :name j)
                   (ready customer78)))
                   :language FIPA-SL)
```

FIPA-ACL has been incorporated into some programming frameworks for development of agent systems, e.g. JADE [9]. A review of more specific details relating to agent communication and agent communication languages, beyond the scope of our study, can be found in [30].

2.5 Centralised Coordination

Coordination between agents can be achieved by the use of many types of coordination mechanism, and we will examine these over the following few sections. One way in which

agents in a multi-agent system can coordinate is through using a central *broker agent*. Brokers enable coordination by performing functions such as matching up the most suitable service providers to those agents requiring services. Such centralised broker agents define the scope of an application or a set of applications by limiting the set of agents to those registered with the broker agent [99]. A broker agent is likely to merge functionality for agents to discover each other (name-to-location mapping) with coordination functions.

Centralising coordination has benefits in reducing the amount of work individual agents have to do in order to cooperate, but reduces the ease with which an application can be modified at run time (for maintenance and extension) and may restrict agents' choice in coordinating in different ways for different activities, e.g. an agent may most suitably use different coordination approaches for secure, sensitive transactions and more trivial operations.

2.5.1 Broker Agent Selection

In order to choose between coordination mechanisms to use, the designer needs some criteria for comparison. In [65], Klusch and Sycara compare several existing *broker agents*. Broker agents are coordination mechanisms that involve one or more agents (brokers) holding information on services provided by other agents (providers) within the application. Agents can then identify providers of services they need by asking the broker agents. The set of requester agents and the set of provider agents may overlap. Broker agents provide an implementation of the *capability-to-agent mapping* infrastructure layer discussed in Section 2.3.4 but also provide services additional to this function.

Broker agents are used in various forms and Klusch and Sycara distinguish them on the basis of the services they provide, such as how data on agents are stored or in what ways agents can interact with the broker. They name three distinct types of brokering mechanism, given below. These terms are, however, used interchangeably in the literature (along with others such as facilitator [74], arbitrator etc.)

Mediator A mediator agent integrates the services and data advertised to it by agents. When an agent requests a service, the mediator can produce a packaged solution composed of the services provided by several others. A mediator acts as an arbitrator between requesters and providers of services possibly including translation if they use different communication languages.

Broker A broker agent takes requests and passes them directly on to others providing the relevant services, results again returning via the broker agent. This preserves the anonymity of the agents and is more simple and realistic for highly dynamic systems than a mediator agent.

Matchmaker A matchmaker agent is another simple solution in which requesting agents are given contact information of the providers. As with other types of broker, the matchmaker maintains a database of the services agents can perform but uses this solely to identify a suitable provider for each requester. Requesters and providers then communicate without any further intervention by the matchmaker which allows for greater flexibility.

To ease the choice between the broker types, compact descriptions are provided by Klusch and Sycara. They describe each mechanism in terms of four aspects

1. The *pattern of interaction* that describes how the mechanism is used.
2. The *signatures* (structures) of the messages that the broker accepts and returns.
3. The data structures holding the information required by the mechanism, which they call *states*.
4. The *transitions* that occur when the broker receives each of the acceptable messages, e.g. adding a service description to the database.

This decomposition of properties allows for easy comparison of the brokers, but it is not made clear how the broker aspects are to be reconciled with an application's requirements. In developing a justified application, this is information that the designer will need. We discuss this further in Chapter 5.

2.5.2 Semi-Centralised Coordination

It is also possible to provide coordination through a hybrid of centralised brokering and distribution of control [12]. In such a system, the system is divided into several communicating sub-systems each with centralised control. This has some of the benefits of the centralised approach in having known locations at which to access information but with the fault tolerance of the distributed approach, though it also shares the costs of both approaches by

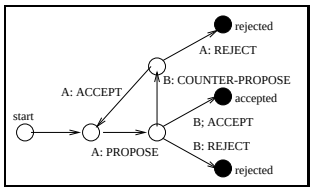


Figure 2.2: Interaction protocol describing agents A and B interacting to find a mutually acceptable proposal

maintaining several potentially large, inter-communicating stores of coordination information.

2.6 Distributed Coordination

An alternative to broker agents is to provide each agent with their own coordination mechanism which they use to decide what to communicate directly to others in order to cooperate.

2.6.1 Finite State Machines

One way of viewing coordination is as the result of *interaction protocols*. These protocols define the communications that are passed between agents as they cooperate in a particular way. For example, an interaction protocol may be initiated by one agent delegating a task to another agent by communicating an order for the task to be completed. The protocol then requires the agent charged with the task to acknowledge its receipt of the order and later to communicate its successful completion of the task. Interaction protocols are often described using *finite state machines* which show how the agents involved in the interaction change from one state to another as communications are passed between them. Figure 2.2 shows a finite state machine describing an interaction protocol loosely based on an example from [6]. It shows how agents A and B can change the state of the interaction by sending communications proposing and counter-proposing solutions to a goal and then accepting or rejecting them. Agent A starts the process and provides the final proposal but B may reject the cooperation. The state labelled 'start' marks the beginning of the protocol's use and the states marked 'rejected' or 'accepted' conclude the protocol's use.

In the simple delegation example given earlier, the delegating agent transforms from having not delegated the task to waiting for an acknowledgement (after it sends the del-

egation message) and then to waiting for the notification of success (after if receives the acknowledgement message from the other agent). An equivalent description is given by adapted UML sequence diagrams in the Agent UML (AUML) protocol diagrams proposed by Odell et al. [7, 79, 80].

2.6.2 Commitment Machines

Yolum and Singh [103] enhance the idea of interaction protocols to provide them with more flexibility and bring them closer to being fully functional coordination mechanisms, where agents have more control over how and what messages are passed. They do this by modelling interaction protocols as *commitment machines*. A commitment machine is described by a set of states that the agents can be in, as with interaction protocol finite state machines. However, each of the states is defined only by the *commitments* which each agent has bound itself to. A commitment is a pledge to undertake a specified course of action [59]. Associated with the set of states are actions that could take the agents from one state to another. In the simple delegation example, the action of the agent acknowledging the delegated task would take the interaction to a state in which the delegated agent has the commitment to the delegated task. The interaction ends when one of the states marked as 'final' is reached. Commitment machines are more flexible than interaction protocols because they do not demand that particular actions are taken to move from one state to another, or at what state the interaction begins. For example, if an agent knows that another possesses a goal that it cannot complete itself, then it could send an acknowledgement committing itself to take on the goal without first receiving a command from the delegating agent. Commitment machines can generalise the representation of a range of coordination mechanisms based on commitment exchange.

2.6.3 Dynamic Selection

A possible alternative to using a single coordination mechanism per agent is to allow agents to decide between mechanisms at run time. This dynamic selection of coordination mechanisms is suggested by Excelente-Toledo et al. [33, 34] as a way of ensuring that coordination between agents adapts to the current system state. The agents in the system they propose choose between commitment-based coordination mechanisms when they need to cooperate and decide upon whether or when to drop commitments. Coordination mechanisms are

described declaratively by a tuple of two measures (t, p), where t is the number of time steps required to set up the mechanism and p is the probability of eventual success in the coordination. Along with values for the degree of commitment which agents have (probability of *not* dropping a commitment) and the penalties for dropping commitments, the agents can decide on *bids* for taking on cooperative, and non-cooperative, tasks. The highest bids are chosen by the agent managing the completion of each task. By the absence of any other measures, it is implied that the requirements of the multi-agent system as a whole are completely met by the successful completion of the tasks in as short a time as possible.

2.6.4 GPGP: Modularising Coordination Mechanisms

Particular forms of coordination mechanism can be used for domains defined as *worth-oriented*. In these domains, there is similarly a single monetary-style metric (utility) which can be used to compare suitable tactics at run-time, but the agents actively cooperate in order to maximise the overall system utility. The Partial Global Planning (PGP) [32] algorithm is one example of multi-agent systems based on this cooperative model, in which agents possess partial views of plans to achieve their goals, due to the limited perspective they have on the system. The agents can then communicate between themselves in order to find better solutions to their goals with more complete information. This was later extended by Decker and Lesser [27] into Generalised Partial Global Planning where the PGP algorithm is divided up into several coordination mechanism parts that can be used or not used as the agent considers most appropriate at the time, as well as extended and added to. The coordination mechanism parts allowed agents to do the following.

1. Update other agents on the ways in which tasks could be best completed, from each agent's local perspective.
2. Communicate results of completed tasks.
3. Handle redundancy of effort of the system of agents by informing others of commitments made to perform tasks.
4. Where one task must be completed before another task can be completed, an agent can commit to, and communicate the commitment to, the 'enabling' task.

5. Where one task helps in completing another task or improves the worth of its solution, an agent can commit to, and communicate the commitment to, the 'facilitating' task.

2.6.5 Barber et al.: Activities Decomposition

Barber et al. [5] also propose shifting the decision of which coordination mechanism to use to the agents, though they allow their decision making procedure to be used by the designer as well. This allows agents to use whichever mechanism is most suitable for the current situation. The mechanisms the agents can choose between are encoded as plans, known by the agents. They suggest dividing problem solving into the five activities given below, and analysing which mechanism would be best for each.

Agent Organisation Construction specifies how the agents should interact with one another.

Plan Generation works in the organisation decided by the agent organisation construction, selecting the actions or subgoals that the agents must execute to accomplish their goal.

Task Allocation deals with the assignment of actions or goals to specific agents for execution or further planning.

Plan Integration joins the sub-plans and schedules from the previous two steps to coordinate agent actions.

Plan Execution deals with monitoring the execution of each agent's schedules to insure that actions are performed as expected from the plan integration.

Each agent involved in an interaction communicates with others during this activities. The most suitable coordination mechanism is chosen by examining the following factors.

1. The structural requirements of the mechanism, e.g. decision making abilities for the agents using the coordination mechanism.
2. The resource cost of using the strategy.
3. The *quality* of solution that the mechanism produces.
4. Application domain requirements.

Barber et al. assess several coordination mechanisms (self-modification, voting, negotiation and arbitration) for cost in terms of time to execute and number of communications used, but do not specify what criteria these factors should be compared against to decide on a suitable coordination mechanism, or what defines the *quality* of a solution in a given application.

2.7 Organised Coordination

Distributed coordination allows for great flexibility but relies on agents having a large amount of knowledge about each other in order for the MAS application as a whole to achieve the best results it can. To limit distributed coordination without centralising it, agents can be organised so that activities over which they coordinate are restricted without necessarily restricting the coordination activity, including the agents with which agents could be involved in cooperation.

A useful way to model the intended restrictions on and expected interactions of an agent is in terms of the *roles* that it plays [102]. A role is an abstraction from any one agent, defining a subset of behaviours. It is based on the positions that people may take in a human organisation, e.g. manager, road sweeper etc. A role may have associated responsibilities, in which case ensuring every role in an application is being played by at least one agent at any time should thereby ensure the application will achieve its requirements (to some quality). Various ways of representing roles and groups of agents through extending UML have been proposed by Parunak and Odell [86]. Broadly, roles are used as specification of behaviour at a more abstract level than that of agents. This level, called *the social level* is discussed below.

2.7.1 The Social Level

The opportunism of an application depends on it being made up of a set of loosely-coupled agents. This emphasis on collective ability has led some researchers to state that an analysis of the collective form of the agents is essential [85]. The group of agents comprising an application is often called a *society* or *organisation*. In [57], Jennings terms this level of analysis the *social level*, and it is argued that by specifying the behaviour expected from this level, the unpredictability of the application due to run-time agent flexibility can be

reduced.

Organisations provide a powerful means of communication between designers and users. A MAS application structure can be described as being comparable to a human organisation. This allows users to see a succinct presentation of how the application is to work. Additionally, the use of roles to describe responsibilities and rights of users and agents is compatible with common security models such as *role-based access control* [90].

2.7.2 Dignum et al: Coordination Models

A different view of coordination is taken by Dignum et al. In [31], they take the *coordination model* to be the primary component of analysis. A coordination model is an application-wide coordination mechanism limiting agent interactions to those applicable to the model definition. For instance, agents in a market coordination model can interact by bidding for services. Dignum et al. outline a methodology that proposes deriving the coordination model directly from the application domain. For example, if the agents within the application will be in competition for resources (such as in an auction) then they use the 'market' coordination model where coordination is limited to using *matchmakers* to match potential suppliers and providers. Alternatively, in a domain where some parts have authority over others, the 'hierarchy' coordination model is used with different structural needs. This approach is similar to beginning analysis by deciding on an *organisational structure*, only with more detail on the coordination mechanisms used. The coordination in the models is mostly provided by other agents. The models identified and the agents providing coordination in each are given below.

Market In the market coordination model, *matchmaker* agents match suppliers to providers, *reputation* services keep track of the reliability of trading agents for security and *banking* facilities deal with valuation of goods and currency issues.

Network Networks are used to provide shared context to a group of agents aiming to achieve a mutually possessed goal. Contracts are exchanged between agents, and norms (rules restricting agent actions) are enforced. *Matchmakers* are used as in the market, along with *gatekeepers* to vet agents wishing to join the network, *notaries* keep track of contracts between agents, and *monitoring agents* are trusted witnesses for the contract that check that contracts are obeyed.

Hierarchies Coordination models in which some agents 'manage' the system as a whole, and the suppliers of services are pre-determined are called hierarchies. The managing agents are called *controllers* while agents that communicate between the system and other systems are called *interface agents*.

2.7.3 Shehory: Organisational Coordination Models

The classification above is mainly based on the purpose of each coordination model (competition, shared goals or organised distribution of work). This reflects the aim of choosing models to suit the domain. Shehory [91] uses a classification of coordination models (organisational structures) based on the relations between agents. Again, a hierarchy is identified as a potential model with fairly centralised control. Shehory adds flat organisations where each agent interacts freely with all other agents, subsumption models in which some agents are components of other agents and modular organisations where the multi-agent system is composed of several loosely coupled, smaller multi-agent systems. The difference between these models is mostly a matter of degree to which agents are constrained in their actions by other agents.

2.7.4 Adelsberger and Conen: Encouraging Coordination

The use of economic evaluations for goals is used by Adelsberger and Conen [1]. A dynamically changing coordination mechanism sets the (simulated) monetary value of resources at run time so as to encourage coordination between self-interested agents. The mechanism implements this by bundling disparate resources together which are then bid for and allocated to agents. However they point out that the applicability of negotiation on economic measures is restricted to a subset of applications where "...it is reasonable to relate individual objectives to valuations.", i.e. where the goals of the application can be assigned comparable numeric values that signify their relative worth with regards to the requirements.

2.7.5 Flow Control

Another way in which to model coordination is by the path of interactions between agents and the decisions which affect those paths. Such a path is called a *workflow*. Agents in the workflow are points at which decisions are made. Such a model describes the alternatives

open to an agent in terms of the choices of which agent(s) will be communicated with next in order to achieve a task. The domain, therefore, must be pre-defined and so a closed system in the general case.

2.8 AOSE Methodologies

Once we have understood the infrastructure and coordination mechanisms required to instantiate agent-based applications, as described in the sections above, we are in a better position to judge the effectiveness of existing agent-oriented methodologies. This section describes several such methodologies. For brevity, the methodologies are divided into groups below, corresponding to each subsection, where members of each group share a common broad approach. For each group, we describe the approach of one prominent methodology and then provide details on the similarities and differences with other closely-related methods. We examine the advantages and disadvantages of each in relation to our particular problem (justified design of opportunistic applications in open systems) in the next chapter.

2.8.1 Gaia: Role-Based Design

The Gaia methodology [102] uses organisational roles as the primary modeling concept. The early form of Gaia, as given in [102], is intended only for closed systems (we discuss improvements to Gaia below). In Gaia, the system is analysed as an organisation from the outset and roles are directly recognised from the requirements during the analysis phase. Roles in Gaia are defined by four attributes:

Responsibilities specify what the agent fulfilling the role must attempt to bring about and prevent happening in certain conditions;

Permissions specify the system resources an agent can and cannot use;

Activities specify the actions an agent can perform without involving other agents;

Protocols specify what an agent can achieve collaborating with other agents.

Gaia is a full methodology with analysis and design stages consisting of several models derived from the requirements. In the analysis phase, roles are derived and the relationships between those roles are modeled in the Interactions Model. The design phase then

derives implementable structures such as agent types (aggregations of several roles), services (encoded sub-agent functionality which agents can execute) and acquaintances (necessary communication links between agent types).

2.8.2 Tropos: Requirements Engineering in Design

Tropos [16, 20, 45] is an agent-oriented methodology based on requirements engineering techniques (specifically i* [104]). At the start of the Tropos methodological process, the requirements are examined and the roles of the (human) organisations providing services to or requiring services from the application. These roles are distinguished from those in Gaia in that they do not suggest the structure of the application, but only the structure of the domain. The goals of each role with respect to the application are then identified. This model of roles with goals is transformed, by a series of steps including decomposing goals into sub-goals, into a set of agents with plans on how to enact the roles.

A comparable approach is EXPAND [15] in which a model is identified that includes the agents external to the application and their *expectations* on the system behaviour. Expectations are properties which the users (modelled as agents) believe should be manifested in the application, e.g. that responses should be received when commands are given. A multi-agent system is then implemented in which the agents follow rules (norms) that restrict their behaviour. The system is analysed to see whether the emergent behaviour matches the agent expectations This model is then iteratively transformed towards a form better matching the expectations, by alteration of the norms based on repeated analysis.

2.8.3 Bussmann: Domain-Based Decision Points

Some methodologies emphasise the decision-making capabilities of agents. Due to their flexible, reactive nature, agents can make timely decisions based on the current state of the system. Agents can be used, therefore, as points in an application's workflow where decisions are made. For application domains in which the workflows are obvious, decision points may be a useful modelling concept. In [18], Bussman et al. suggest that analysis of production control systems begins with identification of decisions to be made, e.g. which conveyor belt a work piece should be shunted onto given the current load on each. These decisions are specified in schemas then the dependencies between decision points are identified. Agent

roles are derived from the decision points in the way which best separates resources (sensors and effectors) and responsibilities and reduces dependencies.

2.8.4 Kendall et al.: Manufacturing Workflows

Similarly, in [64], Kendall et al. derive agent models for enterprise integration using workflow models. These express the decisions which need to be made in the application, and the actions which then transform input to output at that decision point. Use case diagrams are used to identify the different contexts within which the decisions are made. By decomposing the decisions into influencing factors (contexts), plans can be derived for agents at the decision points to execute when a decision needs to be made.

Use Case Maps (UCMs) is another approach in which agents are 'plugged in' to a design in the form of a workflow [17]. UCMs allow workflows to be recursively designed so that the features of an application, and interactions between agents, can be analysed on various levels separately. Depke et al. [28] model the flow in a system in terms of *transformation rules* taking the system from one state to another. Each state is described by the relations between agents and other objects. The transformation rules are then used to develop a design in which the rules are implemented. An alternative to workflow is to describe the system in terms of flow of data. This approach, taken for example by Prometheus [84] places similar demands on the designer: modelling of a flow through business processes, agents at points where decisions need to be made and explicit specification of the relations between agents.

2.8.5 Yu and Schmid: Workflows with Roles

The design framework described by Yu and Schmid in [106] represents a workflow as a collection of agents which control separate activities and interact whenever their are interdependencies between the activities. The activities are treated as agent roles which decision making agents will fulfill.

Decision points and workflow models are primarily aimed at closed systems because, while data may flow between decision points and unspecified sources, functionality is either specified within the application or not specified at all.

2.8.6 SODA: Society-Based Design

A group of interacting agents or agent roles performing an activity represents a more flexible concept than a role in a design. This is due to the fact that agent groups can be loosely coupled providing weak interdependencies and a natural way of distributing the agents. By distributing agent groups, activities can be spread over more than one subsystem so open systems can be exploited. SODA [82, 83] takes this approach, analysing the system in terms of *societies* of agents. Tasks identified from the requirements are mapped to individual agents as role responsibilities, or to groups of agents. Tasks to be performed by groups are called *social tasks* and the roles assigned to agents within the group are called *social roles*. Groups are modelled as *societies* of agents each with their own coordination model to best accomplish the social task. SODA also analyses the environment to the same degree as the agent societies, developing *infrastructure classes* that provide access to system resources, possibly distributed over an open system.

While agent societies solve some of the problems with developing organisations for open systems, they require a mapping from roles/societies to agents at design-time so the agents involved must be known in advance, even if they are in a connected subsystem. This type of domain is classified as *static open systems* in the classification given by Shehory [91]. An application for a dynamic open system allows for exploitation of agents that are only connected or created during run-time.

Dignum et al. [31], similarly, allow designers to decide on a coordination model for a society based on the events that will be received by the society's environment and the society's overall purpose.

2.8.7 Zambonelli et al.: Organisational Rules

Another way of using a collection of agents as a more primitive concept than individuals is by specifying rules which must hold between them. This approach allows the designer to leave fully specifying agent roles (and interaction protocols) until after a set of organisational rules have been identified [108]. The system is then tailored to obeying these rules which makes the design open to changes to the organisation while still obeying the organisational rules, so the design may have less arbitrary restrictions than if the roles were derived directly from the requirements.

The design approach described by Zambonelli et al. [108] is primarily concerned with limiting agents' actions, i.e. organisational rules limit potential role activities. An alternative approach, given by Ciancarini et al. in [21], advocates using rules to control the access to central data stores, which has a comparable end result but with potentially better scaling.

Zambonelli et al. identify three organisational concepts and use them to extend Gaia.

Organisational Rules assert system requirements in terms of the relation between other analysis concepts such as roles, protocols and agents. For example, it could be expressed that only one agent at a time is allowed to use a particular resource, i.e. adopt the role related to accessing that resource. In [108], Zambonelli et al. specify organisational rules in temporal logic.

Organisational Structures specify the way in which roles in the organisation will be related in general terms, e.g. "a multi-level hierarchy based on a work partitioning control regime at the highest level and on a global coordination control regime at the [lower] level." [108].

Organisational Patterns catalogue generally useful organisational structures for re-use in the same way as object-oriented design patterns [40].

In the extension of the Gaia methodology, a preliminary set of roles and protocols are first identified from the requirements. Organisational rules are then identified that restrict these roles and protocols. The rules are used to identify organisational structure, provide further design detail to the preliminary roles and bring the design towards implementation.

As open systems contain agents that may not be completely constrained by rules, it may also be useful to take an approach to modelling organisations where the autonomy of agents over the rules are taken into account [69]. Automating the process of extracting rules from descriptions of the intended organisation has also been suggested [97].

2.9 Summary

In order to solve the problem of creating justified designs consistently and with wide applicability, a software engineering methodology can be used. A methodology should follow certain

principles to solve the common problems that occur in and after development. Re-use of designs in object-oriented software engineering is facilitated by design patterns. Agents are a promising technology for open system applications and agent-oriented software engineering has useful characteristics distinct from object-oriented development. A structured, informal methodology would be most suitable for solving our problem.

We argue that existing agent-oriented methodologies do not allow designers to create justified, opportunistic designs. This is either because they fail to consistently provide a connection from design decisions that need to be made to the requirements, or because they deny the designer the means to express interoperation with services that do not exist in the open system at design time. This argument is presented in full in the next chapter.

In this chapter, we have examined various aspects of multi-agent systems in order to understand what a methodology must provide. The infrastructure underlying a multi-agent system can be modularised in various ways, though structuring it to match the structure of the domain is not always justified by the requirements. To aid design, infrastructure parts can then be classified into *models* which can be chosen between (rather than developing part of the infrastructure from scratch).

For MAS applications to be opportunistic, the agents must be able to coordinate their activities. This can be done using centralised brokers, commitments, structured interactions within organisations etc.

In this chapter we have introduced several existing methodologies. In the next chapter we analyse them to discover in what ways they succeed and fail in producing justified designs for opportunistic applications, and use this analysis to inform our solution.

Chapter 3

Agent Interactions as a Modelling Concept

In this chapter, we discuss what obstacles there are to designers consistently producing justified designs for open system applications, compare the approaches described in the previous chapter and introduce our own methodology based on agent interactions.

We would like to be able to judge whether, in general, a methodology aids a designer in creating justified, opportunistic designs. To do this, we have to consider the *capabilities* provided to a designer by a methodology. Section 3.1 examines the way in which we can judge the capabilities of a methodology to produce justified designs and opportunistic applications. In Section 3.2, we use the specification of these methodology capabilities to evaluate how well the existing approaches discussed in the previous chapter resolve the problem of producing justified, opportunistic applications. The results are analysed in Section 3.3, and a new approach is derived in Section 3.4. The chapter is summarised and the structure of the remainder of the thesis outlined in Section 3.5.

3.1 Methodology Capabilities

If the designer is able to create consistently justified designs using a methodology, we say that the methodology has the *methodology capability* of producing justified designs. Similarly, if the designer can consistently create opportunistic applications, then we say that the

methodology has the methodology capability of producing designs of opportunistic applications.

In practice, determining whether a methodology has a methodology capability of producing justified designs or of producing designs of opportunsitic applications, is made difficult because of the broadness of the topics. Therefore it is useful to break down the methodology capabilities into sub-capabilities which are easier to check for. In the sections below, we consider useful ways to decompose the methodology capabilities of producing justified and opportunstic designs.

3.1.1 Producing Justified Designs

Following the idea of *traceability*, i.e. being able to trace back from any design decision to the requirements of the application [80], we take an inductive approach to decompose the methodology capability of producing justified design. We argue that, for a justified design to be produced, the methodology must meet two criteria. First, the methodology must consistently allow the designer to map from the requirements to the concepts and structures used to make design decisions in the methodology (requirements analysis). Second, after requirements analysis any necessary step in the methodology must be justified by the structures already identified by the design process (or by the requirements directly). If both of these criteria are met then, by induction, the design as a whole will be justified. Methodologies that meet these two criteria have the corresponding methodology sub-capabilities of *identification* and *connection* with the requirements, which are described further below.

Identification Analysis of requirements in a structured methodology involves placing information from the requirements into a standard structured form. For example, in object-oriented design, requirements are described as objects with various properties and relations between them, while Gaia uses roles and SODA uses societies (see Chapter 2 on these methodologies). With design approaches that use components with relatively complex behaviour, *identification* of those components in the requirements may be difficult. For a methodology to produce justified designs, their analysis structures should be reliably identifiable from their requirements.

Connection Design decisions should ultimately be based on requirements. For a methodology to produce justified designs, the design decisions made using it should be clearly

connected to information in the requirements.

3.1.2 Design of Opportunistic Applications

Similarly, we divide the capabilities of producing opportunistic applications into two criteria. First, an agent able to take opportunities must be *flexible* enough in its behaviour that it can decide to do so. This will depend in part on its method of cooperating with other agents, but also requires that the design has not unnecessarily limited the set of actions it can perform. The second concern is with regards to the *interoperation* between agents. A methodology for open systems must be able to express interaction between those agents added to the system by the designer and those whose design is outside the control of the designer. Methodologies that meet the two criteria have the sub-capabilities of *flexibility* and *interoperation*, which are summarised below.

Flexibility In order for an application to take full advantage of services in an open system it must be constrained as little as possible in use of services. Opportunsim within an application, therefore, depends on a design not placing unnecessary, i.e. unjustified, constraints on the agents comprising the application. For a methodology to produce opportunsitic designs, the way in which the designs are created should not add unjustified constraints to the applications.

Interoperation Opportunism requires that agents in an application have the ability to access resources, such as other agents, outside of their local domain. In order for design decisions to be made based on agent interoperation, the methodology must allow the representation of interaction between agents under design with those already existing, or existing in the future.

Other criteria can be used to judge whether one methodology is better or worse than another, such as the *separation of concerns* during the design process and the *intuitivity* of the concepts used to construct designs. These properties are described in Chapter 2 and are generally fulfilled adequately by the existing AOSE methodologies. We do not investigate whether existing methodologies fulfil these criteria here as our scope of comparison is concentrated on how well methodologies produce justified, opportunistic designs.

Methodology	*Identification*	*Connection*	*Flexibility*	*Interoperation*
Gaia	No	No	No	Yes
Tropos	Yes	Yes	Yes	No
Role Modelling	Yes	No	No	Yes
Domain Decisions	Yes	No	Yes	No
Workflows	Yes	No	No	Yes
SODA	Yes	Yes	No	Yes
Organisation Rules	No	Yes	No	Yes

Table 3.1: Open system design problems addressed by each approach

3.2 Methodologies Evaluation

Making comparisons of effectiveness between methodologies is made difficult because designers using most methodologies can legitimately arrive at different designs for the same application requirements. An informal, structured methodology only guides a designer in analysing the problem and making design decisions. Therefore, comparing the effectiveness of methodologies for creating justified, opportunistic designs must involve examining whether they have capabilities, such as those described above.

In the sections below, we examine the methodologies discussed in Chapter 2 and answer whether they possess these methodology capabilities. The results for the sub-capabilities of seven representative methodologies are summarised in Table 3.1. Clearly where a methodology does not possess a capability this may not be a shortcoming on its own terms if the capability is not relevant to the methodology's target domain, e.g. there is no need for opportunism in closed systems. The importance of the evaluation is in determining how the methodologies succeed or fail in solving the problem we are concerned with.

3.2.1 Identification

The main criteria we can justifiably adopt for deciding whether entities are consistently identifiable from requirements are whether they have been judged as such from the past experience of the software engineering community. Goals, as identified in Yu and Schmid's workflow-based methodology [106] and SODA [83] are recognised as commonly derivable from requirements [26]. Entities within the application domain, e.g. manufacturing equipment as used to illustrate Bussmann's methodology [18], are clearly identifiable as they already exist. Tropos [16] also identify goals and domain elements (the user roles, rather

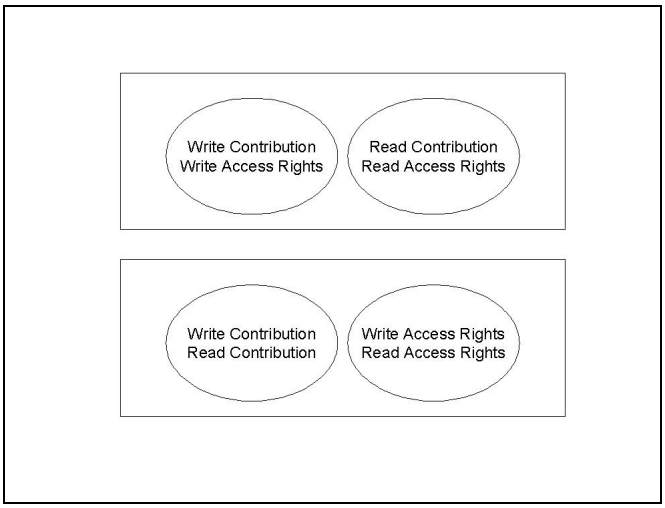


Figure 3.1: Roles: Problem with Identification

than the agent roles), using standard requirements engineering.

Directly identifying all the agent roles of an application from the requirements has been recognised as a difficulty and not always possible [18, 83]. Roles do not describe required functionality in the system and therefore there is no reason to believe they will be obviously present in application requirements in general.

In the design of agent-based applications for production control systems examined by Bussman et al. in [18] and it is concluded that it is "necessary to extend [agent-oriented] methodologies by a preceding analysis step that derives roles from the production control problem". Role modelling is used in specifying object-oriented systems, and Kendall [63] adapts this approach for AOSE, but the role modelling is based on previously identified entities such as goals. For methodologies in which roles are identified from the requirements, either with detailed properties as in Gaia [102] or in a preliminary state as in Zambonelli et al.'s extension of Gaia [107], there is a danger that the roles may not be readily identifiable and that any division into roles will be arbitrary in part, and so not justified by the requirements. Efforts have begun to address this particular shortcoming in Gaia [62].

To illustrate the above points, Figure 3.1 shows two possible ways of dividing up the functionality of the case study into two roles. In the upper organisation, there are two roles. The first is a role for an agent able to write the contributions made to the weather map and also for editing the access rights of users to the map. The second role is the equivalent for

reading weather data and access rights. In the lower organisation, there are also two roles but in this case the functionality is split between a role dealing with access rights and a role dealing with weather map contributions. Either of these organisations could have been derived from the requirements using the Gaia methodology but the decision would not be informed and could affect how well the application as a whole met the requirements. For example, the upper organisation is more secure in case of accidents because the 'writing' role can be tailored to require stringent integrity checks, while the 'reading' role would not need these checks and so could perform read operations quickly. On the other hand, the lower organisation can be more secure in terms of administration because the reading and modifying of access rights can have more security checks than reading and writing contributions. It depends on the functional and non-functional requirements to determine which of these organisations best fits the requirements, and so these should be identified from the requirements before agent roles.

3.2.2 Connection

If identification can take place successfully, the division of functionality produced by design decisions must be connected to the requirements in order for the design to be justified. Designs whose structure is based on the structure of the domain may not be justified by that mapping for several reasons described in [107]. First, the domain may itself not be well structured. Second, the creation of the application may change the way the organisation works. Third, the reasons behind the structure of the domain may not apply to the requirements of the application. These limitations apply both to designs based on mechanical domain structure (such as Bussmann's [18] and Yu and Schmid's [106]) and role-based designs where roles are derived from human jobs.

In the case where roles are not derived from the domain structure there must be some other means of attaching the division of functionality in the design to the requirements. Zambonelli et al. [107, 108] achieve this for Gaia by first extracting the simple organisational rules that apply to preliminary roles and deriving a role-based design from them. Tropos [16] uses repeated transformations, such as decomposing a goal into two parts to move from each successive model to a more detailed one.

Justification of a design with many parts is liable to be complex and produce a large amount of reasoning. SODA [83] balances the complexity of justification with an ease of

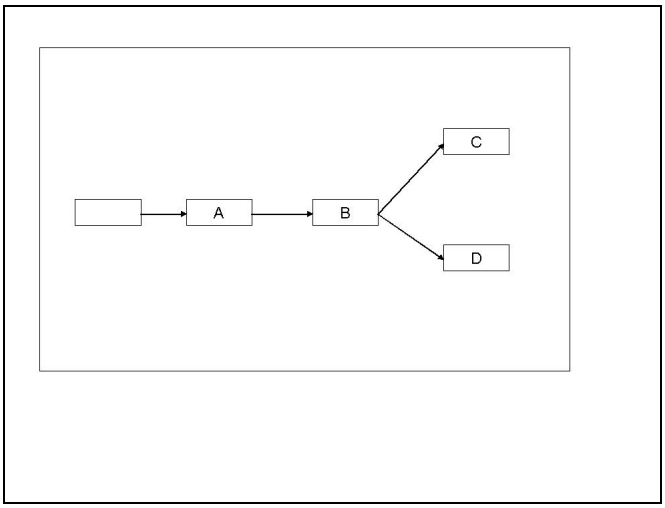


Figure 3.2: Domain Decision: Problem with Connection

specification by making design decisions for whole societies (groups) of agents based on the goals the group will attempt to achieve. For example, a market-based coordination model may apply to a society of agents if solutions to the goals it attempts to achieve are likely to be most easily found in a marketplace environment.

We illustrate the points above using Figure 3.2. In this diagram we show a software architecture based on the structure of a manufacturing domain, for example. Each box represents an agent performing a process and then making a decision as to where control will be passed next and maps on to parts of a production line in which those processes and routing can actually take place. However, the structure of the domain may be determined by constraints that do not apply to the software, such as the availability of machines with particular functions, space, necessity for continuous running when one machine was replaced by another etc. The part marked B in the diagram involves a convoluted process in which a decision must be taken which affects the process performed. This decision is the same as that which determines whether C or D are executed next. It would be more efficient in the software organisation to make the decision at A, and then split into two processes, B1 and B2, instead of B alone, where B1 leads on to C and B2 leads on to D. If this is done, the division of functionality will be connected to the requirements in a way that following the division in the domain would not be.

3.2.3 Flexibility

The assumption in agent-oriented software engineering that the components making up an application are autonomous, flexible and social (or even a subset of these) places demands on a design and implementation to provide for these capabilities (as compared to, for example, object-oriented systems where there are fewer assumed requirements for the comprising objects). By using agents in the design process, application developers must be aware that constraints are being placed on the system simply by the existence of the agents. AOSE methodologies should guide the designer in justifying the existence and functionality of agents while utilising their flexibility.

One way to achieve this is to reduce the structural requirements to those that are truly necessary right from the start. Bussmann [18] achieves this by minimising the structure to the necessary decisions that need to be made and not attaching any other demands on the decision points apart from their interdependencies. Tropos [16] similarly restricts the identified functionality to simple goals with no additional constraints other than being attached to the user possessing each goal.

Roles generally have associated responsibilities (or obligations) and other restrictions [63, 102, 106, 107]. These need to be justified by the requirements to show that they are necessary, though the justification also depends on the preceding division of functionality into roles being justified. Unjustified (as well as justified) restrictions on agents due to the roles they play could reduce the amount of opportunism that they can engage in. Similarly, associating restrictions with membership of a group of agents at design time, as in SODA [83], could unjustifiably reduce the opportunistic actions the agents could perform when the open system changes.

We illustrate the points above using Figure 3.3. A group of agents is allocated to achieving a particular goal, and a particular coordination mechanism to aid them in cooperating towards that goal. This tying of goals to methods of achieving them works against the flexilibity that agents are normally assumed to have in achieving a goal. For example, in some circumstances it may better achieve the requirements to use a coordination mechanism based on judging the reliability of other agents by their past record rather than negotiating with them to find a good solution. The process of negotiation may take a substantial amount of time and this is not applicable when speed is a requirements.

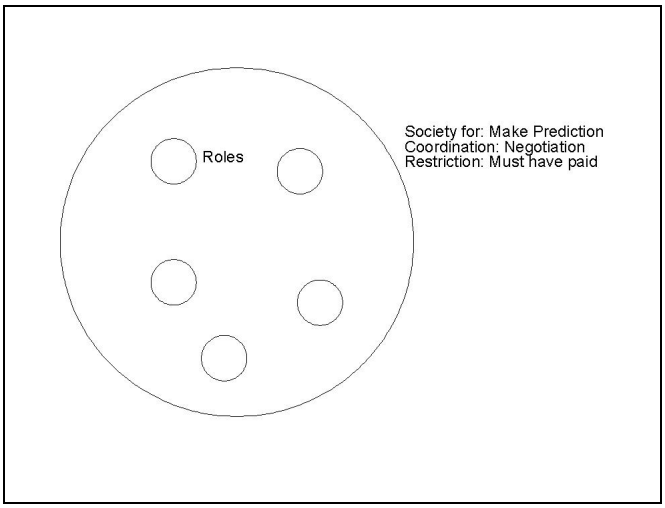


Figure 3.3: Societies: Problem with Flexibility

3.2.4 Interoperation

Most of the methodologies allow interoperation between known and unknown agents to be expressed. Bussmann's methodology [18], targetted at closed manufacturing domains, does not have this methodology capability as it requires the source and target decision points to be expressed in dependencies within each workflow. Tropos [16] is also aimed at closed systems as the roles of external users and service providers are identified at the start and used as a basis for design.

As agent roles are expressed at a more abstract level than agents, there is no reason why any agent, regardless of the time or location at which it is deployed in the open system, should not take on a role if it has the capabilities to do so. Similarly, there is no reason why agents from anywhere in an open system could not be part of SODA societies [83] as long as they obey the restrictions and can use the coordination model.

In Figure 3.4 we illustrate the points above. In the architecture shown, agent roles A and B are limited by tying them to particular entities within the closed system (in this case, two users). This is not opportunistic with regards to the application goals as it means that functionality developed by others and available in the open system cannot be re-used because it does not have that dependency.

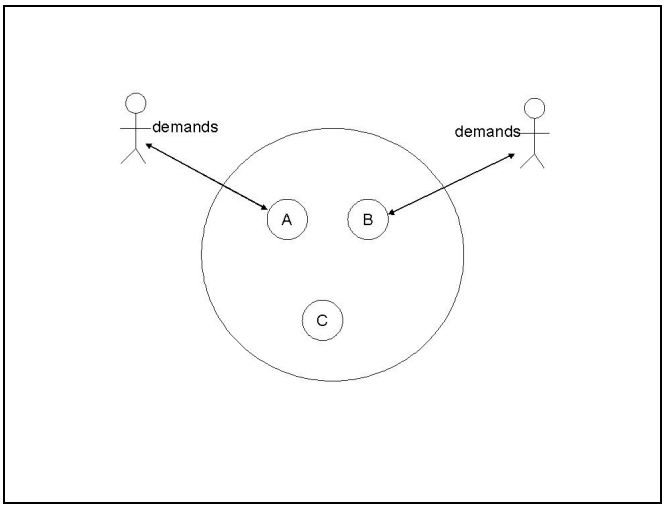


Figure 3.4: Requirements Analysis: Problem with Interoperation

3.2.5 Comparison Conclusions

Existing methodologies do not guide the designer in producing justified, opportunistic designs. Roles are a useful abstraction for agent-based system design but are not readily identifiable, cannot justify their own division of application functionality and their associated responsibilties restrict, possibly unjustifiably, the opportunism of agents adopting the roles. However, as an abstraction over particular agents they are useful in providing a *placeholder* for unspecified agents in discussion, so that any agents within an open system may play a pre-defined role at run-time.

Other conclusions can be drawn from our comparison, such as that goals are among the entities usefully identifiable from requirements for agent-based systems. Also, to ensure flexibility, the application should be specified in terms of necessary, minimal entities completely derived from the requirements (such as workflow decision points). The following section provides a more thorough analysis to determine how our approach should be informed by the successes and failures of existing methods.

3.3 Analysis

If the existing methodologies do not consistently allow for the design of justified, opportunistic applications, as we claim above, we must create a new methodology to achieve our aim.

Taking each of the four methodology sub-capabilities discussed in turn, we briefly derive the properties of an approach that satisfies them.

3.3.1 Identification

Identification of structures in requirements is the domain of requirements analysis, or, more broadly, requirements engineering. Entities found to be consistently identifiable in requirements include objects, functional goals and non-functional goals. We intend the function of objects to be largely replaced by agents in the eventual designs, so it may be more useful to identify functional and non-functional goals. The former type of goal expresses a state of the system desired within a given context. Non-functional goals are priorities and restrictions associated with achieving functional goals. In this thesis, we refer to functional goals simply as *goals* and non-functional goals as *preferences* to match folk psychology [14, 13] terminology often applied to agents.

3.3.2 Connection

In order for design decisions to be justified they must be matched by (traceable to) the application requirements. If we take the approach described above and identify goals and preferences from the requirements then further design decisions must be based on achieving the goals and matching the preferences. In particular, the agents in our applications should be designed to achieve the goals while matching the preferences.

3.3.3 Flexibility

Opportunism requires agents to be allowed to decide which others to cooperate with to achieve desired functionality, and that this should be done without regard to whether those others existed at design time or not. Also, any restriction on their choice should be justified by the requirements. If the designer identifies goals and preferences in the requirements, then the design should not constrain which agent achieves each goal at any one time, unless the preferences demand such a restriction.

3.3.4 Interoperation

The description of interoperation between agents created by the designer and agents elsewhere in the open system, possibly unknown at design time, requires that the designer is able to describe interactions between a set of agents, at least one of which may not have well-defined functionality. Roles are useful in abstracting over any particular agent and so can be used to express interactions between unspecified agents.

3.4 The Agent Interaction Analysis Methodology

From the analysis above we can generate a more specific approach to solving our problem.

3.4.1 Agent Interactions

Identification of goals and preferences by requirements analysis seems a useful starting point. For these goals and preferences to be manifested in an open system, and, therefore, for the application to exist in the open system the application designer will add agents to achieve the goals (and match the preferences). If these agents are opportunistic they will interact with existing agents and resources in the open system so as to best achieve the goals. The notion of whether a goal is achieved in a better or worse way depends on the preferences. For example, in our case study the goal to return a weather prediction to a user would have the preferences of, and so the goal achievement judged upon, the prediction's accuracy.

We stated in the analysis above that *interoperation* between unspecified agents could be modelled using the abstract notion of roles. However there are noticeable problems with identifying roles, justifying division of functionality into roles and only placing the restrictions of roles on agents where justified. The latter two problems (corresponding to the methodology capabilities of *connection* and *flexibility*) are due to the responsibilities and other restrictions associated with roles. Therefore, we can remove this problem by modelling interoperations using roles that have no associated properties aside from their part in those interoperations. These minimal roles are called, in our approach, *interaction roles*.

The other problem mentioned above is that of *identification* of roles from the requirements. However, we now say that interaction roles are defined by nothing more than their part in agent interactions and that those interactions take place in the open system application at precisely those times when agents wish to achieve goals. Therefore we can

identify interaction roles by specifying that for each goal (or instance of a goal as the goal may occur more than once), there is an interaction in the open system and in each interaction there are interaction roles played by agents. We are, therefore, modelling the application in terms of agent interactions in an open system.

3.4.2 Design Decisions

A question that may be asked is: how many interaction roles are there in each interaction? For example, in trying to get an accurate weather prediction an agent may ask a broker where to find a suitable predicter agent (which, therefore, involves three agents) or directly ask for a prediction from a known predicter (which involves two agents). Another reasonable question is that, if all responsibilities and restrictions are removed from the roles, how is the design in any way tailored to the application?

The reason these questions arise is that we have taken a useful concept for describing interoperation (a role) and removed enough associated restrictions from it that it becomes as minimised as unelaborated decision points or object classes (in workflow modelling and object-oriented development respectively). This minimisation of meaning is useful for the initial stages of identification and justified division of functionality. However once we have modelled the application as a set of agent interactions between minimal roles in an open system, we can then make design decisions that add restrictions to those roles as long as each one is ultimately based on the requirements (the *connection* methodology property).

3.4.3 Implementable Agents

Of course, the outcome of the design process should not be a set of abstract roles but implementable agents. The designer wishes to know what agents, and in what form, to implement and add to the open system in order to realise the application. In terms of our approach, they wish to add agents that will ensure achievement of the application goals while matching the application preferences. There are also other demands common to many applications, such as reducing the number of agents added to the system (as argued in Section 3.2.3).

As these agents will be attempting to achieve application goals, this question amounts to design decisions on which agents to implement that can fulfill the interaction roles previously modelled. We will return to this problem in Chapter 5.

3.4.4 Agent Interaction Analysis

To solve the problem of creating justified, opportunistic designs we have created a methodology based on the approach formulated above. It is called *Agent Interaction Analysis*, and will be explained in full over the next three chapters.

3.5 Summary

Methodologies can be judged to produce justified designs if they consistently allow identification of the analysis entities they use from the requirements and at each stage the design decisions are guided by information derived solely from the requirements. Methodologies can produce designs of opportunistic applications if they only restrict application behaviour where justified by the requirements and allow modelling of interoperations in the open system.

Existing AOSE methodologies do not consistently produce justified, opportunistic designs. Identification should be based on concepts of user requirements (such as goals) rather than concepts of internal software structure (such as roles). There must be some means of attaching (justifying) the design structure, including division of functionality to the requirements. Due to their complex operation the existence of agents places demands on the system. The agents should not be further constrained from opportunism except where the requirements demand it. Abstraction from particular agents allows open system interoperation to be modelled.

We suggest an approach, *agent interaction analysis*, in which goals are identified in the application requirements, and the application is modelled as a set of agent interactions to achieve those goals in an open system. Each interaction is made up of a set of interaction roles that are tailored, through justified design decisions, to match the application preferences. Agents are then implemented, when necessary, to be able to play those roles.

The next chapter will provide detail on the initial stages of *agent interaction analysis* such as requirements analysis and modelling in terms of agent interactions. Chapters 5 and 6 examine how to derive designs from the interactions, the former concerned mostly with infrastructure supporting agent interactions, the latter providing techniques for selection of coordination mechanisms. Chapter 7 provides an evaluation of the methodology, with reference to the case study (the whole of design of which is in Appendix A). Chapter 8

summaries the contributions and limitations of our work and discusses further work to be attempted.

Chapter 4

Requirements Analysis and Goal Decomposition

4.1 Introduction

The agent-oriented software engineering approach we are taking is based on the idea of using interactions between agents as primary abstractions in analysis and design. This chapter describes the analysis phase of our methodology. We start with an overview of the methodology as a whole, and identify the analysis phase within it, in Section 4.2. Later sections detail the steps taken in the analysis phase to extract requirements information relevant for design and provide a worked example for our case study application, whose requirements were given in Chapter 1. We start with identification of system goals in requirements analysis (Section 4.3) and then decomposition of those goals into sub-goals (Section 4.4). We illustrate each step with examples from the weather mapping case study, introduced in Chapter 1. The complete set of results from analysing the case study is included in Appendix A. In Section 4.5, we examine how interactions between agents are modelled and a chapter summary is presented in Section 4.6.

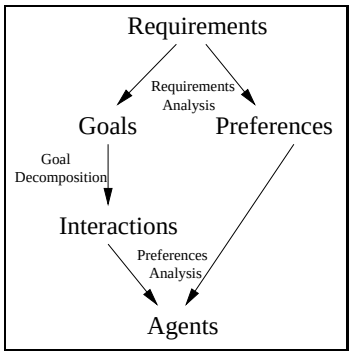


Figure 4.1: The transformations involved in *agent interaction analysis*

4.2 Methodology Overview

In Figure 4.1, we show the modelling artefacts produced by the agent interaction analysis methodology. This process is derived from the desire to achieve in a practical way the methodology capabilities defined in the previous chapter. The arrows in the diagram show the dependencies of the lower (later) analysis and design artefacts on the higher (earlier) ones. The labels on the arrows show the methodology stages required to transform one set of artefacts to another. As suggested in Chapter 3, requirements are analysed to identify system goals, and preferences on those goals. The system goals are modelled by interactions between unspecified agents, as are all subgoals derived in the *goal decomposition* stage.

Goal decomposition is the division of each system goal recursively into subgoals that are more easily analysed and implemented. Also, decomposition of goals into subgoals allows for more opportunism as agents can decide which others will cooperate over each subgoal. In the *preferences analysis* stage, the preferences inform the design decisions that create a design from the specified agent interactions. This chapter describes the *requirements analysis* and *goal decomposition* stages of the methodology, while the *preferences analysis* stage is specified in Chapter 5.

4.3 Requirements Analysis

As with every development methodology, an early stage of the design process is to derive the information needed, in a useful form, from the user requirements. This stage is the first of relevance to *agent interaction analysis* and the forms of information we want to be

produced are goals and preferences. Goals are descriptions of states of the system which the application is intended to realise. The lifetime of instances of goals in the system may be continuous over the system lifetime or dependent on context, e.g. appear at regular intervals or when invoked by a user. Preferences may take different forms determining, for instance, the measures of success for goals or restrictions on resources. Examples of goals and preferences are given below in Section 4.3.1.

A range of requirements analysis techniques are available. Several are described in [26], and one aimed at agent-based systems is described in [2]. It is interesting to note that these techniques derive relations between agents that are both internal and external to the system being designed and, because we are concerned primarily with interactions, there is no explicit distinction between these different classes of agent in our analysis stage. Requirements may take the form of a document or expectations from prospective users adapted to a useful form [50], and will probably be a mix of those that evolve over time.

Each section below describes part of the requirements analysis process with worked examples for the case study application.

4.3.1 Goal and Preference Examples

As an example of the products of an analysis, the following simplistic translations could be made from our case study application requirements.

- "Contributors can add data they have gathered locally, to the map..." translates to a goal to add data to the weather map. It also states a fact about the interface, i.e., that the contributor causes an instance of the goal to be present. This is the *context-based* appearance of goals.

- "...the functionality of the application should be available locally at each node,..." places a restriction (a strict preference) on the system and describes a system goal to keep the basic application functionality available locally to the user. This is the *continuous* appearance of goals.

- Aside from context-based and continuous appearance of goals, another possibility is *regular* appearance, e.g., "The weather data is distributed among the contributors... This distribution should be watched regularly to reflect the current demands." requires that regular redistribution should occur.

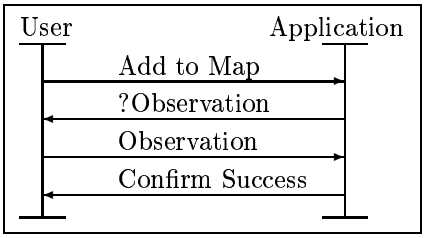


Figure 4.2: Event trace for modifying the weather map

- "After the priorities of the particular operations of the application, speed should be the most important factor in considering how the application is implemented" describes a preference concerning the result of processes, and so is a measure of quality of all the application goals.

Requirements come in a variety of forms and while goals are recognisable, as shown by the examples above, parts of a requirements document may be described in other terms. *Requirements engineering* is the process of deriving from requirements documents, goals that are coherent and easy to implement [95].

4.3.2 Scenario analysis

Event traces can be analysed for the requirement *scenarios* (interactions between the user and the local application) [23, 96]. We provide a set of analyses for our case study application. Exchange of commands, queries for supporting information (indicated by question marks) and information are shown as arrows from the sending entity (either the user or the application) to the receiving entity (either the user or the application). The scenario starts at the top of each diagram and a series of exchanges (events) occurs as time progresses to the bottom of the diagram.

- Figure 4.2 shows how a user expects to modify the weather map. In the trace, the user asks to modify the map, provides operation data and receives a confirmation of the success (or failure) of the operation.

- Figure 4.3 shows a user obtaining a prediction regarding a location at a particular future time.

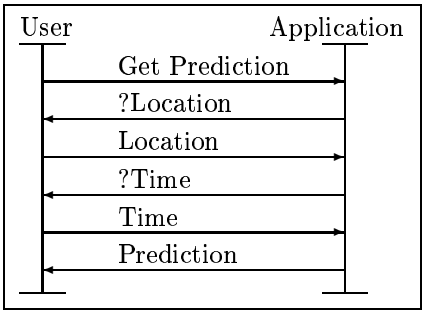


Figure 4.3: Event trace for choosing a map location prediction to speed view

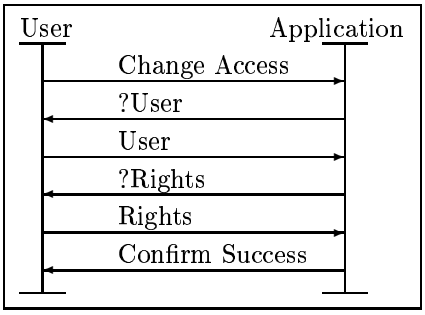


Figure 4.4: Event trace for changing the access rights of another user

- Figure 4.4 shows how an authorised user may change the access rights of other users.

The designer must decide which of these traces represents unique goals to be given to the agents, which are out of the agents' control and which are equivalent to other goals but possibly with different parameters. In the cases above, the events suggest an obvious goal to be achieved. An example of a scenario outside of the agents' control is starting the local part of the application, which is for the operating environment to achieve.

The goals identified from scenario analysis are named in a *data dictionary* shown in Table 4.1.

4.3.3 Entity Analyses

The requirements of any application are likely to be expressed in terms of domain entities, including physical entities and logical entities. In our case study, the physical entities include users and local computer resources, while the logical entities include access rights and

Type	*Name*	*Description*
Goal	Contributed	User contributed data to the weather map
Goal	Speed Viewed	User presented with view of map location prediction of specified time as fast as possible
Goal	Accuracy Viewed	User presented with map location prediction of specified time, as accurate as possible
Goal	Set Access	User has set access rights of another user

Table 4.1: Data dictionary after scenario analysis

predicters. As discussed in Chapter 2, the structure of the domain does not always match the desirable structure of the application. We would like, instead, to investigate what goals and preferences the existence of the entities imply. We have created the *entity analysis diagram* to examine the implications of an entity being mentioned in the requirements, and examples from our case study are given here.

The following entities from our case study application requirements are analysed to determine the goals and preferences implicit in their requirements usage, adding to the data dictionary as shown in Table 4.2. We analyse the application entities mentioned in the requirements to determine the goals and preferences their presence implied. In the analysis diagrams for the entities listed below, the application is shown as a solid circle, with the entity in question shown as a dashed shape positioned inside or outside the application (to show the relative position implied by the requirements). To determine the implications of the entity's existence, we examine its suggested interactions with other entities, including the user, and the properties it is described to have.

- Figure 4.5 describes the 'access rights' logical entity. A set of access rights for a user is an entity that filters the operations of the user on the weather map data. Access rights are themselves editable. This implies a goal (named 'Set Access') describing the state of access rights having been edited.

- Figure 4.6 shows an analysis of the 'predicter' entity mentioned in the requirements. The application answers requests for predictions, with either speed or accuracy prioritised. The predicters process the prediction requests. A prediction can be located within the local application (under the control of the designer) or elsewhere in the open system.

The full set of entity analyses for the case study is given in Appendix A. After these

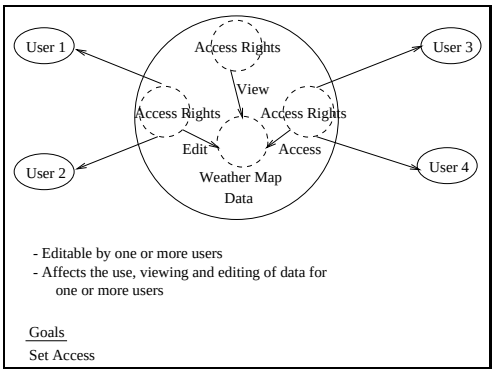


Figure 4.5: Entity analysis for access rights

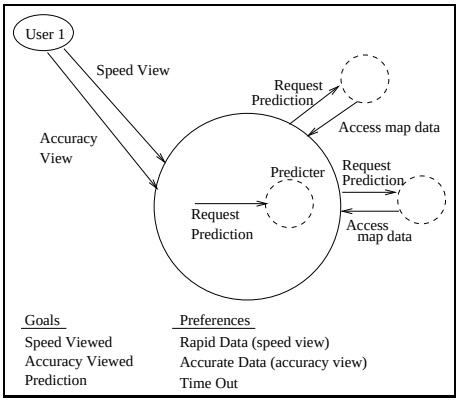


Figure 4.6: Entity analysis for predicters

Type	*Name*	*Description*
Goal	Contributed	User contributed data to the weather map
Goal	Speed Viewed	User presented with view of map location prediction of specified time as fast as possible
Goal	Accuracy Viewed	User presented with map location prediction of specified time, as accurate as possible
Goal	Set Access	User has set access rights of another user
Goal	Redistributed	Map data redistributed for high access speed
Goal	Prediction	Prediction regarding map location at given time has been made
Preference	Open	No limit on number of users
Preference	Validate Edit	Only authorised users can contribute to map location data
Preference	Validate View	Only authorised users can view predictions on map location data
Preference	Time Out	If prediction takes longer than 10 seconds then stop and warn user

Table 4.2: Data dictionary after entity analysis

analyses the data dictionary is extended to that shown in Table 4.2.

4.3.4 Goal Analysis

Each system goal is analysed to determine the actions that trigger the addition of instances of the goal to the application and the preferences associated with each. The analysis adds preferences to the final data dictionary at the end of the requirements analysis stage, as shown in Table 4.3. In the analysis diagrams for the goals analysed below, we specify the trigger (as an interaction with other entities) and the end state of the goal divided into those entities affected. When showing information being passed between entities, we use the UML convention of an arrow tailed by an empty circle for the passing of parameters required for goal achievement, e.g. Contribution in Figure 4.7, and an arrow tailed by a filled circle for the passing of feedback on a goal's achievement, e.g. the acknowledgement and warning shown in Figure 4.7.

- Figure 4.7 shows an analysis of the Contributed goal, which, as stated earlier occurs when a user wishes to contribute data to the weather map. The goal is triggered (an instance of the goal created) by the user on submitting contributing data. The resulting state should be either that the weather map is updated and the effect acknowledged to the user or the user warned that the update cannot take place. The preference is for as many contributions to take effect as possible (Edit Effect).

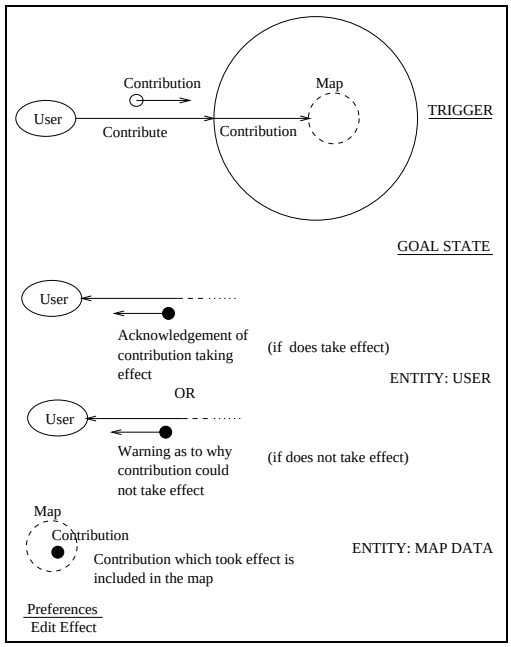


Figure 4.7: Contributed goal analysis

- Figure 4.8 shows an analysis of the Accuracy Viewed goal, triggered by the users when they request weather predictions for a given location and future time. The result is the prediction or a warning returned to the user, with a preference for accurate data being returned (Accurate Data). There are also restrictions on which users can access the predictions (Validate View) and on how long the process may take (Time Out).

- Figure 4.9 shows an analysis of the Redistributed goal describing a state in which weather data is stored locally to the users that access it most. Instances of the goal are triggered regularly by the application and the preference is for the users to have, collectively, as fast access as possible to the data.

Goals are desired end states so refer to the consequences of actions rather than an action itself, e..g 'User *contributed* data' rather than 'User *contributes* data'. It may seem strange many of the goals refer to the user triggering the goal, when the goal state does not involve the user, e.g. User contributed data. There are two reasons for referring to external entities not obviously within the goal state. First, it is useful for the goal description to refer to all information that the designer knows has to be provided from external sources in order for the goal to be achieved. The user must be known in order for the agents to examine the

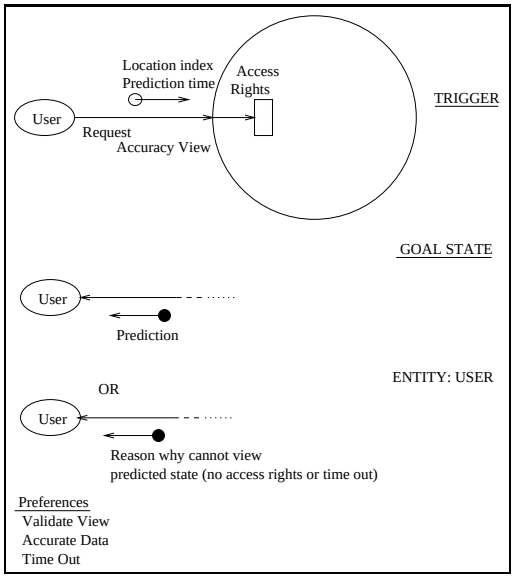


Figure 4.8: Accuracy Viewed goal analysis

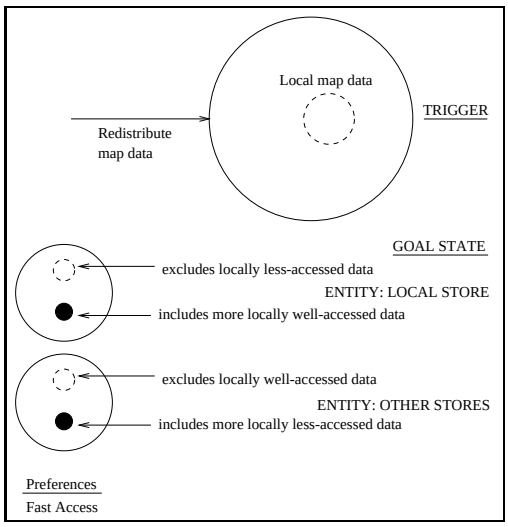


Figure 4.9: Redistributed goal analysis

Type	*Name*	*Description*
Goal	Contributed	User contributed data to the weather map
Goal	Speed Viewed	User presented with view of map location prediction of specified time (as fast as possible)
Goal	Accuracy Viewed	User presented with map location prediction of specified time (as accurate as possible)
Goal	Set Access	User has set access rights of another user
Goal	Redistributed	Map data redistributed for high access speed
Goal	Prediction	Prediction regarding map location at given time has been made
Preference	Open	No limit on number of users
Preference	Validate Edit	Only authorised users can contribute to map location data
Preference	Validate View	Only authorised users can view predictions on map location data
Preference	Time Out	If prediction takes longer than 10 seconds then stop and warn user
Preference	Validate Access	Only authorised users can edit access rights
Preference	Edit Effect	As many contributions as possible take effect
Preference	Rapid Data	Predictions are presented as quickly as possible
Preference	Accurate Data	Predictions are as accurate as possible
Preference	Fast Access	Map data should be distributed to give as rapid access by users as possible
Preference	Opportunism	The application should prioritise taking advantage of the most suitable functionality available in the system.

Table 4.3: Data dictionary after goal analysis

access rights of the user and decide whether the action is allowed to take effect. Second, the user should be informed of the success or failure of the operation. Therefore the goal 'User contributed data to the weather map' is an abbreviation for 'User is given feedback on goal failure, or access rights of user allows contribution to be integrated into the weather map, contribution is integrated into the weather map and user is given feedback on the goal success' (as shown fully in the goal analyses).

4.4 Goal Decomposition

In a dynamic system, agents working towards a goal which takes a substantial duration to complete may have to change their activities dependent on changes in the system. Agents changing activity based on context are said to be *reacting* to the environment. Goals may also be best achieved by cooperation between agents. Both of these flexible behaviours are fundamental to opportunistic behaviour and so should not be constrained by an agent-

oriented methodology. The behaviours require goals to be divisible so that agents can distribute the workload between them and find suitable points in time to reconsider the method of achieving the rest of a long duration goal. For example, the Accuracy Viewed goal (which provides the user with a requested prediction) will involve checking the user's access rights, interacting with a known accurate predicter then returning results to the user.

The actual decomposition of goals will have to take place at run-time by agents that can optimise the division in relation to the current state of the system. However, in order for the agents to have decompositions to choose from, these must be developed in advance. Also, in order to optimise system behaviour so that it corresponds to the preferences given in the requirements, designers need to analyse how system goals may be achieved. Therefore, the designers must develop possible decomposition of goals at the analysis stage. By decomposing system goals, the designers can create a set of goals each of which is decomposed into sub-goals. Alternatively, some goals will require only simple agent activity (actions) to achieve or only be achievable by agents outside of the system being designed.

The aim of the goal decomposition stage is to discover a suitable division of system goals into sub-goals such that, when the sub-goals are achieved in a specified way, the system goals are completed. The decomposition should reflect the useful divisions that an agent could make on possessing a goal. The smallest, lowest level goals should be those for which no further division would bring a useful separation of tasks or which can only be achieved by agents other than those provided by the designer. For example, in the holiday booking service suggested in Chapter 1, the goal to book a hotel room will be entirely dealt with by services provided by hoteliers, so there is no reason for further decomposition.

4.4.1 Integration Methods

The purpose of decomposition is to provide an interpretation that is easier to solve (turn into action) by viewing the goals as simpler parts. There must also be a stated method of integrating the parts which is simple and does not affect interpretation of the parts. For example, when an Accuracy Viewed goal is decomposed the access rights check must occur before the prediction is displayed (in other cases subgoals could occur in parallel) but neither of the subgoals depends on this particular integration method. Ability to divide goals can come from the separate entities that the goal refers to or different contexts (system states)

in which the goal may be attempted.

Alternatives

A goal may be divided into *alternative* subgoals. Alternative solutions to a goal (decompositions of a goal) may depend on context: the state of system entities and information passed with, or relevant to, the goal.

Considerations for Use The designers should consider how the entities whose existence and state are specified by the goal must be affected by the data given and how the state of the system, users and connected systems could pose problems or provide opportunities for achieving the goal.

Integration Method This type of decomposition has the simplest integration of sub-goals as when one of the sub-goals is completed the goal is completed without further effort. The method of integration chooses one of the sub-goals based on the current system state.

Concurrent Subgoals

A goal may be divided into *concurrent* or *contributive* sub-goals or actions. Goals or actions may achieve *part* of the state described by the goal. A goal may only be achieved while other actions are taking place or while a complementary goal exists in another agent or part of the system.

Considerations for Use The designers should consider how the goal entities can be created or achieve the desired state without the rest of the goal and how goal entities should be divided and the eventual states of the parts. They should also consider what activities require more than one agent or a constant feed of information to achieve all or part of a goal.

Integration Method This type of decomposition has the most complex integration of sub-goals in that it requires scheduling and communication between agents. The more coordination is required, the more complex it becomes.

Sequential Subgoals

A goal may be divided into sub-goals or actions whose achievement then makes achieving the goal easier. Goals or actions can be performed in a pre-defined sequence or steps can be taken so that an obviously simpler sub-goal remains left to be achieved.

Considerations for Use The designers should consider the states which entities must be in to be transformable, by an action, to the goal state and actions which make available more information useful for determining which alternative to take. They should consider actions that disentangle the effect of achieving one part of the goal, e.g. one entity's state, from another.

Integration Method The method of integration for this decomposing type involves achieving some goals before others. The more constraints apply, the more complex the integration becomes.

4.4.2 Examples

Goal decomposition is the process of dividing goals into several subgoals each of which can be coordinated over. As an example, the following goals from the case study are decomposed to separate concerns in the application functionality and to enable implementation. In the goal decomposition diagrams, a goal is given by a name in a box along with the information (parameters) needed to achieve the goal. The subgoals and preferences of the goals are shown at the end of lines below the goal. Preferences are marked with the word 'Preference'. Where a single horizontal line draws across the set of lines leading to subgoals, all of the subgoals must be achieved for the goal to be achieved. Where a double horizontal line draws across there is a choice between alternative sets of subgoals. For example, in the Contributed decomposition in Figure 4.10, the goal may be completed either by checking that access is denied for the user to contribute data *and* a warning is given to the user, *or* the map is edited *and* the success of the goal is reported to the user. Where goals are drawn in a heavy-edged box, there are no further decompositions given for that goal, e.g. Access Denied, Map Edited etc. in Figure 4.10.

- Figure 4.10 shows a decomposition for the Contributed goal where either the access is denied and a warning is given; or the contribution added to the map and the user

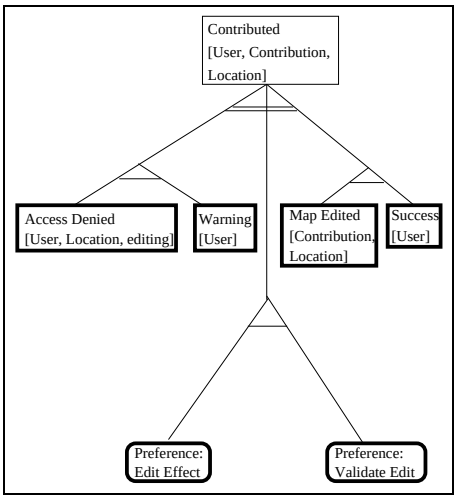


Figure 4.10: Decompostion of the Contributed goal

informed. Influencing the choice are two preferences: that as many contributions take effect as possible (Edit Effect) and that contributions only take effect when authorised (Validate Edit).

- Figure 4.11 shows the analysis for the Speed Viewed goal. Access rights are checked as with the Contributed decomposition but the Speed Viewed goal also states preferences for rapidity and to be opportunistic in using open system resources.

After decomposition, we include the resulting subgoals into our set and regard them in the same way as the goals originally derived from the requirements. The full list of goals and preferences at the end of the analysis phase is given in Table 4.4.

A goal can be decomposed in one of the three ways, as we discuss in the following sections. For each type, we describe the semantics of the decomposition, the factors a designer should consider before selecting that decomposition and how the goal parts are integrated to make the whole.

4.5 Agent Interaction Modelling

In Chapter 3, we noted that agent interactions must be mapped one-to-one from application goals in order to be an identifiable modelling concept. As a secondary part of the analysis, interactions also represent sub-goals derived from the goal decomposition phase. In this

Type	*Name*	*Description*
Goal	Contributed	User contributed data to the weather map
Goal	Speed Viewed	User presented with view of map location prediction of specified time (as fast as possible)
Goal	Accuracy Viewed	User presented with map location prediction of specified time (as accurate as possible)
Goal	Set Access	User has set access rights of another user
Goal	Redistributed	Map data redistributed for high access speed
Goal	Access Denied	It has been checked that the user is not authorised to perform the specified action
Goal	Warning	User has been warned of action failure
Goal	Map Edited	Map location data has been changed to a new value
Goal	Success	User has been informed of success of action
Goal	Prediction	Prediction regarding map location at given time has been made
Goal	Displayed Prediction	User views prediction
Goal	Rights Edited	Access rights for user have been changed
Goal	Least Accessed	The least accessed local map data has been identified
Goal	Move Data	Map data stored locally is moved to a remote store
Preference	Open	No limit on number of users
Preference	Validate Edit	Only authorised users can contribute to map location data
Preference	Validate View	Only authorised users can view predictions on map location data
Preference	Time Out	If prediction takes longer than 10 seconds then stop and warn user
Preference	Validate Access	Only authorised users can edit access rights
Preference	Edit Effect	As many contributions as possible take effect
Preference	Rapid Data	Predictions are presented as quickly as possible
Preference	Accurate Data	Predictions are as accurate as possible
Preference	Fast Access	Map data should be distributed to give as rapid access by users as possible
Preference	Opportunism	The application should prioritise taking advantage of the most suitable functionality

Table 4.4: Data dictionary after goal decomposition

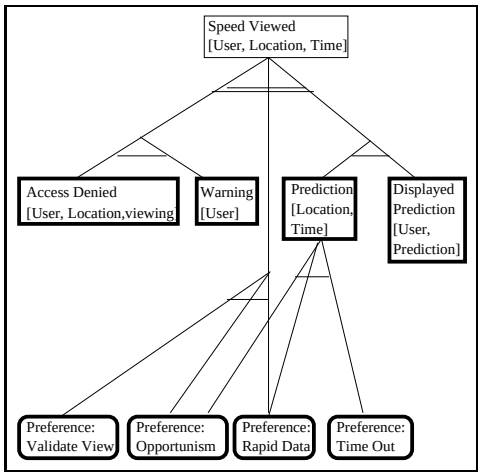


Figure 4.11: Decomposition for Speed Viewed goal

section we consider what abstract agent interactions actually represent in terms of the application.

Interactions occur when an agent possesses a goal and, therefore, needs to achieve it. An interaction represents the unification of all stages where at least one agent is acting towards the achievement of the labelling goal. This *cooperation* between agents may be the result of some unspecified amount of message passing which results in one or more agents attempting to achieve a goal.

The cooperation as modelled is not between particular agents in the system but abstract interaction roles. Interaction roles are the minimalistic roles which agents would play whenever the interaction goal comes to exist in the system. They represent potentially any agent in the open system, whether added by the application designer or otherwise, though this may be restricted if justified by the requirements. A single agent may play more than one role in the interaction. For example, an agent possessing the Contributed goal and able to edit the data map may achieve the goal through its own actions. This would still be termed an interaction in the context of our methodology, but would be a limiting case.

Figure 4.12 illustrates the structure of an interaction with three roles for co-operating agents to take in achieving the labelling goal. In contrast to organisational roles mentioned in the last chapter, interaction roles have no responsibilities, or other restrictions, in the application and, therefore, can potentially be adopted by an agent anywhere in the open

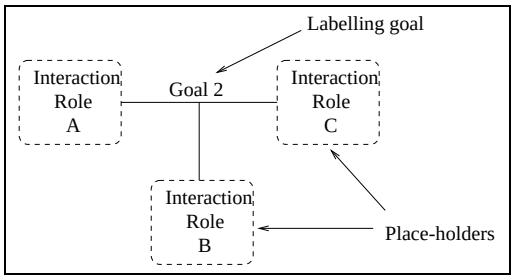


Figure 4.12: The structure of an example interaction

system.

Each interaction is labelled by a goal. There may be more complexity to an agent interaction than message passing between objects, in that an agent interaction may involve several communications being exchanged, communication of social context and even communication with third parties, but it has a similar eventual effect in that control is transferred between system components in a limited and meaningful way. Before cooperation can take place for a goal, one of the agents involved must possess that goal (be acting towards it), therefore, one of the interaction roles involved is being the *originator* of the labelling goal. For example, an agent representing the interface with the user may originally possess the Contributed goal when the user asks to contribute data, and will then find other agents to cooperate with in achieving this. The other agents in an interaction also have roles within that interaction such as being delegated goals, acting in parallel, monitoring the appropriate execution of the goal, and so on, with the exact roles depending on the goal. It is important to note that, due to the fact that agents may refuse to co-operate, an interaction only illustrates the successful case where the originator has eventually found a co-operator. It could also represent a chain of delegations where only the last agent in the chain knows how to continue its decomposition or completion [71].

4.6 Summary

Agent interaction analysis consists of transforming requirements into agents through the intermediaries of agent interactions consisting of interaction roles. We can analyse the requirements provided in terms of scenarios of use, entities mentioned and goals to achieve. From these we derive application goals and preferences. The goals may be decomposed to

provide more circumstances in which agents can exhibit opportunistic behaviour.

The goals are modelled (mentally, as there is no additional benefit from graphical specification) as being achieved by *interactions* between agents playing minimalistic *interaction roles*. A single agent can play more than one role in an interaction and the agent initiating an interaction is called the *originator*.

Once a designer has the goals and preferences and, implicitly, an interaction-based model, design decisions must be made in order to tailor the application, including the interaction roles, to the goals and preferences. This is considered in Chapter 5.

Chapter 5

Preferences Analysis

5.1 Introduction

Analysis of the requirements in the *agent interaction analysis* methodology expresses the application in terms of interactions between agents playing interaction roles to achieve application goals. The interaction roles are initially minimal to ensure no unjustified restrictions are placed on the agents in the application, so that the design remains justified and opportunistic. However, to ensure that the design meets the application requirements, the interaction roles must be tailored to the goals and preferences of the application. This will then allow the designer to implement agents that can play the interaction roles (where they do not already exist in the open system).

In this chapter, we describe the design phase of *agent interaction analysis*. As with other modern methodologies, such as Fusion [22], the design phase includes the creation of several models based on different aspects of the application under design. This often consists mostly of graphical representations to help communicate and encode the application structure. However, the potential complexity of multi-agent systems acting in dynamic open environments also requires much supportive structure.

The aim of the analysis phase is to derive and refine a structured specification of the necessary functionality from the requirements. The aim of the design phase is to bring those models closer to implementation, highlighting the design decisions to be made and the criteria influencing the decisions. The detailed analysis supporting the design decisions

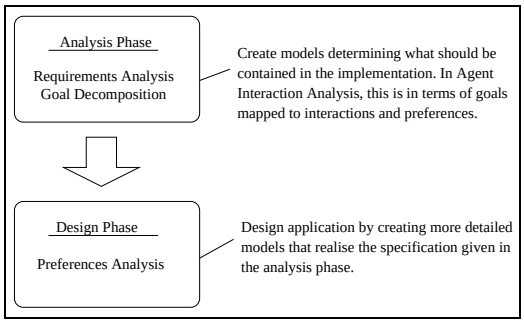


Figure 5.1: Modelling Processes in the Analysis and Design Phases

in our methodology is called *preferences analysis*, as the application preferences are the criteria on which design decisions are made. Without preferences analysis, a system design based on the interactions derived in the analysis phase could fail to match the priorities and restrictions in the requirements, i.e. the preferences, so the application design would not be justified. The absence of a substantial design phase to the methodology would also leave the designer with a potentially huge task in making design decisions without support from the methodology. The design is complete if an implementation can be made that unambiguously follows the design and meets the requirements. To achieve this, the designer is guided into making *design decisions*, which are choices made from weighing the benefits of various structures for realising the requirements. Figure 5.1 shows the modelling processes in each of the two phases, described in this chapter and Chapter 4.

An overview of the aims of the design phase is provided in Section 5.2. Section 5.3 describes the essential use of *modularisation* in the design process and how it can be usefully applied in Agent Interaction Analysis, with discussion of how a designer selects between possible module designs in Section 5.4. In Section 5.6, we describe the content of preferences analysis and how it aids design decisions. We illustrate the initial part of the design phase as applied to our weather mapping case study. The process is summarised in Section 5.7.

5.2 Designing Application Infrastructures

In order for the designs produced to be justified, an informal methodology must produce, through analysis, information that guides the designer towards justified design decisions.

Agent interaction analysis was developed with the following design aims.

Supporting Possible Interactions Guide the designer in producing a design for an application where it is possible for the agent interactions identified in the analysis phase to occur. Permitting the occurrence of agent interactions is fundamental to *agent interaction analysis.*

Separation of Concerns Provide a uniform, modular structure that supports the designer in analysing the infrastructure parts that support interacting agents, comparing models which instantiate those parts and making design decisions.

Traceability Highlight the design decisions that best match the application preferences so the designer can create a design conforming to the requirements.

Extensibility and Generality Ensure the process is flexible enough to accommodate application-specific infrastructure parts (those that support agents performing activities specific to the application), and the incorporation of models yet to be developed. In accordance with the software engineering principle of *generality,* we try to keep the design phase as widely applicable as possible.

In the sections that follow, we will address these aims.

5.3 Modularisation

The *design phase* of a methodology is the part of the engineering process which involves refining the models produced in the preceding analysis phase (see Chapter 4) to develop a complete design.

In Chapter 2, we gave principles that have been discovered to aid the process of software engineering. One of these principles is *separation of concerns*, which states that it is often possible to divide problems in various ways which allow the designer to focus on smaller, less complex targets. This can be applied to modelling in the design phase, particularly by way of the more specific principle of *modularity.* Modularity is separation of concerns by the structures implementing different functionality. This allows the designer to first focus on the individual modules and then the interaction of those modules. Ideally,

modules should display *high cohesion* (component elements are strongly related) and *low coupling* (modules are independent of each other to as high a degree as possible).

In Chapter 2, we reviewed several pieces of work into modularising *multi-agent system infrastructures*. Each agent-based system will have underlying resources allowing agents to perform actions necessary to achieve goals. An existing open system domain may provide some or all of these resources. In the sections below we will analyse how to apply the work described in Chapter 2 and examine how modularity is used in the design phase of *agent interaction analysis*.

5.3.1 Infrastructure Parts

Each application will have modules of its infrastructure that are not relevant to most other applications. For example, an application that analyses and transforms data from a database will have to consider the mechanism by which the database is accessed, whereas an application controlling a manufacturing process may not need to make a design decision on a database access mechanism. The database access mechanism is an example of an application-specific infrastructure part. We define an infrastructure part, in general, as follows.

Infrastructure Part A module of an infrastructure serving a stated purpose that can be identified as being potentially instantiated by one of several models.

Infrastructure Part Model A structure or algorithm that achieves a stated purpose in an agent-based system infrastructure.

For example, 'agent communication language' is an infrastructure part, and KQML and FIPA-ACL are models that instantiate it (see Chapter 2 for more details on these communication languages). Infrastructure part models may be developed at the same time as an application but they are more likely to be derived from previous work by others, such as with many agent coordination mechanisms, or from development experience, such as with many management services. Therefore, infrastructure part models can be *re-used*.

5.3.2 Decomposing an Application Infrastructure

The infrastructure needs to support the agents' operations. In order to identify the components of the infrastructure required for agents to interact over the system goals, the designer

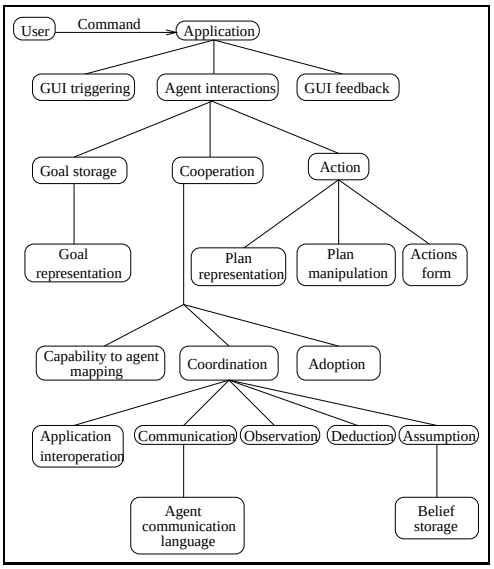


Figure 5.2: Infrastructure modularisation showing the support required for goal triggering via the GUI and for agent interactions

can examine various aspects of the application operation. In particular, the designer can refer back to the analyses of goals made earlier to determine the functionality required for goals to be triggered, examine the infrastructure required to allow the agents to interact and also examine existing infrastructures (as discussed in Chapter 2).

To modularise the infrastructure, the designer can follow the steps below.

1. Identify how the external entities, e.g. the user, will interact with the application.

2. Identify any further stages to transform interactions with external entities into agent interaction over goals.

3. For each distinct part identified as necessary for achieving the appropriate interactions with external entities, recursively decompose the part into a set of more specific parts required to achieve them.

4. Stop decomposing parts when it is considered that the infrastructure part models available cover all aspects of that part (so further decomposition is irrelevant or meaningless).

Figure 5.2 shows a decomposition of our case study application in which the user issues commands, such as requesting a weather prediction, through a graphical user interface

(GUI). The functionality of the application is achieved by interacting agents that possess goals.

- The goals are caused to exist (triggered) by the user interacting with the GUI, and the feedback from any goal achievement or failure is presented via the GUI. This is shown in the diagram as a decomposition of the application.
- The agent interactions are further modularised into the possession of a goal by an agent (the *originator* agent for the interaction), cooperation in achieving the goal and action arising from the cooperation.
- Goals possessed by an agent must be represented in a pre-specified form, as must the plans (plans are decompositions of action in this case).
- Cooperation is shown to require infrastructure to enable agent capabilities to be identified, agents to be coordinated to best achieve the goal and adoption by agents of the goal over which cooperation takes place (agreement to cooperate).
- Coordination between applications may require infrastructure to translate between them (as identified by Sycara et al. [94], described in Chapter 2).
- On an individual agent level, coordination depends on the coordination mechanisms used but certain common infrastructure parts can be identified. We will discuss these fully in Chapter 6.
- Communication requires a language in which to communicate and assumption may require some way of storing those assumptions explicitly (as beliefs).

It is likely that, in many cases, a designer can re-use a modularisation created for another application with modifications, and also use as guidelines the decompositions into layers, characteristics etc. [41, 91, 94] described in Chapter 2.

5.4 Basis of Model Selection

For each of the infrastructure parts identified in modularisation, a model will be chosen. In Chapter 2 we discussed Logan's classification into comparable models of the multi-agent system as whole. We utilise this idea for all infrastructure parts. There are several aspects to making a decision on the model for an infrastructure part, as follows.

- The models will most easily be compared if they can be expressed in a uniform way. However, there is no limit to the range of infrastructure parts or models for those parts so the standard must be flexible.

- The choice of a model should be based on the application preferences. In order for the methodology to guide the designer in making justified design decisions the model description should, therefore, aim to highlight the preferences that each model matches.

- A direct comparison between models may still be difficult in cases where the models operate in widely differing ways. For example, the coordination mechanisms described in Chapter 2 perform very different activities in order to achieve similar ends (effective cooperation between agents). Therefore, comparison of models for an infrastructure part may require a *supporting analysis* of the models to make them more easily comparable.

The influences on choice of model are shown in Figure 5.3. As is shown, the choice of model for each part is influenced by the comparison of described models, with possible supporting analysis, with respect to the preferences. For example, when choosing an agent communication language, a preference for interoperation with existing agent-based systems may suggest using KQML; while a preference for interoperation with future agent-based systems could suggest using FIPA-ACL.

5.4.1 Interdependencies

The other consideration in choosing models is the existence of interdependencies between models. A dependence could occur between models for different infrastructure parts, e.g. using a broker agent which takes monetary-style bids in order to choose between potential cooperators requires an agent communication language that can support bidding.

Dependencies can also occur between the same infrastructure part in different agents in a single interaction, e.g. an agent whose coordination involves making commitments can only cooperate with agents that will accept commitments. These interdependencies are illustrated in Figure 5.4. The arrows show influence from the choice of model for each instrastructure part on the choice for others, and also the influence of the preferences and interactions defined on those choices.

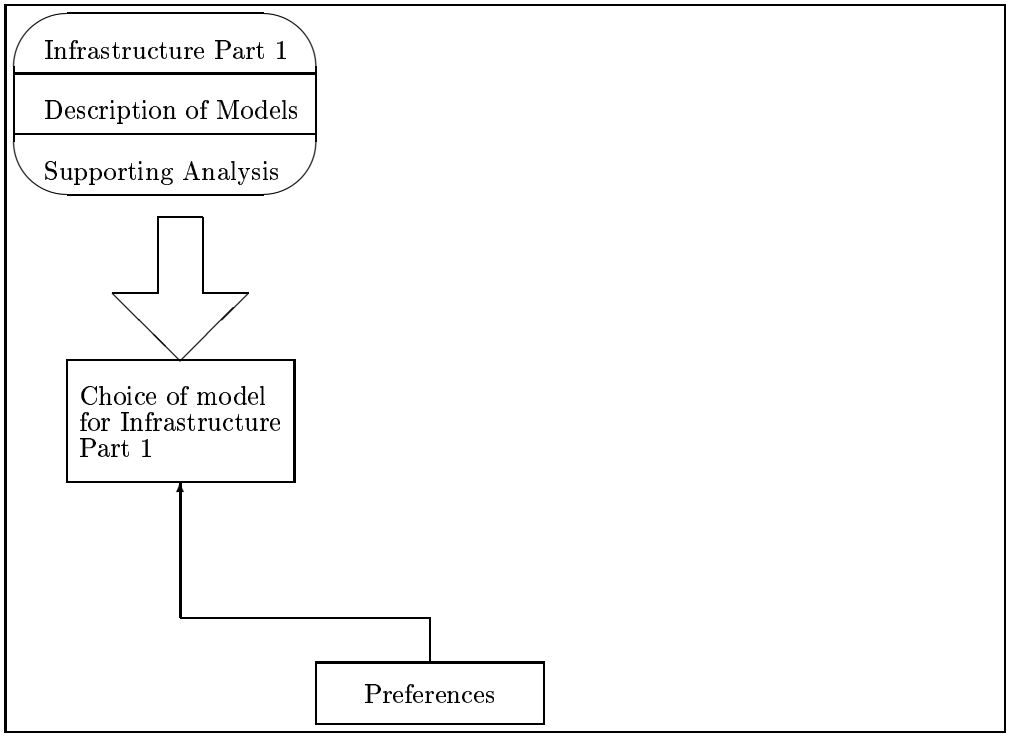


Figure 5.3: The influences on infrastructure part model selection

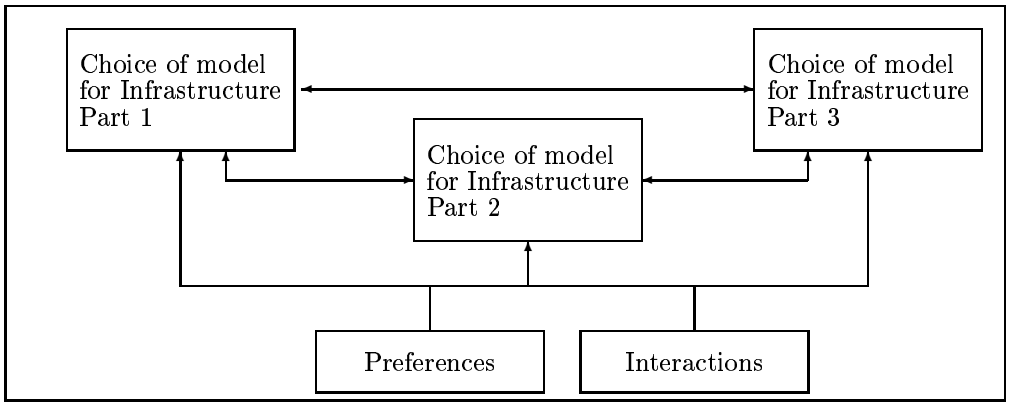


Figure 5.4: The interdependencies between infrastructure part models

The methodology should both make the designer aware of interdependencies and provide a process to resolve them. Awareness can be generated by a suitable method of describing infrastructure part models. We will cover the process of resolving interdepencies in Chapter 7.

5.5 Infrastructure Part Models Language (IPML)

The infrastructure part model language (IPML) is a schema format, standard over all the infrastructure parts, used to decompose infrastructure part descriptions. The language structure takes account of the influence that allowing interactions and satisfying preferences should have on the design. This approach is directly influenced by *design patterns* [40] in which object role models (derived from software developers' experience) are expressed in *pattern languages*, as discussed in Chapter 2. A pattern language is explicitly designed to best allow the designer to judge how appropriate each model is for an application and to compare models.

The IPML is a set of labelled properties which form a schema describing the model. We can compare our use of a language to that of Klusch and Sycara in uniformly describing broker agent types [65], as reviewed in Chapter 2. In that paper, they described each broker agent model by a schema consisting of four properties: the *pattern of interaction, signatures, states* and *transitions* of the model. While the language is clearly designed for comparison and selection, it is not based on any particular methodology and so does not provide any relation with the requirements of an application. Our methodology provides *justification* of design decisions by relating them to the elements in the requirements document that informed them.

The purpose of the *infrastructure part models language* (IPML) is as follows.

Infrastructure Part Model Language A set of properties for describing infrastructure part models that could be incorporated into an agent-oriented design, used to make comparison between models easier.

IPML's structure helps the designer to assess the influences of interactions, preferences and other models as shown in Figure 5.5. Given below are the properties that IPML demands be defined for each model, along with the reasons why the property is deemed essential where it is not obvious. An example use of IPML is given in the following section.

Part Name The name of the infrastructure part that the model instantiates, e.g. security mechanism, agent communication language.

Model Name The name of the model, e.g. FIPA-ACL.

Description An informal description of the model to aid understanding by the designer.

Algorithms Definition of the processes making up the model. It is essential for implementation that the procedural aspects of the model are clearly given. It is also useful for further decomposition and analysis of a model if its procedural steps are given. The algorithms will assume the presence of agents in the system as well as other appropriate resources (described in the Resource Use property below).

Priorities Many models will be based on prioritising one factor or another as expressed by this property, to aid selection of models by their match with application preferences. It is assumed that, when a model is implemented, the agents within the application affected by the model will behave in a way that prioritises those factors described in this IPML property.

Resource Use This IPML property is used to specify the system resources, e.g. services in the open system or storage space, required by the model. It is effectively providing the opposite information to the priorities of the model by stating its costs. The application preferences may limit the resources available as a whole or to individual agents.

Support Required To determine the influence on other infrastructure parts, the designer must specify the models for those infrastructure parts which are needed in order for this model to be used. In an example given earlier, we stated that choosing a coordination model based on bidding required, as a resource, an agent communication language that had performatives for placing bids. Other forms of support include the data storage and GUI requirements as these will be controlled by agents.

Scaling With a potentially large number of agent interactions occurring, it is important to analyse how the use of a model for a few interactions scales up for use in many interactions. For example, using a broker agent to organise coordination in part of an application may be wasteful for a small number of interactions, as the broker service may have high costs for setting up and maintaining brokering information. However,

if it were used for a large number of interactions, the costs may be outweighed by the benefit of having a single store of coordination information for all the interactions.

Applicability This IPML property, also used in pattern languages, used to describe preferable conditions that are not covered by the other properties.

Problems This property describes additional detrimental factors in using the model.

5.5.1 Use of IPML

Example uses of IPML are shown in Tables 5.1, 5.2, 5.3, 5.4. The informal division of the model's description allows the designer to compare the attributes of the model with the preferences of the application. It is intended that IPML model definitions be provided by the third-parties who have created those particular models. An application designer will then simply choose between the known models.

The following high-level parts were identified as fundamental to the functioning application. We refer to the graphical user interface below as the GUI.

GUI Triggering When users interact with the interface, by clicking on buttons for example, they issue commands to the agents making up the application. The mechanism by which the interface and agents interact is an infrastructure part. Two models that the designer could choose from for this infrastructure part are shown in Tables 5.1 and 5.2.

Capability to Agent Mapping This infrastructure part allows an agent requiring a particular capability (ability to achieve a goal) to discover an agent that possesses that capability. Two models that the designer could choose from for this infrastructure part are shown in Tables 5.3 and 5.4.

It is important to note that IPML models should not be created for each new application design. The IPML descriptions are deliberately application-neutral so that they can be provided by third-parties and *re-used*.

Part Name	GUI Triggering
Model Name	GUI Trigger Agent
Description	A single agent accepts all goals triggered from the local GUI and discovers other agents to cooperate with over each.
Algorithms	For each goal triggered from the GUI, the agent takes the following steps: 1. Adopt the goal benevolently (see Benevolent model for Adoption part). 2. As the originator for the goal, coordinate with other agents to achieve it.
Priorities	The model has the potential to be flexible through being agent-based and allows for coordination to be tailored to the goal from the time it is triggered.
Resource Use	There must be a reference to the GUI trigger agent from the GUI to allow goals to be communicated to the agent.
Support Required	A benevolent goal adoption mechanism is required in the agent.
Scaling	With a large number of goals passing from the GUI, the trigger agents may become a bottle-neck.
Applicability	Applicable where the goals from the GUI are to remain low in frequency, similar in preferences but with complex activities required to achieve each.
Problems	The agent may not be able to be tailored to all goals triggered from the GUI and may become a bottleneck with increased frequency.

Table 5.1: An IP model for the adoption of goals of a specific type.

5.6 Preferences Analysis

Bringing together the points raised above there is a procedure in *agent interaction analysis* for guiding the designer in making justified design decisions on infrastructure part models: *preferences analysis*. Once models have been chosen for all necessary parts of the infrastructure then the application design will be complete. Figure 5.5 shows the procedural structure of *preferences analysis*. Starting at the top of the diagram, each infrastructure part - e.g. agent communication language, management services, agent coordination mechanisms etc. - can be instantiated by one of several models. Each model is described in the uniform *infrastructure part model language*, which we give details of below. Some, more complex, infrastructure parts will also have a supporting analysis procedure in order to specify the models in a more easily comparable form. Based on the description and analysis, the designer can make a choice about which model to use for each infrastructure part. There are three sets of criteria for judging models.

1. The models should allow the *interaction of agents* over the particular goals derived in the analysis.

Part Name	GUI Triggering
Model Name	Designated Local Adopters
Description	When goals are triggered, the local infrastructure is queried to discover which agent is designated to deal with the goal. An agent is chosen for each possible goal triggered.
Algorithms	On triggering a goal, the GUI does the following. 1. Query the local infrastructure to determine the agent designated to address this goal; 2. Offer the goal to the agent; 3. The agent must accept the goal and cooperate to achieve it.
Priorities	The model prioritises division of goal execution between agents and a centralised, reliable point at which to query the agents currently available.
Resource Use	Requires a local store that the GUI can query for references to the designated agents.
Support Required	Designated agents must be compelled to adopt the designated goals on demand (see the Benevolence model for Adoption and the Forced Cooperation model for Coordination).
Scaling	The number of agents may increase as the number of triggered goals does. The effort required to extend the application by adding GUI triggered goals is therefore higher than with some other models.
Applicability	Most applicable where the frequency of triggered goals is high and diverse, the local resources are large and the goals can be communicated quickly so that the queries for local adopters do not overwhelm the local infrastructure.
Problems	As the local infrastructure is not agent-based itself, extra effort is required to maintain it and the triggering may be less flexible in this regard.

Table 5.2: An IP model for the adoption of goals of a specific type.

Part Name	Capability to Agent Mapping
Model Name	None / Broadcast
Description	No idenitification of agents by capability is made. Requests of cooperation are either directed at known individuals or broadcast to all agents in the open system, as dictated by the coordination mechanisms.
Algorithms	No algorithms are required.
Priorities	This model prioritises lower use of information storage, less reliance on a particular representation of capabilities and less reliance on agents registering their capabilities.
Resource Use	No resources used.
Support Required	The model relies on the communication mechanisms to provide discovery of an adequate range of agents so that the required capabilities can be found from among them. The communication mechanisms must allow broadcast of requests.
Scaling	The communication required for broadcast increases as the number of agents in the system increases.
Applicability	Applicable where the number of agents in the system is likely to remain low, where storage space is low and where capability registration is difficult.
Problems	The amount of communication is high as agents are contacteed regardless of capability.

Table 5.3: An IP model for mapping capablities required to agents possessing them.

Part Name	Capability to Agent Mapping
Model Name	Broker Agent
Description	Agents pass on requests for goals to be achieved through a broker agents in this model. Agents register their capabilities with the broker agent. The broker agent is an identified model for the coordination infrastructure part.
Algorithms	See the IPML description of the Broker Agent model for the Coordination infrastructure part.
Priorities	The model prioritises extensibility through being agent-based, centralisation of capability mapping and speed by use of a relatively simple mechanism.
Resource Use	The broker agent requires all the infrastructure needed in order for agents to register capabilities and make requests.
Support Required	Agents must use a boker agent as a coordination mechanism using this approach.
Scaling	The broker agent may become a bottleneck if the frequency and size of capability requests becomes large.
Applicability	Applicable where service requests are small or infrequent, the number of capable agents is large and where it is easier for external agents to register their services with an agent than be discovered.
Problems	The broker agent adds complexity to the design and the requirement that it be used potentially reduces the design's match to the preferences by restricting the choice of coordination mechanisms.

Table 5.4: An IP model for the adoption of goals of a specific type.

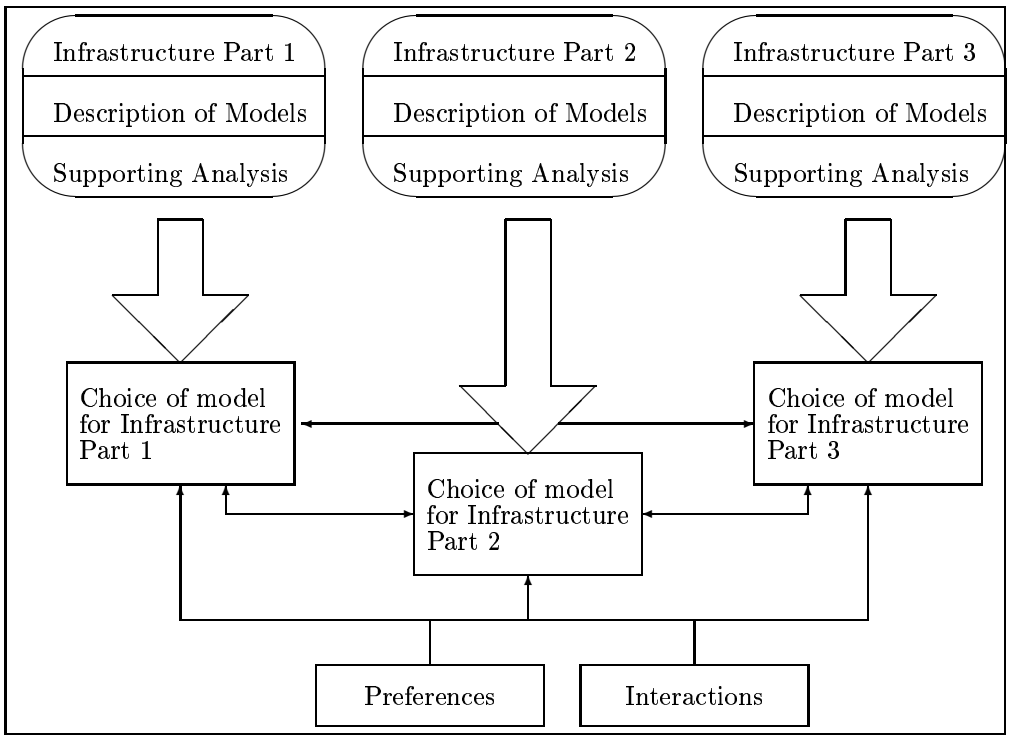


Figure 5.5: The procedural structure of *preference analysis*

2. The models should match the *preferences* derived from the requirements.

3. The models should be *compatible* with one another, i.e. any interdependencies must be taken into account.

For example, the set of interactions for an application may include one in which agents have a goal to confirm the password of a user. In that case, the *security* model, one of the infrastructure layers proposed by Sycara et al. [94] as discussed in Chapter 2, must allow external authorised agents to check the passwords while preventing unauthorised agents from discovering what they are.

5.6.1 Choosing Between Models

Preferences are the basis on which the designer chooses between two infrastructure part models. For example, the requirements may state that when a user clicks on a button on the user interface, the visible response is as fast as possible. By examining the significant aspects of the competing models, the designer may choose the model that is probably fastest in most cases. As preferences are derived from the requirements, there is *traceability* from the requirements to the design. There are several further issues concerning how one model is chosen over another.

While IPML deals with comparison in the general case, allowing the methodology to be used and extended easily, it does not provide structure for the more specific detailed analysis that may need to be done. Some infrastructure parts may be complex enough to warrant a supporting analysis process to further examine the suitability of one model over another. This is true in the case of agent coordination and we will examine the supporting analysis process for coordination in Chapter 6.

Also, there may be preferences that the designer does not derive from the requirements but believes are desired anyway. We refer to these as *implicit preferences.* An example for many applications is that the application should be as fast as possible, i.e. rapidity is a preference for every goal. Implicit preferences are likely to be of lower priority than explicitly stated preferences though the designer may be able to discover the actual priorities of the user through consultation.

5.6.2 Application and Agent Infrastructure Parts

One of the main aims of the design phase is to allow the agent interactions specified in the analysis phase to occur in the application. How this is achieved depends on the type of infrastructure part. We distinguish *application infrastructure parts* from *agent infrastructure parts*.

Application Infrastructure Parts Some infrastructure parts, e.g. management services, are implemented over the whole application and independent of individual agents. They may affect how and whether some interactions occur. Selecting a model for this type of infrastructure part means ensuring that it allows all application interactions to take place and obeys the preferences of the particular interactions it affects. Application infrastructure parts support more than one agent's operation.

Agent Infrastructure Parts Others infrastructure parts, e.g. coordination mechanisms, are specific to each agent. Selecting a model for an individual agent means analysing the *interaction roles* of each interaction (as described in Chapter 3) and selecting the infrastructure parts by the preferences of the interaction. The preferences of an interaction are those associated with the goal over which the interaction takes place, e.g. prioritising accuracy in retrieving weather predictions. The interaction roles will eventually determine the form of agents within the application and so they should be tailored to achieving the preferences of each interaction.

In the next chapter, we illustrate the extent of preferences analysis by examining a complex but essential infrastructure part: agent coordination. This is an *agent infrastructure part* as it is potentially different for each agent. Agent coordination is useful as an illustration for several reasons. First, it is an *infrastructure part* present in all designs produced in our methodology due to the use of agent interactions as a primary component of analysis and design. Furthermore, agent coordination is clearly effected by all the models shown in Figure 5.5 including allowing interactions to happen, obeying preferences and being effected by other *infrastructure part models*. Finally, it is a complex *infrastructure part* which requires a supporting analysis process.

5.6.3 Application Infrastructure Part Decisions

For most infrastructure parts, the designer can choose a model by comparing the priorities of the models with the application preferences. For application infrastructure parts, it should also be ensured that all identified goals can be interacted over and achieved. The designer needs to check that the resources used by each model (from the IPML Resource Use property) are available, that the other infrastructure parts required by the model (from the IPML Support Required property) do not conflict with decisions made on those parts and that the model will scale well enough for the application (according to the IPML Scaling property). The full list of IPML descriptions used in our case study is given in Appendix A.

After choosing a model for each of the application infrastructure parts not requiring further analysis, the models shown in Table 5.5 have been chosen by the designer. Justifications are given relating the properties given in the IPML descriptions above to the application requirements in Chapter 1, wherever a choice of models exists. Different models may be chosen for the Speed Variation and the Interoperability Variation of the requirements (see Chapter 1). The variations are abbreviated to SV and IV, respectively, in Table 5.5 for reasons of space.

The application infrastructure parts are annotated on the modularisation diagram shown in Figure 5.6.

5.7 Summary

In order to ensure an application has all the necessary supporting infrastructure to operate the designer must make design decisions regarding how infrastructure parts are instantiated. The instantiations are called *infrastructure part models* and the choice is justified by comparison between them with respect to the application preferences.

A standard pattern language, IPML, provides structure for application-neutral (reusable), flexible, comparable descriptions of models. Decisions may also be supported by further analysis to clarify the comparison. Infrastructure parts can either be application-wide or specific to interaction roles, the latter being illustrated in the next chapter.

Part	Model	Justification
GUI Triggering	Designated Local Adopters	The GUI triggered goals vary widely in their preferences.
GUI Feedback	Local GUI Access	
Goal Storage	Queue	
Actions	Local Access Acting	
Goal Representation	Predicate Logic	
Plan Representation	Tree	
Plan Manipulation	Tree Position	
Actions Form	Predicate Logic	
Application Interoperation	SV: Uniform Standard, IV: Translators	Where interoperation is paramount, translators are necessary, but otherwise the extra resource requirements means that a uniform standard is more applicable to applications with tightly defined functionality.
Communication	Stubs	
Observation	Polling	Agents are driven by the need for weather map data rather than being driven by current system state.
Deduction	No Additional Deduction	
Agent Communication Language	FIPA-ACL	Interoperation in the future is important to the opportunism of the application and using a subset of FIPA-ACL should not be significantly slower than other models.
Belief Storage	Predicate Logic	

Table 5.5: Infrastructure part models chosen for application infrastructure parts.

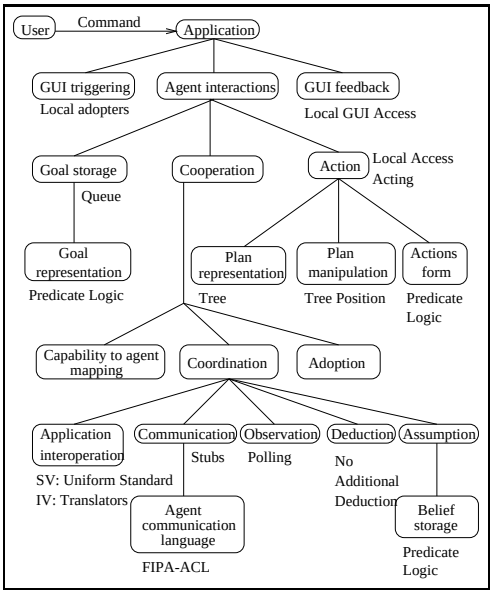


Figure 5.6: Infrastructure modularisation annotated with application infrastructure part model decisions.

Chapter 6

Coordination Design Decisions Using Assurance Analysis

In order for an application to be opportunistic, agents attempting to achieve application goals must be able to cooperate with other agents and resources in the open system. Cooperation needs to be controlled in order to ensure the application preferences are met. The mechanisms by which this cooperation is controlled are called *coordination mechanisms*.

By using interactions between unspecified agents for specifying the application structure, *agent interaction analysis* guides designers in creating application designs that are justified while allowing them to be opportunistic as well. However, this only *permits* a design solution and effectively pushes the problem of how to balance restrictive justification with the flexibility for opportunism to the level of interacting agents. More is required in order for a designer to know how this balance will be achieved in principle and in designs.

After briefly describing how choices on coordination fit into the design phase in Section 6.1, we introduce the theory of *flexibility bias* to allow agents to balance justified restriction with opportunistic flexibility in Section 6.2. We then illustrate how to define coordination mechanisms in IPML (Section 6.3) and how to decide between them with supporting analysis (Section 6.4). A summary of coordination design is given in Section 6.5.

6.1 Centralised, Distributed and Organised Coordination

In Chapter 2 we discussed how coordination can be centralised, distributed and organised. Organised coordination imposes a structure onto the use of distributed coordination mechanisms. Having distributed coordination does not require that the method by which agents discover each other is also distributed (we consider the problem of discovery to be one already solved to some extent by broker agents and non agent-based middleware, and do not consider it further here).

Following on from the work in Chapter 5, we consider coordination mechanisms to be infrastructure part models described in IPML. This chapter contains several example coordination mechanisms. An interesting piece of work for comparison is by Hayden et al. in which system-wide (i.e. centralised) coordination mechanisms are described in their own pattern language [51]. As this work is not presented in the context of a wider methodology, their language cannot be said to be tailored to matching preferences (or non-functional goals) and decisions on use are left entirely to the designer.

In the context of interactions, the *originator* of a goal is the one that initiates interaction (as described in Chapter 4) and so determines the (initial) coordination mechanism to be used. Coordination mechanisms, such as those described in Chapter 2, vary widely and are not readily comparable. To aid this, the coordination infrastructure part has supporting analysis procedures. For ease of understanding, we separate the analysis of coordination by agents over individual goals from the organisation of the multi-agent system as a whole. The former supporting analysis process is called *assurance analysis*, while the latter is called *collation* and we describe them in Sections 6.4 and Chapter 7 respectively. However, before analysing coordination mechanisms and models, a designer must understand the aims which dictate whether one coordination mechanism is better than another.

6.2 Opportunism, Justification and Coordination

Coordination mechanisms are as much part of the application as any other piece of functionality, and therefore have their own costs and benefits. A *justified* coordination mechanism is one that will match the preferences of the goal it is coordinating over. The agent uses

the coordination mechanism to decide which agents to cooperate with and then carries out the necessary process to coordinate and achieve the goal. The decision will be based on the preferences of the goal, but the coordination mechanism itself must match the preferences. For example, if a user of our case study application requests speed view (fast prediction on the weather map) then they wish speed to be prioritised. For the process to be fast, both the coordination and prediction should be as fast as possible.

Opportunism in the application depends on agents that are attempting to achieve application goals being able to cooperate with others that exist and were created by other developers or may exist in the future. An *opportunistic* coordination mechanism is one that does not restrict the set of agents with which to communicate at design time.

We can view the balance between application restrictions for justification and flexibility required for opportunism in terms of the *bias* of a methodology. An *implementation biased* methodology is one which has a strong emphasis on quickly reaching something implementable, achieving this by reducing the allowed designs at each stage of the methodology [102]. In *agent interaction analysis* the information provided by the methodology guides the design towards being justified and opportunistic. When we consider coordination, however, these two properties are opposed to each other. If we consider the organisational forms where justification or opportunism bias the methodology to the exclusion of the other, the problems are evident.

Justification Bias To achieve the greatest justification in a design, there must be no chance that an agent playing an interaction role will match the preferences less well than if a single reliable agent designated by the desginer always played that interaction role. As, in practice, the coordination between agents provided by the designer and new, unknown agents will involve costly procedures and risks, opportunism cannot be allowed to take place for any goal. Therefore, a justification biased design is effectively a closed system.

Opportunism Bias To achieve greatest opportunism any agent should be able to play any interaction role. In this case, however, no part of the application will be tailored towards matching the preferences, and so the design cannot be justified.

While a designer may desire either of the above biases for a particular application, we assume that, for any application intended to take advantage of an open system and

containing priorities among the requirements, the above forms are not appropriate. We suggest an alternative bias, called *flexibility bias* that assumes opportunistic behaviour will be positive on average but also that preferences should be matched on average.

It is based on the observation that, for quantitative priorities, such as speed, accuracy, amount of interoperation, storage space used etc., the aim is for the application's *expected* match of those preferences to be as high as possible. Therefore, we wish that the agents most likely to achieve goals with a better match of the preferences are those most likely to take on the corresponding interaction roles. In terms of design, a flexibility biased approach will ensure the following is true of a design.

- Goals are always achievable by the application (except where system failures make it impossible);
- There are agents that are tailored to particular goals;
- The agents tailored to a goal are more likely interact over those goals than other agents, but may not do so exclusively.

Ensuring that goals are always achievable where possible is realised by providing agents with the capabilities to achieve goals and offer to do so where required. The tailoring of agents to particular goals depends on the agents having some power to appropriately influence coordination with others. Appropriate influence, in this sense, means actions which ensure that the preferences are satisfied, and is caused by the coordination mechanisms the agents use. Ensuring that agents tailored to a goal are most likely to be chosen to interact over that goal, is a choice of the agent originally possessing the goal, and so is also a coordination issue. Comparing, analysing and selecting coordination mechanisms is discussed in the following sections.

6.3 Defining Coordination Mechanisms

Opportunism requires cooperation. Flexibility bias requires, amongst other things, cooperation to be controlled in such a way that preferences are matched both in the process and outcome of cooperation. Such control is achieved by *coordination mechanisms*. As they aid the agents' operation in achieving application goals, coordination mechanisms are infras-

tructure parts. As for any infrastructure part, the first step is to define available (known) models in a suitable form for comparison. We explore alternatives below.

6.3.1 Definitions of Coordination Mechanisms

Dynamic systems pose a particular problem for agent coordination as there will be no information at design time on some of the agents that will be coordinating during the application's execution. This means that the designer cannot tailor the design to take advantage of the superior capabilities of some agents and protect the application from the inferiority of others. The system must adapt at run-time.

In the literature review, we discussed work in which the designer selects the coordination mechanisms to be used by agents [31, 65, 91] and others in which the agent chooses between mechanisms at run-time depending on context [1, 5, 32, 33]. Using IPML to allow the designer to choose between mechanisms, we take the former approach. As, on first glance, the latter model seems more flexible, it is necessary to justify our position.

While agents can be given the ability to dynamically select a coordination mechanism at run-time, it should be noted that this is, in itself, a coordination mechanism. Such mechanisms are more complex and use computational resources more heavily than the coordination mechanisms they choose between, due to the fact that they include those coordination mechanisms as well as a further mechanism for selection. Their complexity adds flexibility and reliability to the single agent possessing the mechanism at the expense of the system as a whole. For this reason, we suggest this capability is more suited to agents which are designed alone, rather than as part of a multi-agent organisation. Therefore, as we are targetting the methodology at multi-agent system applications, while the dynamic selection mechanism is a valid model, the choice should be left to the designer in our methodology. Dynamic selection mechanisms are allowed rather than required.

Also, in the work described in the literature review, mechanisms were described either solely by their eventual effects [1, 33] or as procedures for their operation [5, 32, 65]. Our approach should be able to express the coordination models of both, but in a common form. An *eventual effect* form, i.e. a form in which a mechanism is described only by its end state or effect, is often specified as a tuple of values each of which describes some aspect of the mechanism's effect, e.g. duration required to operate or quantified benefit from operation. The *procedural* form is often specified as an algorithm for use or as changes in the state of

the coordinating agents. In our methodology, as part of the IPML, we specify mechanisms in a procedural form. The mechanisms are defined in the form of algorithms for use. We use a procedural rather than eventual effect description because agents are loosely coupled so coordination requires inter-agent processes. An eventual effect definition of coordination excludes all steps between start and completion and so excludes the interactions. Eventual effect definitions are not suitable for the design phase as they do not give enough details for implementation.

A possible useful supplementary technique for a designer wishing to create novel coordination and communication mechanisms to fit an application is given by Guerin and Pitt [49]. In this model, preferences (or equivalent non-functional application properties) are used to design a formally specified interaction protocol, which can then be implemented as an agent communication language suited to that particular type of interaction. In their work, this is particularly applied to interactions in which one party needs to cope with the deception or unreliability of another.

6.3.2 Describing Coordination Mechanisms in IPML

Coordination mechanisms, such as commitment machines and broker agents, mentioned in Chapter 2, are infrastructure part models for the coordination infrastructure part. We illustrate how IPML is used to describe coordination mechanisms by an example.

In Chapter 2, we looked at the work of Klusch and Sycara on classifying broker agents [65]. This included their own schema describing broker agent models by four criteria: *pattern of interaction, signatures, states* and *transitions*. We remarked that it was not made clear how these were to be related to the application requirements. On the other hand, the IPML properties directly associate the model (coordination mechanism) description with their criteria for selection. The criteria are preferences and interactions from analysis of the requirements and other infrastructure part models from the application design. IPML example descriptions for six distinct mechanisms are given in Tables 6.1 to 6.6. To ease explanation of their comparison these mechanisms are summarised below.

Commitments A fast, simple mechanism where a request is made for collaborators to commit to a goal and the first one(s) to reply are chosen. See [56] for more on commitments.

Trust A mechanism whereby agents offering to cooperate are assessed on the previous quality of solutions (best match of preferences) they have produced and the most *trustworthy*, i.e. the one that has the best match so far, chosen. See the IPML description in Table 6.3 and also [19, 35, 46, 48, 73] for more on trust-based coordination.

Broker Agent An agent-based mechanism where a broker agent (facilitator) records registrations of agents offering to attempt a goal. The broker can then be asked to supply cooperator references for the originator to coordinate with. See [65] for more on broker agents.

Forced Cooperation Object-oriented style cooperation where an agent is known to always accept requests for cooperation on command. See [22] for more on object collaboration.

Intention A mechanism whereby the originator does not cooperate but achieves the goal itself. See [42] for more on intentions.

Negotiation A mechanism allowing a group of agents to send a series of exchanges until they have agreed on assigning particular values to objects or tasks which enables the goal to then be best achieved. See [8, 55] for more on negotiation.

6.4 Assurance Analysis

Coordination is potentially a complex process, and the many mechanisms suggested are very varied. This can make it harder to select appropriate mechanisms to use in an application. One cause of the complexity is that coordination mechanisms involve several parts or stages each of which may or may not satisfy particular preferences.

For example, in an application using the matchmaker broker agent described in Table 6.4, the requesting agents need to store only a reference to a matchmaker to discover providers of services they will need, rather than holding references to providers themselves. Therefore, it appears that the data stored in the system will be reduced as replication of service provider references among agents is reduced. However, the data that a matchmaker must hold on each service provided may have to be extensive in order to correctly match the service to a request. As there may also be several providers for each service at any one time, the storage requirements may be larger than alternative coordination mechanisms.

Part Name	Coordination
Model Name	Forced Cooperation
Description	In this model, certain agents are required to cooperate over a goal on demand and are known to the agent employing this mechanism. The mechanism is approximately the same as message passing in (concurrent) objects.
Algorithms	To request cooperation from a forced agent, there is only one step. 1. Demand cooperation over the goal
Priorities	The model prioritises speed, reliability that the cooperators will be capable (through explicit design) and security in knowing the cooperator is pre-determined to be trustworthy.
Resource Use	Local information regarding the forced cooperation in agents is required by the agent employing this mechanism.
Support Required	The model requires mandatory adoption of goals in some agents providing the capability for this goal.
Scaling	Scales easily as it requires no communication beyond the minimum demand for cooperation, but does require suitable functionality to continue to be available within the application over time.
Applicability	This model is most applicable where security or reliability are of much greater importance than opportunism and where it is known that the forced functionality will always be available within the application.
Problems	Forced cooperation allows no flexibility in choosing cooperators so provides for no opportunism in exploiting the open system.

Table 6.1: An IP model for a coordination mechanism.

Part Name	Coordination
Model Name	Commitments
Description	Agents using this coordination mechanism send requests for goal achievement and request offers, accepting the first offer that is received in reply that is acceptable by their adoption mechanism. An agent offering to achieve the goal and subsequently being accepted is committing is committing themselves to the goal's achievement.
Algorithms	1. Send out requests to viable agents (see Capability to Agent infrastructure part); 2. Wait for offers; 3. Pass all offers to the adoption mechanism to consider; 4. Send acceptance of first viable offer.
Priorities	The model minimises processing and so is fast but still able to take opportunity of all services in the system.
Resource Use	The model requires communication of requests and acceptances.
Support Required	The agent communication language needs to be able to express requests, offers and acceptances in a standard way.
Scaling	As the mechanism does not require a large amount of resources or any degree of centralisation, it should scale well in most cases.
Applicability	The model is applicable where ensuring the speed of coordination is of greater importance than the quality or reliability of solutions. It is also useful where the application needs to avoid centralisation.
Problems	As the commitment offers are not assessed to determine their suitability, the agent using the mechanism has no guarantee of achieving the highest quality solution, where this is a concern.

Table 6.2: An IP model for a coordination mechanism.

Part Name	Coordination
Model Name	Trust
Description	The agent using this mechanism checks the quality of any solution provided by an agent and uses these assessments to decide which offers to accept in the future. The checks can be either by observation of the state achieved, if the goal attempts to achieve a particular observable state, or by an independent production of the same information, if the goal attempts to derive some information.
Algorithms	1. Send requests to agents; 2. Wait for a suitable duration to receive offers; 3. If no offers received, resend requests; 4. If some (one or more) offers are received, assess them to determine which comes from the most trustworthy agent; 5. Accept the offer from the most trustworthy agent.
Priorities	A trust-based mechanism prioritises quality of solution and reliability in obtaining a solution.
Resource Use	The agent using the mechanism will need to possess quantitative assessments of the trustworthiness of other agents, which rise and fall depending on observed quality of solution. The agent will make observations or request extra information for each goal.
Support Required	As the agent using the trust mechanism algorithm must wait a specified duration, it requires a scheduling mechanism capable of this.
Scaling	With a large number of possible collaborators for a goal, the number of models possessed by the agent will also be large. The observational checks will add to the time taken to process each goal by the agent.
Applicability	Where the quality of the goal is able to be checked in some way and is of more importance than speed or the low use of resources.
Problems	A trust-based mechanism may add a significant amount of processing to each goal the agent seeks cooperators for.

Table 6.3: An IP model for a coordination mechanism.

Part Name	Coordination
Model Name	Matchmaker Broker Agent
Description	A matchmaker agent is a brokering solution in which requesting agents are given contact information of the providers. The matchmaker maintains a database of the services agents can perform (goals they can achieve) but uses this solely to identify a suitable provider for each requester. Requesters and providers then communicate without any further intervention by the matchmaker, which allows for greater flexibility.
Algorithms	The algorithm for an agent registering a provided service with the matchmaker requires only one step: 1. Communicate service provision to matchmaker. An agent using a matchmaker to find a service must also get a commitment from the provider as the matchmaker agent does not do this. 1. Request communication channel to provider of service from matchmaker; 2. Receive provider channel from matchmaker; 3. Communicate goal commitment from provider; 4. Receive commitment to achieve goal.
Priorities	The model allows agents to find service providers, gives flexibility in communication between provider and requester and has a relatively simple implementation.
Resource Use	The mechanism requires a data store recording the services that each agent provides.
Support Required	The agent communication language used must support registering and requesting services. Agents also need a reference to a matchmaker or to be able to get one.
Scaling	Little communication is needed for making a request and, by transferring communication responsibility to the requester, the matchmaker has no further work after matching requester with provider. However, the recorded services and the search of the data increases as the number of agents registering services increases. If the adverts for services are to be large in size then the database will quickly expand.
Applicability	The mechanism is applicable where agents will and can easily register services to brokers accessible by those others that will need the services. The model allows an agent-based model for the Capability to Agent Mapping part providing flexibility and extensibility. It is also applicable where communication between requester and provider must be flexible, complex or private.
Problems	The data store may become large with many agents or services. Agent identities must be revealed in order to provide services, so privacy and looseness of coupling are reduced. A requester must ask both the matchmaker and the provider for a single goal.

Table 6.4: An IP model for a coordination mechanism.

Part Name	Coordination
Model Name	Intentions
Description	When an agent needs to find a suitable other to cooperate with over a goal, it may be that it would be best for the agent to attempt the goal itself. This mechanism causes the agent to always choose this option for a goal.
Algorithms	On becoming originator for a goal, the agent adopts the goal itself.
Priorities	The mechanism prioritises speed in decision making and reliability in cooperation over flexibility and opportunism in using external capabilities that may provide a higher quality service.
Resource Use	No resources are required in addition to those necessary to achieve the goal.
Support Required	The adoption mechanism must allow several goals to be adopted benevolently as there are no other agents to take them on.
Scaling	If the number of goals becomes large, the agent could become overloaded as it must adopt them all itself.
Applicability	The model is applicable where security, speed and reliability are of so much importance that the goal should be achieved by the same agent requiring assurance that it will be done.
Problems	The mechanism does not take opportunity of the potentially higher quality services available in the system.

Table 6.5: An IP model for a coordination mechanism.

Part Name	Coordination
Model Name	Negotiation
Description	Broadly, we use negotiation to mean repeated communication between agents over one or more values ending in an agreement to action by the agents.
Algorithms	The value(s) negotiated over represents the quality of the goal solution and its meaning will, therefore, depend on the preferences. For this reason, we cannot provide a generic negotitation mechanism. In general terms, each communication will be a request for suggestions, a suggested allocation of values between agents or an acceptance.
Priorities	The model prioritises reaching a high quality solution through communication.
Resource Use	Agents engaging in negotiation must have a means of judging the next step of a negotiation given the current position.
Support Required	The agent communication language must be sophisticated enough to allow the expression of values being attached to domain concepts in order for those values to then be negotiated over.
Scaling	Negotiations need not involve many agents but can involve many communications between agents. Therefore, negotiation may not be suitable where this is a lot of communication occurring in the system for too little benefit.
Applicability	Negotiation is primarily applicable where there is something of value to be exchanged or distributed.
Problems	Negotiation can involve much complexity to work well.

Table 6.6: An IP model for a coordination mechanism.

The storage difficulties may increase if the matchmaker agent is stored on a single node of a network, whereas alternative coordination mechanisms may involve the service provision information being spread amongst the agents all over the network. We can see that one mechanism may be more suitable than another, depending on the priorities of an application. Also, to discover its suitability, it may be useful to analyse different stages in a mechanism's use, e.g. division into service registration, service discovery etc. in the case of a broker agent.

Another cause of difficulty in selecting coordination mechanisms is that they are often described in various different terms in the literature to reflect different priorities in coordination. In an attempt to compare the mechanisms, and solve the problem of the properties of each part of a mechanism being different, we introduce an supporting analysis process for coordination. As mentioned in Chapter 5, supporting analysis can be provided for complex infrastructure parts to aid model comparison. The supporting analysis process for the coordination infrastructure part is called *assurance analysis*.

6.4.1 Assurance and the Single Agent Perspective

To provide the balance suggested by flexibility bias, we wish to allow the designer to tailor interaction roles to application goals. Therefore, a designer will analyse coordination issues from the perspective of individual agents (those taking on interaction roles). This is analysis from the *single agent perspective*. Analysing how to coordinate agent actions from the single agent perspective requires an assessment of what information the agent may have about the coordination process. Consider an agent A that possesses a goal and cooperates with agents B and C to achieve it (A is then the *originator* for the goal in this interaction as described in Chapter 4). The ideal behaviour of B and C, from A's perspective, is for them to work towards and eventually achieve the goal while prioritising the preferences associated with the goal. B or C may be used by different application and aim to match different preferences, or may be unreliable and so never achieve the goal. Then again, an agent unknown to the designer at design time may provide a better service and better match the preferences than known agents. Other techniques exist that examine the utility of different coordination between an already known set of agents [10].

We require a concept that expresses the priorities of an originator agent in coordinating over a goal, in order to allow comparison of coordination mechanisms for a single interaction role. Following the convention of describing the pieces of information possessed

by an agent as *beliefs* [43, 42], we describe this information as an *assurance belief* defined as follows.

Assurance Belief The certainty of an agent that a goal it possesses will be achieved in accordance with its priorities.

We refer to the agents' 'priorities' in the definition above rather than application preferences because the assessment is from the single agent perspective. As described in the section on flexibility bias, the designer should be guided to tailor agents to the application so that agent priorities match application preferences. The information that an assurance belief contains is considered to be possessed by an agent, in the design process, but may be implicit in its use of a coordination mechanism rather than explicitly represented at run time. An originator's priorities should be the preferences associated with the goal so that the designer can ensure that an agent maximising its assurance belief that its goal will be achieved will also be satisfying the application preferences. The certainty contained in an assurance belief may not be stated as a numerical probability in design, as the actual probability will vary between interactions over a goal. Instead it is an unknown value that can be raised or lowered through the use of different coordination mechanisms.

Assurance is similar to the concept of *trust*, particular associated in agents research with the work of Marsh [73]. Trust, in its broadest sense, is information possessed by one agent regarding the reliability of other agents in providing services. As with assurance, trust can be used to make coordination decisions, such as which other agent to cooperate with. Assurance is more general because it represents several priorities rather than simple reliability. It is more useful for justification because the priorities are derived from application requirements, which is essential for a justified design.

6.4.2 The Process of Coordination

The aim of assurance analysis, as with all supporting analysis processes for complex infrastructure parts, is to make comparison between models easier. This analysis is not qualitatively different from other software engineering analysis processes and so, we have drawn on the principles described in Chapter 2. In particular, comparison is aided by *separation of concerns* and *abstraction*, i.e. dividing coordination into more manageable pieces and abstracting from coordination mechanisms to find the processes common to all.

We achieve this by looking at the broad process of coordination, from a single agent perspective, and dividing it into generally applicable steps.

1. In order for an agent to coordinate with another, it must know how to communicate with, or at least observe, the other agent otherwise it cannot adjust its actions or those of the other agent. Therefore, it must have access to data regarding how to contact the agent. In addition, an agent may use other data regarding a potential cooperator in order to control coordination, such as its reliability. We call this collective data on an agent: an *agent model* (following the terminology of other research such as [47]).

2. To create an agent model, the modelling agent must acquire information on the modelled agent or the information must be built in by the designer.

3. The information acquired will affect (be used to maintain the correctness of) the agent models.

4. In order for there to be coordination, there must be action, otherwise there is nothing being coordinated over.

5. In cooperating over a goal, the agent aims to act so as to bring about the goal's achievement through the actions of the coordinating agents. Therefore, an action caused by a coordination mechanism will be based on predictions about the actions of the other agents in the coordination. A mechanism could only derive these predictions from the agent's knowledge of the agents, the agent models.

6. The aim of cooperation is to achieve the goal being cooperated over. The way in which the goal is achieved may also be important to the agent (i.e. the agent aims to match its priorities). The actions produced by a coordination mechanism will, therefore, aim to increase the probability of the goal being achieved by an appropriate method. We call this probability *assurance*. As the assurance concerning a goal is based purely on the agent's beliefs, it can also be modelled as a belief (an *assurance belief*), even though it may not be represented explicitly by an agent as such.

We use the modularised *process of coordination* above to enhance comparability of coordination mechanisms in the way described below.

6.4.3 Belief Acquisition

Step 2 of the process of coordination stated that agents acquire information to construct agent models. *Belief acquisition* is the primary method by which the agent acquires the information it needs to use a coordination mechanism. We identify four qualitatively different methods of belief acquisition for autonomous, social agents. These represent the two channels of input into an agent and then two internal operations of an agent.

Communication An agent may acquire information by other agents communicating that information.

Observation An agent may have access to the state of the open system resources, the application data and changes that occur to it.

Assumption Agents and coordination mechanisms may be built with assumptions on other agents in the system and on the application environment.

Deduction Some mechanisms may be based on the transformation of beliefs acquired previously by other means into new beliefs.

6.4.4 Generalised, Single Agent Preferences

The application preferences as given in the requirements need to be translated into single agent perspective priorities and preferably generalised from the particular application or goal to allow re-use of the assurance analysis. The following are translations of the case study preferences derived in Chapter 4. The new translations are shown in italics.

Edit Effect requires *reliability* of each (Contributed) goal being completed successfully.

Validate Edit requires *access to access rights* for the cooperating agents and also *security* in the acquisition of the rights.

Validate View requires *access to access rights* for the cooperating agents and also *security* in the acquisition of the rights.

Validate Access requires *access to access rights* for the cooperating agents and also *security* in the acquisition of the rights.

Rapid Data requires *speed* in obtaining results.

Accurate Data requires high *quality* of results (accuracy of predictions).

Fast Access requires high *quality* of results (fast access through suitable distribution).

Opportunism requires high *flexibility* in the choices of agent cooperators.

Time Out requires nothing in terms of coordination as it is monitored by the agent possessing the goal.

Speed Variance requires *speed* in obtaining results.

Flexibility Variance requires high *flexibility* in the choices of agents.

The translated priorities identified (*reliability, access to access rights, security, speed, quality* and *flexibility*) are all preferences but expressed in a form more suitable for comparison with application-independent coordination mechanism models, as they are more application-independent themselves. The final two preferences in the list are those application-wide preferences derived from the variations on the case study requirements given in Chapter 1.

6.4.5 The Generalised Assurance Mechanism

Assurance analysis can be described as the analysis of coordination mechanisms to highlight to what degree each fulfils application preferences. The analysis allows for a more accurate comparison because the mechanisms' use is examined in more detail than the informal IPML description allows. When analysing a single interaction, we propose using the *generalised assurance mechanism* (GAM) as a guideline for decomposing mechanisms into relevant parts for detailed analysis.

The GAM is an abstract structure used to represent the process of coordination from the viewpoint of a single agent. The structure of the GAM is shown in Figure 6.1. It can be considered as a procedural representation of how an agent uses a coordination mechanism to make decisions regarding coordination. In the GAM, we use the term *belief* to refer to an element of information possessed by an agent. There are three types of belief shown in the diagram. Agent models are collections of information that the agent using the mechanism has about other agents. A model of an agent is used to make more accurate predictions about that agent and may vary in complexity from simple recording of communications from that agent to more detailed observations of and deductions from the

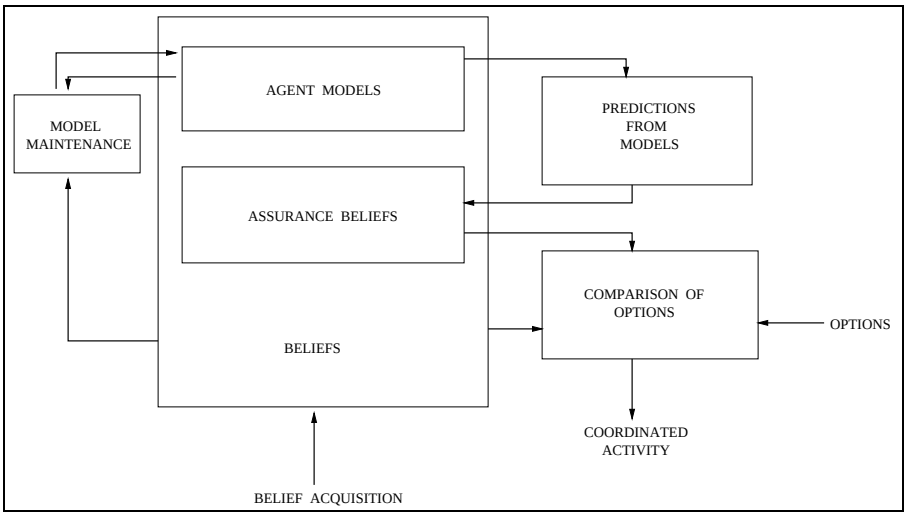


Figure 6.1: Generalised Assurance Mechanism

agent's behaviour. Assurance beliefs, as described above, are predictions on the certainty that a goal will be achieved in accordance with the agent's priorities. Other beliefs are not explicitly identified but could be information about available system resources or security protocols, for example.

After acquiring beliefs, the GAM updates the agent models based on the new information. This process is called *model maintenance*. From these models, the agent using the mechanism can assess the other agents coordinating over a goal and make assessments on the likelihood that the goal will be achieved, described as *assurance beliefs*. The agent using the mechanism can use the assurance beliefs to decide which agents to cooperate with, how to continue the cooperation or in any other way act in order to maximise the assurance belief certainty.

It can be seen that elements of the GAM as shown in Figure 6.1 match the steps of the *process of coordination* in the following corresponding order: agent models, belief acquisition, model maintenance, coordinated activity, predictions from models, assurance beliefs. The arrows show the flow of information between parts of the process.

6.4.6 Applying the GAM

In this section, we give an example of applying the GAM to analysing the interaction role of an interaction's originator. This means analysing how well coordination mechanisms match

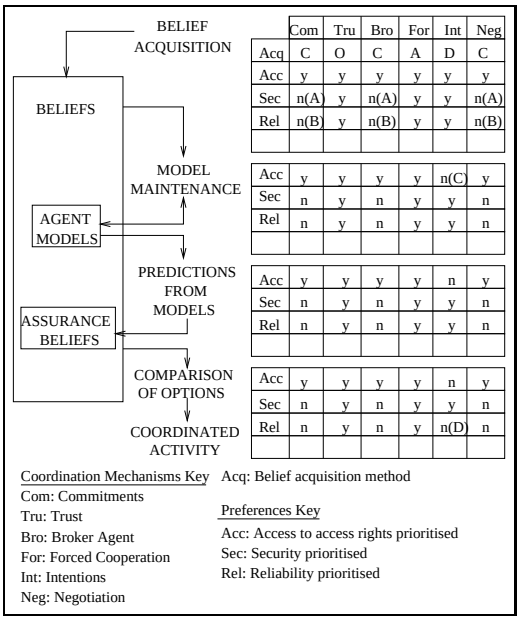


Figure 6.2: Assurance analysis of Contributed goal

the preferences of a goal from the perspective of the agent wishing to find cooperators to help achieve that goal.

The GAM abstraction is used to decompose the process of coordination to give the designer a more detailed assessment of the suitability of different coordination mechanisms. Figure 6.2 shows how this analysis is modelled for the originator of interactions over the case study's Contributed goal (which aims to add a user contribution to the weather map).

On the left hand side of Figure 6.2, the GAM is shown in rearranged form to show the main flow of activity from top to bottom. On the right hand side are tables giving assessments of coordination mechanisms at each *procedural* stage of the GAM. The top table compares coordination mechanisms at the 'belief acquisition' stage, the second table describes the 'model maintenance' stage, the third concerns the 'predictions from models' stage and the lowest shows the analysis at the 'coordinated activity' stage. Each table column is the assessment for a single coordination mechanism. The values in the table cells answer a question of the form

"Does coordination mechanism X allow preference P to be matched in stage S?"

the answer being yes or no ('y' or 'n' in the tables).

The preference and coordination mechanism names are abbreviated to fit into the table, with expansions shown under the table. The preferences included are those associated with the Contributed goal (though they are translated in the manner described in the previous section). The top row of the top table refers to the primary method of belief acquisition used for each coordination mechanism (C for communication, O for observation, A for assumption, D for deduction).

An analysis of informally described coordination mechanisms with regard to generalised preferences depends very much on the designer's informed opinion. To justify unobvious judgements in the analysis, some table entries are annotated with references to a table of explanations. For example, the broker agent mechanism (Bro) is judged not to allow security (Sec) to be prioritised at the belief acquisition stage and the entry is 'n (A)'. Explanations, including A, are given in Table 6.7. We will also explain the entries in further below.

There are useful patterns to notice in order to speed up assurance analysis. As the emphasis is on mechanisms *allowing* preferences to be matched, explanations will generally only be given when they are judged to not allow such matching, i.e. when the table entry is 'n' for 'no'. Also, if a coordination is judged unsuitable for matching a preference at an early stage (higher table) then it will be unsuitable at all later stages, i.e. a 'no' entry in one table implies a 'no' entry in the corresponding cell of all lower tables. This is because each stage of the coordination process depends on earlier ones. For example, if an agent model cannot be maintained in a way that matches a preference, then any attempt to make predictions from that model will also imply the preference is not matched.

To clarify the entries in Figure 6.2, we provide the reasoning for each coordination mechanism (column) in turn below. We refer to other assurance analyses shown in Figure 6.3 for the originator of the Speed Viewed goal, Figure 6.4 for the originator of the Accuracy Viewed goal and Figure 6.5 for the originator of the Set Access goal. The explanatory notes are given in Table 6.7.

Commitments

In the analysis, the designer has judged that there is nothing inherent in the commitments mechanism that prevents access to access rights. If an agent commits to achieving the goal in cooperation with the originator, then it is assumed that it, or an agent it delegates to, has

access to access rights else the commitment would not be made. Therefore all Acc entries are 'yes'.

However, the mechanism prioritises neither security nor reliability in an open system through its use of communication to acquire commitment information. The possibility passing goals to a third agent in an open system is inherently less secure than achieving it itself (for example), as information could be submitted anywhere in the open system. Also, reliability is reduced as open system agents can lie about or misjudge their commitments. Therefore, Sec and Rel entries from belief acquisition onwards are 'no' for this mechanism.

Trust

The trust-based mechanism overcomes the reliability problems of commitments alone by monitoring the performance of cooperators and only choosing the most reliable. Security is also enhanced by monitoring whether information is passed on to untrustworthy (insecure) third agents. Both forms of monitoring rely on building up models assessing how well agents match the application preferences. All entries for this mechanism are 'yes' in this analysis.

Broker Agent

Using a broker agent is, in this case, similar to getting a commitment directly from a service provider. Again, access to accesse rights is adequate but security and reliability could be compromised. The entries are the same as for commitments.

Forced Cooperation

Using a forced cooperation mechanism, the designer provides and assigns a single agent to always take on the goal when it appears in the system. The particular preferences analysed in Figure 6.2 all require some amount of reliability in operation, for which a forced cooperation mechanism is intrinsically suitable. As both the agent forced to cooperate and the access rights to the weather map are provided by the designer, there is no problem in ensuring access. The activity of the agent can also be made secure and reliable (even if not as high quality in other ways to other agents in the open system). Therefore, all entries are 'yes' for this mechanism in this analysis.

Intentions

Using the intention mechanism, the originator agent performs the actions necessary to achieve the goal itself. As with forced cooperation, this can be secure. However, there is a problem at the stage where the agent builds up information on the user attempting the goal as the agent playing this interaction role may not have access to the access rights itself. Similarly, at the stage of acting on the goal, the agent may not have access to the resources to achieve the goal itself. These 'no' entries can be seen in the model maintenance and coordinated activity stages.

Negotiation

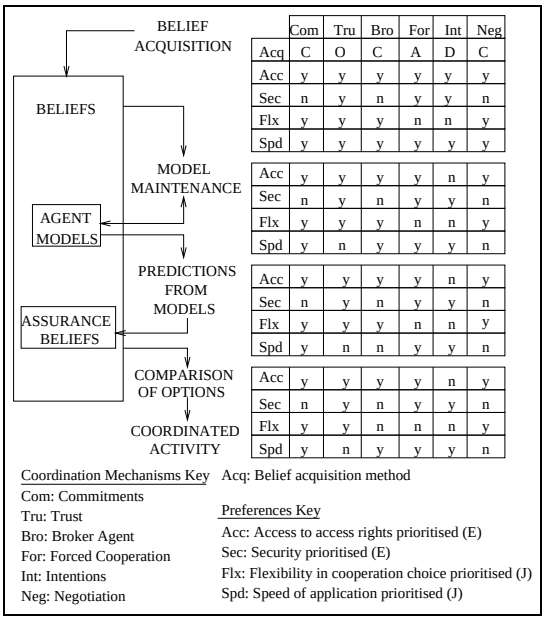
While negotiation could help improve some qualities of a solution, the security and reliability problems of communication-based coordination exist for negotiated agreement as much as for commitments, so the entries are the same.

Further Notes

To get a further feel for how this analysis can help evaluate mechanisms for comparison, look at Figures 6.3, 6.4 and 6.5. It can be seen, for example, that trust-based mechanisms are poor where speed is a priority as maintaining trust information in agent models may take substantial time. Also, while being slower, negotiation prioritises quality of solution more than simple commitments. The rest of the case study's assurance analysis is provided in Appendix A.

6.4.7 Coordination Mechanism Decisions

The choices for coordination mechanism are informed by the assurance analyses. In general, the mechanism chosen for a goal is one of those most likely to allow the goal's preferences to be followed, i.e. the one with the most 'y's at the end of the analysis. For example, the case study application designer may decide that a trust-based or forced cooperation mechanism would be most suitable for agents originating cooperation on the Contributed goal based on the analysis in Figure 6.2.



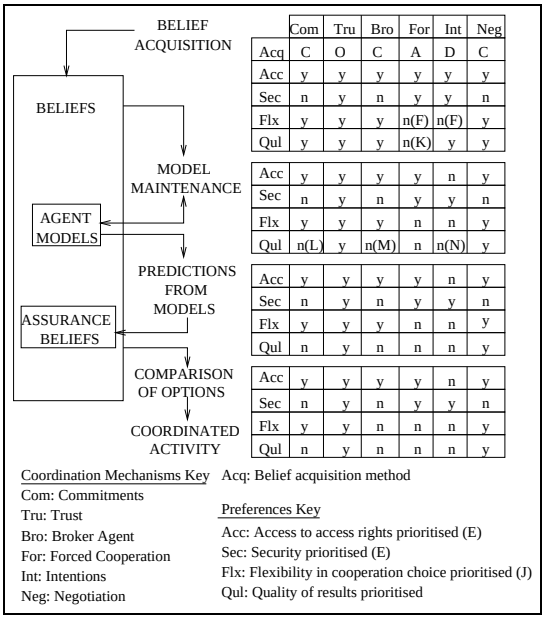
	Com	Tru	Bro	For	Int	Neg
Acq	C	O	C	A	D	C
Acc	y	y	y	y	y	y
Sec	n	y	n	y	y	n
Flx	y	y	y	n	n	y
Spd	y	y	y	y	y	y
Acc	y	y	y	y	n	y
Sec	n	y	n	y	y	n
Flx	y	y	y	n	n	y
Spd	y	n	y	y	y	n
Acc	y	y	y	y	n	y
Sec	n	y	n	y	y	n
Flx	y	y	y	n	n	y
Spd	y	n	n	y	y	n
Acc	y	y	y	y	n	y
Sec	n	y	n	y	y	n
Flx	y	y	n	n	n	y
Spd	y	n	y	y	y	n

Coordination Mechanisms Key
Com: Commitments
Tru: Trust
Bro: Broker Agent
For: Forced Cooperation
Int: Intentions
Neg: Negotiation

Acq: Belief acquisition method

Preferences Key
Acc: Access to access rights prioritised (E)
Sec: Security prioritised (E)
Flx: Flexibility in cooperation choice prioritised (J)
Spd: Speed of application prioritised (J)

Figure 6.3: Assurance analysis of Speed Viewed goal.



	Com	Tru	Bro	For	Int	Neg
Acq	C	O	C	A	D	C
Acc	y	y	y	y	y	y
Sec	n	y	n	y	y	n
Flx	y	y	y	n(F)	n(F)	y
Qul	y	y	y	n(K)	y	y
Acc	y	y	y	y	n	y
Sec	n	y	n	y	y	n
Flx	y	y	y	n	n	y
Qul	n(L)	y	n(M)	n	n(N)	y
Acc	y	y	y	y	n	y
Sec	n	y	n	y	y	n
Flx	y	y	y	n	n	y
Qul	n	y	n	n	n	y
Acc	y	y	y	y	n	y
Sec	n	y	n	y	y	n
Flx	y	y	n	n	n	y
Qul	n	y	n	n	n	y

Coordination Mechanisms Key
Com: Commitments
Tru: Trust
Bro: Broker Agent
For: Forced Cooperation
Int: Intentions
Neg: Negotiation

Acq: Belief acquisition method

Preferences Key
Acc: Access to access rights prioritised (E)
Sec: Security prioritised (E)
Flx: Flexibility in cooperation choice prioritised (J)
Qul: Quality of results prioritised

Figure 6.4: Assurance analysis of Accuracy Viewed goal.

Code	Note
A	Security in communication-based coordination is worse than for observation, assumption or deduction as there is another stage (the communicating agent) that has to pass through, beyond the security provided by the infrastructure.
B	Communication-based mechanisms rely on accurate information in messages passed to them, which leaves an agent using the mechanism unable to reliably achieve the goal. Assumption is also unreliable in general, but not in the case of forced cooperation, where the reliability is explicitly determined by design.
C	The agent achieving the goal does not necessarily have access to the rights itself.
D	The agent does not reliably have the necessary capability to act on the goal.
E	See the Contributed goal diagram (Figure 6.2) for annotated analysis of this preference.
F	Assumption and deduction limit knowledge to products of that already known, so are not as flexible.
G	Building up trust models may take significant amounts of time.
H	Negotiating may be significantly time consuming.
I	Broker agents take on the choice of collaborators and so limit the flexibility of choice in the originator.
J	See the application preferences diagram (Figure A.22) for annotated analysis of this preference.
K	Assumption will not offer the highest quality options if the options in the open system improve beyond those assumed.
L	Commitments provide no quality information.
M	Brokers provide no quality information.
N	Intention only models the agent itself so it does not address quality in other agents.
O	See the Accuracy Viewed goal diagram (Figure 6.4) for annotated analysis of this preference.

Table 6.7: Explanatory notes for the assurance analysis diagrams.

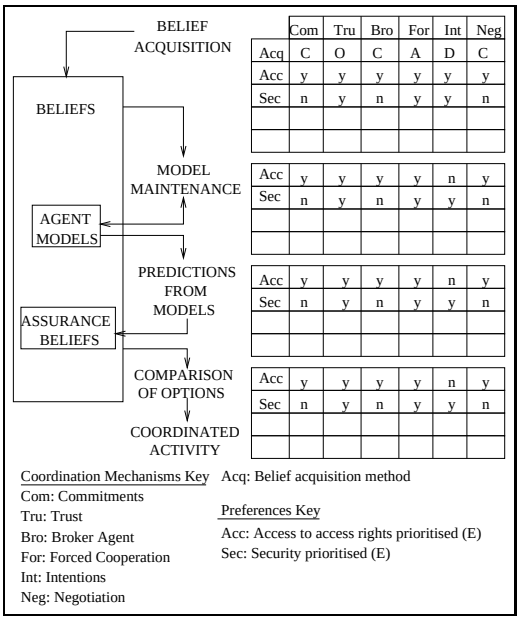


Figure 6.5: Assurance Analysis for Set Access goal.

6.4.8 Re-using Analysis Information

The designer does not need to use assurance analysis to analyse every interaction role. The aim of assurance analysis is to provide a more detailed procedural model to aid in making a design decision on the coordination infrastructure part. It is only necessary where the choice of coordination mechanism is not clear, though due to the complexity of information surrounding coordination it may be necessary in most cases.

There are ways in which the designer can reduce the amount of analysis required. If two goals have similar preferences then the coordination requirements are likely to be similar and so the same coordination mechanisms will be best. If the goals in the application are all similar or there are few preferences then a system-wide coordination model, as described in Chapter 4, could be chosen based on the assurance analysis of one goal.

Furthermore, the same preferences may be associated with more than one goal. There is no need to reanalyse coordination mechanisms with regard to this preference every time it appears. For example, the analysis of mechanisms for the 'Access to access rights' preference as made for the Contributed goal originator in Figure 6.2 is the same as for the Speed Viewed goal originator in Figure 6.3. The rows for the preference are copied between analyses.

The most significant form of re-use, however, is between application designs. If goals of two applications have similar preferences, one analysis can be re-used for the other.

6.5 Summary

In order for the design of an opportunistic application to be justified, the services used by the application should be those that best match the application preferences. The mechanism by which an agent chooses which of the available agents and non-autonomous services to interact with and in what way is a *coordination mechanism*. Coordination mechanisms can have their own costs in order to use them, so this factor must also be examined in selecting which would be most suitable for each interaction role in a design.

A coordination mechanism is a type of infrastructure part model, as described in the previous chapter. Therefore, they are described in a re-usable design pattern form in order for designers using our methodology to select between them.

In terms of coordination, the restrictions that are placed on an agent to create a justified design are conflicting with the flexibility required for opportunistic behaviour. In *agent interaction analysis* we use the idea of *flexibility bias* to describe the restriction of agent activities, including coordination, to as minimised a level as the requirements allow. In practice, this means ensuring the following are true of a design.

- Goals are always achievable by the application (except where system failures make it impossible);
- There are agents that are tailored to particular goals;
- The agents tailored to a goal are more likely interact over those goals than other agents, but may not do so exclusively.

Coordination can be a complex process involving different stages with different priorities and costs. Where IPML descriptions are inadequate for the choice of infrastructure part models, supporting analysis processes can be supplied to the designer (by any party). In this chapter, we have presented the *assurance analysis* process which provides a framework for the detailed (and re-usable) analysis of coordination mechanisms when compared to individual preferences. Assurance analysis is based on the observation that agents operating

within an open system wish to use coordination mechanisms to increase their *assurance* that others will achieve goals in ways that the application requirements state.

For detailed analysis to take place we separate the concerns of the coordination process (from the perspective of a single agent in an interaction). The process of coordination is divided into four stages: acquisition of information (beliefs) on other agents, updating of that information when appropriate, analysis of the information to make coordination decisions and activity based on the analysis. The division allows different aspects of each coordination mechanism to be drawn out to make comparison clearer.

The next chapter examines how the choice of mechanism for each interaction role affects the design of the multi-agent system as a whole, and provides a way to evaluate the effectiveness of our approach.

Chapter 7

Collation and Evaluation

7.1 Introduction

The previous two chapters have concentrated on guiding the designer in deciding the best form of agent to play each role in the interactions over application goals. This included choosing the mechanisms by which the agents comprising the application would best use to coordinate with each other. However, this is not the completed model that the designer is aiming for, i.e. a specification of the agents which should be added to the open system in order to realise the application.

In this chapter we consider the final design both in terms of its construction and its evaluation. The stage of the methodology concerned with taking the choices of model for infrastructure parts and combining them into a final design in terms specific agents is called *collation*. After completing the preferences analysis stage with a few important decisions in Section 7.2, we describe and demonstrate the collation process in Section 7.3.

Evaluating whether our approach fulfills its aims is problematic as the domain (open systems) is one in which there are characteristics change during the lifetime of the application. Along with the fact that there is no reason to believe two designers would come up with the same design from the same requirements, this means that the diagnostics of any particular application are not repeatable results and provide no real indication of the suitability of the design approach [53, 109]. Therefore, we use the concept of *traceability* to judge whether the case study design is truly based on the requirements, i.e. that it is

justified, and that it is opportunistic as far as possible. This analysis is given in Section 7.4.

Further evaluation of the methodology in general can be made by comparing it with the software engineering principles given in Chapter 2. This evaluation aids in judging whether it is an effective methodology in engineering terms. The discussion is presented in Section 7.5, and the chapter conclusions are summarised in Section 7.6.

7.2 Completing Interaction Role Design

Assurance analysis addresses the comparison of coordination mechanisms for interaction roles. However, other aspects of interaction roles were identified in Chapter 5. These were collectively called *agent infrastructure parts* and we will complete choices of model for these parts for the case study application in the sections below.

7.2.1 Goal Adoption and Capability Discovery

In tailoring interaction roles, a designer must address when agents playing those roles would best adopt the goal that the interaction intends to achieve. This is called the *adoption* mechanism, as the agent can be said to adopt goals by volunteering to take them on. In our case study, we allow a choice between *benevolence* (the agent always adopts a goal offered when capable), *1-goal benevolence* (the agent only adopts a new goal when old ones have been achieved) and *payment* (the agent only adopts a goal when offered a reward). The most flexible of these for the application as a whole is benevolence, so the designer must justify other choices (following the flexibility bias). In Table 7.1, the adoption mechanism for each originator interaction role is given with explanations where appropriate. For the IPML descriptions of the mechanisms see Appendix A.

Another mechanism required for interactions to take place is discovery of capable agents. We distinguish here between a *broadcast* mechanism where requests are distributed out over the open system with information for volunteers to respond to, and using a *broker agent* as a discovery service. In Table 7.1, the only originator interaction role justifying the extra costs of a broker agent is the Prediction goal for which it is critical that the best agent receives the request for cooperation, rather than relying on unreliable scope of broadcast. IPML specifications of the mechanisms are given in Appendix A.

Interaction Role	Coordination	Adoption	C-A Mapping
Contributed (Originator)	Speed Variation: Forced cooperation (fast engagement of known cooperators), Interoperation Variation: Trust (does not limit choices)	Benevolence	Broadcast
Speed Viewed (Originator)	Commitments (speed is the most important preference to this goal, so commitments are preferable to trust or forced cooperation)	Benevolence	Broadcast
Accuracy Viewed (Originator)	Trust	Benevolence	Broadcast
Set Access (Originator)	Speed Variation: Forced cooperation (fast engagement of known cooperators), Interoperation Variation: Trust (does not limit choices)	Benevolence	Broadcast
Redistributed (Originator)	Negotiation (other contributor nodes are trusted and their are valued objects to be negotiated over, so negotiation is preferable to trust)	1-Goal Benevolence (only one redistribution of data should occur at a time)	Broadcast
Access Denied (Originator)	Commitments (commitments are applicable to both variations and are a simple, fast solution for interacting with other trusted agents)	Benevolence	Broadcast
Warned (Originator)	Commitments	Benevolence	Broadcast
Map Edited (Originator)	Commitments	Benevolence	Broadcast
Success (Originator)	Commitments	Benevolence	Broadcast
Prediction (Originator)	Commitments	1-Goal Benevolence (predictions may take a substantial amount of time so should be forced to be distributed)	Broker (high quality required so informed discovery should be encouraged by allowing registration)
Displayed Prediction (Originator)	Commitments	Benevolence	Broadcast
Rights Edited (Originator)	Commitments	Benevolence	Broadcast
Least Accessed (Originator)	Commitments	Benevolence	Broadcast
Moved Data (Originator)	Commitments	Benevolence	Broadcast

Table 7.1: Infrastructure part choices for case study originator interaction roles

Placeholder	*Capability*
Access Denied (Local Actor)	Ability to check access rights for authorisation
Warned (Local Actor)	Ability to provide the user with a warning that a triggered goal was unsuccessful.
Map Edited (Local Actor)	Ability to edit the current weather map.
Success (Local Actor)	Ability to provide the user with an acknowledgement of an operation's success.
Prediction (Local Actor)	Ability to make a weather prediction for a given location and time.
Displayed Prediction (Local Actor)	Ability to provide the user with the results of a prediction in a suitable form.
Rights Edited (Local Actor)	Ability to edit the access rights for a user.
Least Accessed (Local Actor)	Ability to determine that section of the weather map data stored locally that is accessed least by local agents.
Moved Data (Local Actor)	Ability to move a part of the locally stored weather map data to another point in the system.

Table 7.2: Capabilities of local actor interaction roles

7.2.2 Local Actors

The most suitable agents to achieve goals may not be provided by the application designer, e.g. in our case study, sophisticated predicters may exist already in the open system. However, designers will often want to ensure a minimum functionality is available, regardless of the reliability of third-party service by providing it themselves. Agents able to perform the actions necessary to achieve goals provided by the application designer are called *local actors*. They play the interaction roles of agents committing, negotiating etc. to achieve goals with the originators. In the case study requirements it is explicitly stated that "the functionality of the application should be available locally at each node" so the designer should provide local actors for all goals. We list these capabilities in Table 7.2.

7.2.3 Support Required for Chosen Mechanisms

As the designer has chosen to use a broker agent to find capable agents for our case study's Prediction goal (see Table 7.1), the broker becomes a third interaction role in the interaction over that goal and is specified below.

Prediction (Broker)

Coordination Broker (self)

Adoption Benevolence

Capability to Agent Mapping Broker (self)

Negotiation has been chosen as the coordination mechanism for the Redistributed goal originator and, as the algorithm implementing this coordination mechanism is dependent on the goal (see IPML description in Table 6.6), we need to provide a suitable algorithm. The approach chosen in goal decomposition (see Chapter 4) is to move the least accessed part of the weather map data at a node to a more suitable node. The preference for the Redistributed goal is to prioritise fast access to data, which means ensuring the data stored on a node is that which is most accessed by that node. The algorithm is as follows.

1. Identify the least accessed parts (locations) of the weather map data (Least Accessed goal).
2. Offer one of the least accessed part to other nodes.
3. Receive numeric bids for the data part. The bids are proportional to the number of accesses made on the data by the bidding node. Bidders may include in their bid for one part, the rejection of another part they previously accepted in this redistribution, which is then returned to those parts to be distributed. This will happen if the latest part is of more use (more accesses are expected) than the previously accepted part.
4. The highest bid is accepted and the part transferred to the node containing the bidding agent.
5. If more parts are to be redistributed, the agent returns to step 2 to negotiate on another.

Depending on the requirements variation, up to three goals may be best served by a trust-based mechanism, where agents are assessed to judge their competence and reliability for the future. The trust mechanism IPML description (Table 6.3) notes that a scheduler is required to allow the agent to wait for several offers before one is chosen. A model therefore has to be chosen for the 'scheduler' infrastructure part. For brevity, we will not give a full

IPML description but simply state here that the scheduler model available allows the agent to postpone continuing the algorithm for a specified duration, during which the agent can take part in other activities, i.e. receiving proposals from volunteering agents.

7.3 Collation

In Chapter 5, we mentioned that there should be some process to highlight and resolve interdependencies between infrastructure part models. Of particular concern, as suggested in Chapter 6, is ensuring the coordination mechanisms of individual agents work together to allow an application to function in an organised way. We call this process *collation*.

Other concerns should be addressed when unifying the infrastructure parts into a final design. As discussed in Chapter 3, the designer wishes ultimately to know the implementable form of agents to be added to the open system. It was also argued that the existence of agents could be costly and the number added would preferably be low. These matters will be addressed in the description of collation.

As the organisation of agents adds their collective action in implementing an application, collation can be seen as a supporting analysis process for comparing alternative models for the *organisation* (as described in Chapter 2). The design of the collation process aims, as with all of *agent interaction analysis*, to enforce justification by the requirements and allow flexibility for opportunism.

Comparable processes can be seen in other methodologies. For example, in [18] (Bussman et al.) decision points are collated into roles. A comparable process is used in Gaia [102], where roles are collated into agent types balancing *coherence* of the resultant agent types with efficiency once implemented.

7.3.1 Organisations

The *organisation* of the agents within the application, as discussed in Chapter 2, defines how resources and responsibilities are distributed amongst the agents. Whilst some methodologies take organisations or their component roles to be primary analysis components [102, 107], we use agent interactions.

In the case of the agent organisation, the potential ways of distributing the resources and capabilities to achieve goals is dependent on the application requirements because the

resources and required capabilities themselves are. Therefore, the models to choose between are not application-independent but derived from the choices made for other infrastructure parts.

For example, in our case study an organisation could contain one role for agents coordinating editing (of the weather map and access rights) and one role for agents coordinating viewing predictions. Alternatively, the organisation could contain one role for agents coordinating operations on the weather map (editing and viewing) and one role for agents coordinating operations on access rights (set and check). The differences between these implementations will effect in the long term how well application preferences are matched, and this is precisely why justification is required rather than an arbitrary decision.

7.3.2 Global Application Properties

In related work [25], Davidsson and Johansson identify several *performance attributes*, comparable to system-wide preferences in our approach, such as load-balancing, responsiveness and modifiability. These are used to decide on the best architecture, as a whole, for the application.

7.3.3 Flexibility Bias and Redundancy

Flexibility bias requires each interaction role to be tailored to best achieving the goals of the interaction. However, it also demands that the flexibility be reduced where justified by the requirements which, in the case of the organisation as a whole, means matching preferences that are application-wide (applicable to all goals). There may be domain-specific application-wide preferences but some others will be relevant to most applications.

Reducing the number of roles is one useful preference which may be taken into account. Not only does this make the organisation more comprehendable but it also ensures less roles that are incompatibile so that fewer agents are required to play all the roles.

Reducing unhelpful redundancy is likely to be an implicit preference of most applications. We can see that the collation process has some general preferences, regardless of the application-specific ones.

Low replication To reduce unhelpful redundancy, the organisation should not replicate functionality without reason.

Close approximation If a model has been chosen for an agent infrastructure part based on the preferences, then the designer should attempt to ensure the agents in the implementation contain that model as far as is possible after collation, i.e. interaction roles should be tailored to match goal preferences.

Ease of integration To speed and simplify the development process, two infrastructure parts should only be merged if it easy to do so.

The third factor, ease of integration, is of more importance to maintenance and extension, so we do not discuss it further here. Later in this chapter, we show how it is useful as part of the methodology's approach to maintenance and extension.

7.3.4 Case Study

As described in Chapter 3, we do not assume that the agents taking part in interactions are all provided by the application designer. For the application to act opportunistically, each interaction role can be played by any agent in the open system unless otherwise restricted.

When all the agent interaction roles have been identified and agent infrastructure parts have been chosen, the designer must determine how they comprise the final organisation of the application set. As described above, the designer could examine how well different organisations best fit the requirements by examining *low replication*, *close approximation* and *ease of integration*.

For low replication the following five alternative organisation models are identified, starting with the lowest replication and becoming less suitable.

1. Merge all the interaction roles into one agent.
2. Merge the interaction roles into two agents:
   (a) An originator agent acting as the originator for all goals and coordinating accordingly.
   (b) An local actor agent able with the capability of all local actors.
3. Merge the interaction roles into ten agents:
   (a) An originator agent for the Contributed and Set Access goals (as they share all mechanisms).

(b) An originator agent for the Accuracy Viewed goal.

(c) An originator agent for the Redistributed and Prediction goals (as they share adoption mechanisms).

(d) An originator agent for all other goals (as they share all mechanisms).

(e) A local actor agent for the Access Denied and Rights Edited capabilities (as they share resources required).

(f) A local actor agent for the Warned, Success and Displayed Prediction capabilities (as they share resources required).

(g) A local actor agent for the Least Accessed and Moved Data capabilities (as they share resources required).

(h) A local actor agent for the Map Edited capability.

(i) A local actor agent for the Prediction capability.

(j) A broker agent for the Prediction goal .

4. Merge the interaction roles as above but separate the originator for the Redistributed and Prediction goals.

5. Separate all interaction roles into different agents.

The originators and local actors are separated above because they provide different functionality (the former adds coordination, the latter gives capability when it is lacking in the rest of the system). Decisions to merge or separate originators are based on the similarity of decisions on infrastructure parts given in the previous section. Decisions to merge or separate local actors are based on the similarity of local resources required. A decision to merge an originator with an local actor would only be taken where it was clearly justified to keep the coordination and action on a goal together in one agent, as otherwise we are unnecessarily binding the coordination needed in the future of the application to the possibly temporary capability to act locally. The obvious justification for merging an originator with a local actor would be if the coordination mechanism for an originator was chosen to be to use *intentions*, where the agent acts on the goal itself.

For close approximation the following five alternative organisation models are identified, starting with the lowest replication and becoming less suitable.

1. Separate all interaction roles into different agents.

2. Merge only the originators for the interaction roles using exactly the same coordination, adoption and capability-to-agent mapping mechanisms (results in six originator agents, one broker, nine local actor agents).

3. As above, but also merge local actor agents in the way described by suggestion 3 for low replication above, i.e. Access Denied capability merged with Rights Edited etc.

4. As above, but also merge the Contributed orginator with the Set Access originator (equivalent to suggestion 4 for low replication).

5. Merge all the interaction roles into one agent.

The lists of possible models for low replication and close approximation exclude many others that are obviously no better than the ones given. To be more exact, those excluded do not significantly aid in achieving one priority or the other. For example, if the Contributed originator was merged with the Access Denied originator, some approximation would have to be made so that the merged agent had one coordination mechanism, one adoption mechanism and one mapping mechanism, but there would be no reason to not merge that agent with the originators for Warned, Map Edited etc. as nothing further is approximated in merging them and there is lower replication if it is done.

On examining the suggestions above, the designer may choose suggestion 4 in the two lists, as this ensures no approximation of mechanisms is required but the interaction roles are merged where possible otherwise, i.e. it is the model with the most justified flexibility. This results in an multi-agent system with eleven agents (five originators, one broker and five local actors) shown in Table 7.3. In the table we include the which agent will act as a designated local adopter for the GUI to pass triggered goal instances to (this requires a benevolent adoption mechanism). These are chosen based on the suitability of the agent (how close it approximates the infrastructure parts needed). We also include which goals the agent will adopt if offered. For flexibility, this is kept as wide as possible but may be limited by the appropriateness of the coordination mechanism or capability-to-agent mapping mechanism, e.g. forced cooperation requires agents that will be forced to adopt the goal.

	Coordination	Adoption	C-A Mapping	Capability	GUI Triggering	Will Adopt
1	Forced Cooperation	Benevolence	Broadcast	None	Contributed, Set Access	Contributed, Set Access
2	Trust	Benevolence	Broadcast	None	Accuracy Viewed	All goals
3	Commitments	Benevolence	Broadcast	None	All other triggered goals	All goals
4	Negotiation	1-Goal Benevolence	Broadcast	None	None	Redistributed
5	Commitments	1-Goal Benevolence	Broker (11)	None	None	Prediction
6	Not originator	Benevolence	Broadcast	Warned, Success, Displayed Prediction	None	Warned, Success, Displayed Prediction
7	Not originator	Benevolence	Broadcast	Map Edited	None	Map Edited
8	Not originator	Benevolence	Broadcast	Prediction	None	Prediction
9	Not originator	Benevolence	Broadcast	Access Denied,	None	Access Denied, Rights Edited
10	Not originator	Benevolence	Broadcast	Least Accessed,	None	Least Accessed, Moved Data
11	Broker (self)	Benevolence	Broker (self)	None	None	No goals

Table 7.3: Agents produced by collation with cardinality (speed variation)

7.3.5 Agent Types

If the application is likely to process several goal instances at once, particularly if they are of one type of goal, the designer may decide to use the results of the collation as a set of *agent types* (this term is used for an equivalent concept in Gaia [102]). The designer would then build useful redundancy and replication into the application by instantiating several agents from each type, e.g. eight Redistribution originators in the initial design.

If this option was taken with the example application, the designer may reason as follows.

- The local user is likely to trigger goals at a reasonably slow pace, so there is only need for one agent of each of the Contributed/Set Access and Accuracy Viewed originator agent types.
- Only one redistribution of local data should be occurring at a time to prevent conflicts, so there only needs to be one Redistributed originator, and one local actor for Least Accessed / Moved Data.
- The Prediction orginator and the Prediction, Map Edited and Access Denied/Rights Edited local actors may be used by external sources so the number should be chosen according to the expected number of collaborating weather institutions, to prevent overloading a single agent.
- There should only be one broker for the Prediction goal, as otherwise predictors may not all be registered in one place.
- The originator for other goals may be used several times during the attempt of one triggered goal instance. For example, when Set Access is triggered, the same agent type acts as originator for Access Denied, Warning, Rights Edited and Success. It may be best to have three or four of this agent type to prevent overloading and distribute goals.
- The local actor for Warning, Success and Displayed Prediction is only used once per local triggered goal and displays information to the local GUI, so only one agent is needed.

We add numbers of each agent type in Table 7.4.

	Coordination	Adoption	C-A Mapping	Capability	GUI Triggering	Will Adopt	No.
1	Forced Cooperation	Benevolence	Broadcast	None	Contributed, Set Access	Contributed, Set Access	1
2	Trust	Benevolence	Broadcast	None	Accuracy Viewed	All goals	1
3	Commitments	Benevolence	Broadcast	None	All other triggered goals	All goals	4
4	Negotiation	1-Goal Benevolence	Broadcast	None	None	Redistributed	1
5	Commitments	1-Goal Benevolence	Broker (11)	None	None	Prediction	10
6	Not originator	Benevolence	Broadcast	Warned, Success, Displayed Prediction	None	Warned, Success, Displayed Prediction	1
7	Not originator	Benevolence	Broadcast	Map Edited	None	Map Edited	10
8	Not originator	Benevolence	Broadcast	Prediction	None	Prediction	10
9	Not originator	Benevolence	Broadcast	Access Denied,	None	Access Denied, Rights Edited	10
10	Not originator	Benevolence	Broadcast	Least Accessed,	None	Least Accessed, Moved Data	1
11	Broker (self)	Benevolence	Broker (self)	None	None	No goals	1

Table 7.4: Agents produced by collation (speed variation)

In the next chapter, we highlight the points in the example application that illustrate how this methodology has justified the entire design from the requirements, and how this compares to the approaches of other methods. We also consider the issues involved in implementing the application and agent infrastructure parts.

7.4 Evaluation

The results of applying our methodology, *agent interaction analysis*, to a sample set of requirements are shown throughout this thesis and collected in Appendix A. We can extract a lot of useful evaluations on our approach by examining the results and will do so in the following sections. However, analysis of the case study results is not an analysis of our

approach as a whole.

To evaluate the approach, we judge it by the initial aims. We summarise the primary questions posed in Chapters 1 and 2 below.

- How can a designer create an application that re-uses software available in an open system?
- How can the restrictions required for justification of a design and the flexibility required for opportunism best be balanced?
- How can any solution to the above problems be generally and consistently applicable?
- How can an agent-oriented methodology guide designers in creating justified, opportunistic designs?

We examine how well our approach answers these in the sections below.

7.4.1 Case Study Evaluation: Justification

In Chapter 3 we suggested that judging whether a methodology allowed and guided the production of *justified* designs could be determined inductively. In this way we identified two *methodology capabilities*: consistent *identification* of the analysis entities from the requirements and *connection* from the justification of design decisions at one stage to the design decisions of later methodology stages.

We suggested that, following standard requirements analysis techniques, we could identify *goals* and *preferences* (non-functional goals) from the requirements. In Chapter 4 we showed how this could be achieved using scenario (event trace), entity and goal analyses.

To connect the identified goals and preferences to the initial interaction-oriented design, we suggested that goals would be mapped one-to-one onto interactions. This specified that each goal is seen as the result of cooperation between agents, with a single agent 'cooperating with itself' in the limiting case.

In Chapter 5, the designer modularised the infrastructure parts which agents used (including those that comprise the agents) to achieve the system goals, and used IPML to specify possible choices. The designer chose from IPML models on the basis of the preferences of each goal, or each agent interacting to achieve the goal in the case of coordination mechanisms (in Chapter 6). This ensured that the design decisions were connected to the

Agent	Coordination	Will Adopt
1	Forced Cooperation	Contributed Weather Data, Set Access Rights
2	Trust	All goals

Table 7.5: Agents produced by collation

requirements. The designer then *collated* the agent models to streamline the coordination model.

While this analysis shows that we have attempted to provide connection, and so justification, in the methodology, it is worth proving it for a subset of the case study application by tracing back from the later design decisions to the requirements.

Traceable Design

In this section we trace back from part of the final organistion of the case study, as shown in Table 7.5, to the original requirements. We wish to show that each design decision taken to reach that stage was justified.

The first of the example interaction roles in Table 7.5, includes the functionality identified as most suitable for originators of two goals: Contributed and Set Access. The designer has chosen to merge the interaction roles designed to coordinate over these goals, so the organisation contains one interaction role tailored to providing this functionality rather than two. By simply identifying roles of agents in the original requirements, the designer could, for example, have chosen to implement two agents that separately dealt with the two goals, or possibly identified one agent to deal with operations involving access to the weather map (the Contributed, Speed Viewed and Accuracy Viewed goals) and another to change the access rights. To see why the decision to choose this particular organisation is justified, we can reason (trace) backwards from the *collation* stage, at which the final organisation is chosen.

1. At the start of the collation stage, the designer has decided that an agent tailored to coordinating over the Contributed goal would have an architecture which used forced cooperation to coordinate, benevolence in adopting goals, no capability-to-agent mapping and no capability for acting directly. The designer has also decided that an agent tailored to coordinating over the Set Access goal would have exactly

the same architecture (agent infrastructure parts). These are decisions on the most suitable architectures of *interaction roles* for agents initiating cooperation over the goals (originators). *The decisions on the choice of most suitable architecture for each interaction role are the results of analysis into how the agents can be tailored to match the requirements, and should, therefore, inform the choice of organisation.*

2. In collation, the designer decides what agents make up the final organisation based on the architecture of the interaction roles. It would be likely that at least one agent matching the architecture of the Contributed interaction role is implemented so that when an agent, or the user, wishes to achieve an instance of the Contributed goal, there will be an agent tailored to doing so in the open system. The same is true for the Set Access interaction role. However, as the two interaction roles have the same architecture, having two agents with the same architecture is not necessarily the best organisational division, as having two agents will use up more resources. *Separation of interaction roles into more than one agent type in the organisation is not justified by the requirements if the architectures of the interaction roles are similar enough, as long as the architectures are themselves justified by the requirements.*

3. To see that the architectures of the interaction roles are justified we examine the preference analysis stage, including assurance analysis. The assurance analysis for the Contributed goal gives reasons why the most suitable coordination mechanisms for the goal are Trust and Forced Cooperation, based on the comparing preferences of that goal with the operations of each mechanism. The Set Access analysis comes to the same conclusion for that goal. By analysis of the application preferences, the designer could see that the Trust mechanism could take up a substantial amount of time to check results and build up trust models, so the coordination could delay the goal significantly. For the Speed Variation of the requirements, therefore, the designer chose Forced Cooperation as the coordination mechanism for both of the interaction roles. The adoption mechanism was chosen to be Benevolence for both of the interaction roles, as this allows for the most opportunism in the system and there was no reason found to restrict adoption by choosing another model. For an agent using the Forced Cooperation coordination mechanism, references to trusted cooperating agents are built-in, so no capability-to-agent mapping mechanism is needed (the model for having

no such mechanism is called Broadcast). The decisions on which model to choose for each mechanism (agent infrastructure part) were informed by the IPML descriptions. *The choice of architectures for the interaction roles was decided on the basis of the goal and application preferences* aided by comparison of IPML descriptions and assurance analysis results.

4. The Contributed and Set Access goals and their preferences were extracted from the requirements in the requirements analysis stage by examining the scenarios, entities and goals mentioned in the requirements.

Traceable Infrastructure

As with the agent infrastructure parts mentioned above, application infrastructure parts are chosen on the basis of the preferences and goals identified in requirements analysis and informed by the IPML descriptions of the potential models. For example, in the previous chapter we chose to deal with goal instances triggered from the user interface by passing them to local agents designated to achieve each type of goal, rather than passing all goals to a single interface agent. This was because the IPML description for the GUI Trigger Agent model (the one not chosen) states that it may not be suitable where the goals have widely differing preferences, which is the case in our example application.

Basis on Requirements

Another way to emphasise how the design is justified by the requirements, besides reasoning, is to observe the effect of altering the requirements in some minor way. In Chapter 1 we give two variations on the requirements: one prioritising speed, the other prioritising interoperability. The agents in Table 7.5 are derived from the Speed Variation of the requirements. However, if we analyse the Interoperability Variation of the requirements, the architecture of the first interaction role will most suitably use the Trust coordination mechanism, to allow authorised agents from other parts of the system to edit the map and access rights. It may then be suitable for the first two agents in the table to be merged into one, as they will have very similar architectures. This demonstrates a significant design difference due to alteration of the requirements.

7.4.2 Case Study Evaluation: Opportunism

Aside from justification, we also wish designs to be restricted as little as possible in its opportunism. In Chapter 3 we divided opportunism into two methodology capabilities: *flexibility* wherever allowed (not derived) by the requirements and providing ways to specify *interoperation* between the agents under development with those unknown in the open system. The latter capability is met in *agent interaction analysis* by modelling in terms of interactions between unspecified agents. To judge how well flexibility is preserved we return to the agents shown to be justified above. The two agents in Table 7.5 differ greatly in their interoperability. The first agent is heavily restricted in its operations while the second is highly interoperable. We claim that the opportunism of each is restricted only as far as the requirements demand (there is a *flexibility bias*) and the design decisions are, therefore, justified. We examine the design decisions restricting or allowing opportunism for each agent below.

1. The first agent is heavily restricted in its activity.

   (a) The preference to validate authorisation to edit the weather map and access rights and the emphasis on reliability in contributing to the map require that the agent uses only trusted agents in making alterations. This can either be achieved by checking the agents actions to determine their reliability or by always using standard cooperators chosen by the designer. In the case where speed is more important than interoperability, the designer chose the latter option (the Forced Cooperation coordination mechanism).

   (b) Using Forced Cooperation requires that references to cooperators be known in advance. Therefore, only the goals for which references have been provided can be attempted by the agent, which are Contributed and Set Access in this case. The agent will only offer to adopt these two goals.

   (c) These restrictions are all derived from the requirements (goals and preferences). All other behaviour is flexible to encourage opportunism in the system. For example, the agent uses the Benevolence adoption mechanism to allow adoption of goals from any source.

2. The second agent has very little restriction in its behaviour.

(a) The preferences on getting accurate predictions in Accuracy View operations, leads the designer to use interoperation to find the most suitable cooperators at each instance. To decide between cooperators, the agent uses a trust-based mechanism giving information on the previous likely accuracy of results.

(b) There is no reason to prevent the agent from being given the ability to coordinate over other goals, and so it is given the ability to do so.

(c) The requirements encourage opportunism in this case, and the design of the agent reflects this.

7.5 Re-use, Generality and Consistency

In Chapter 2, we discuss the principles an agent-oriented software engineering methodology should follow to be useful and consistent. We also argue for providing structure to allow future developments in agent technology to be incorporated into the process. In the previous chapter, we additionally suggest that the methodology should be able to scale up to tackle more complex applications than the one given in the example. This section examines how well the methodology achieves all these aims.

7.5.1 Re-Use

As discussed in Chapter 1, an important aim in any approach to developing open system applications is re-use of (parts of) designs. As with re-use of services in an open system, the re-use of designs allows an application to make use of the best available functionality.

Re-use of designs is encouraged in *agent interaction analysis* through following the design pattern approach. We provide a standard structure for description (IPML) of infrastructure part models that allow simple comparison between models and with the application preferences. The IPML definitions will be provided by the third-parties that originally develop the models, and so are likely to be defined with a good degree of knowledge about their appropriate use.

7.5.2 Software Engineering Principles

Several software engineering principles were described in Chapter 2, that are the result of examining the best practices of existing approaches [22, 44, 93]. We stated that achieving them in the methodological process would help to ensure that common problems of software engineering were addressed. The principles identified in [44] are given below, with details on how our methodology achieves each.

Rigour Our process follows a series of structured stages each building on the last and the resulting design is fully justified by the original requirements. It is, therefore, rigorous, though how rigorously the application as a whole is designed depends on how extensive the application modularisation is.

Separation of Concerns / Modularity The methodology uses modularisation by dividing each goal into interaction roles for agents interacting over them, the infrastructure into parts and goals into subgoals. These are independent except where relations are specifically identified by analysis or design.

Abstraction Agent interactions are very abstract, in that the specification of each states only that some unspecified agents will interact over a goal in order to achieve it. The methodology starts with agent interactions and later addresses the details of how agents would be tailored to best achieve the interactions.

Anticipation of Change The methodology errs towards opportunism and restricts the application design only when demanded by the requirements. Extension is illustrated in Section 7.5.4.

Generality The methodology uses generality by the unspecific nature of agent interactions. It is also general itself in that it uses well recognised general requirements description concepts (goals and preferences) and allows selection from whatever models are available for the infrastructure.

Incrementality The methodology is structured to allow the designer to return and edit parts of the specification with relative ease. Incrementality can take the form of maintenance (correction) of the application and iteration during the design process. These are illustrated in Sections 7.5.4.

7.5.3 Iteration

Iterative design, i.e. repeatedly working through the design process adding more functionality each time, and iteration in the design process, to add newly discovered requirements or correct mistakes, are important operations for the methodology to support. Our methodology supports iteration by reducing dependencies between elements at each stage, which helps the designer to edit the design without most of the rest of the specification being effected. The exception is collation, which is the final stage when the methodology is applied sequentially. Adding goals at the requirements analysis stage does not require changes in the decompositions of other goals or the preference analysis of those goals; adding preferences to goals affects only the decomposition and analysis of those goals; changing the decomposition of a goal only affects the analysis of the subgoals etc.

For example, the designer of the weather mapping application may initially assume the access rights of users to be pre-set. The designer can later return to the requirements analysis stage, add and analyse the Set Access goal, provide a decomposition and assurance analysis before collating the interaction roles of the Set Access goal with the interaction roles of the goals previously analysed.

7.5.4 Maintenance and Extension

Maintenance is effectively iteration after implementation, in that the designer returns to correct a pre-existing product, either the implementation, if it does not follow the design correctly, or the design, if it is found to be incorrect. If the design is altered then the implementation will be also. Extension is adding to the design rather than merely correcting it.

As collation makes justified choices on how to merge functionality, it will be affected by all changes at earlier stages. The results of collation then inform the implementation. If the design has not be implemented when it is altered, then the alteration leads to the collation being reappraised and possibly a different organisation chosen. However, if the design has been implemented, and, even more significantly, if it has begun to operate and should ideally not be stopped, then the alteration must be made to the organisation as it is implemented.

In the collation stage explanation in Chapter 5, we discussed three criteria for assess-

ing whether to merge interaction roles to form agents (or agent types): close approximation, low replication and ease of integration. We omitted the last of these from our example, as it is most applicable to maintenance and extension of applications rather than the initial design. Generally, two interaction roles are easier to integrate if they are similar in mechanisms and one of them is not already implemented.

Continuing our iteration example from above, if the designer decides to add the ability to set access rights *after* the application is running, then the ease of integrating the ability must be taken into account. For instance, if it will be easy to alter the originator for the Contributed goal to also include Set Access then collation will suggest the merged originator shown in Table 7.5. If the Contributed originator is in use and cannot be easily altered, collation will suggest a separate agent (type) to coordinate setting access rights. A local agent capable of setting rights may also need to be included in the application.

An interesting approach is proposed by Poutakidis et al. [88] for debugging multi-agent applications which could usefully be applied where particular coordination mechanisms were chosen by the designer. In their technique the running system is monitored to establish whether the interactions expected for each mechanism (the protocol) is apparent. If it is not then an error may exist in one of those interacting agents.

7.5.5 Implementation

The implementation of the design may be aided by additional detail on the structure of the agent and application infrastructure parts chosen. This could involve describing them in terms of concepts already conveniently represented in programming languages, e.g. objects or functions.

For example, implementing the GUI Trigger Agent model for graphical interface triggering of goals, specified in Appendix A, may be aided by specifying an object model showing how messages pass from the interface to an agent. The implementable representation could either be added to the Algorithms property of the IPML description or as an associated annotation.

7.6 Summary

At the end of the design process for an open system application, a designer will aim to end up with a specification of the software that can be implemented and added to the system in order to realise the application. In this chapter, we showed how the analysis performed to determine the most appropriate models for infrastructure parts could be *collated* into a final design. The final design takes the form of a set of agents.

From this point, we demonstrated that the designer, or anyone else, could *trace* backwards from any design decision to the requirements in order to demonstrate how the decisions were justified, and how they retained flexibility for opportunism. Other software engineering properties were examined in relation to our methodology to assess how well they were met.

In the final chapter we outline our main contributions and assess how the work should be progressed in the future.

Chapter 8

Conclusions

8.1 Introduction

In this thesis, we have described the problem of designing *justified* designs of *opportunistic* applications in dynamic open systems. We taken an agent-based approach to solving this problem. An application design is *justified*, in this context, if the designer is able to trace the design decisions back to the requirements that informed. For multi-agent systems, part of the justification required is to ensure that application functionality is appropriately divided between agents. *Opportunistic* applications will aim to use the most suitable (justifiable) functionality, for the activities they are engaged in, available within the open system at any time. In agent-based systems, opportunism is founded on the agents being able to make flexible decisions regarding their activities.

Coarse-grained agents can provide a useful set of functionality for localised decision-making, but they also place demands on the system resources and application infrastructure. Therefore, the way in which functionality is divided between agents, and the mechanisms chosen to achieve it, can have a significant impact on how well an agent-based application satisfies its requirements. The problem of dividing the functionality and infrastructure to match the requirements is significantly harder for applications operating in dynamic open systems, where the functionality most suitable for achieving application requirements at any particular time may come from a connected software environment or application. Expressing the structure of a proposed application of this form, in preparation for its design, without

making arbitrary assumptions about the way in which the functionality is divided, requires that agents are specified in a very minimal, unrestrictive form. We have achieved this by describing the application in terms of abstract *interactions* between unspecified agents, so that each goal of the application will be cooperated over by a set of agents, which may be implemented by the application designer or supplied by third parties.

Consistently justifiable designs require rigorous creation processes, i.e. software engineering methodologies. For the methodology to be most useful, it must be acceptable to developers, particularly in being informal, structured and related to existing technologies. We have used and extended standard requirements engineering techniques for analysis and base our descriptions of infrastructure parts on design patterns. At other stages of the methodology we have used our own notation, as the concepts involved are largely unique to the approach.

Our methodology, agent interaction analysis, contains several significant stages developed from the basis on abstract agent interactions and maintaining a *flexibility bias* to ensure that the applications produced will be opportunistic where the requirements allow. In the analysis phase, we demonstrated the identification of goals with associated preferences (non-functional goals) from the requirements and decompose the goals into more tightly defined parts. In the design phase, we modularised the infrastructure and chose models for each part based on the preferences. We also developed the agents' architectures by analysing different models to see how well they satisfy the requirements. In particular, we provided a general and detailed technique for examining the coordination of agents in open systems. Finally, we constructed the final form of the multi-agent system (its *organisation*) by reducing redundant replication in the architectures identified.

In this thesis, we have described our methodology, and the theories that underly it (Chapters 3 to 6), in the context of existing work into agent-oriented software engineering, drawing upon other useful research in agent-based system infrastructures, agent coordination and software engineering (Chapters 2). We have provided a case study in which we developed a weather mapping application and discuss how it solves the stated problem (Chapter 7).

8.2 Main Contributions

In solving the problem of designing justified, opportunistic agent-based applications, we have developed a set of ideas. By providing concrete techniques based on these ideas, we help others to apply the theories. In the sections below, we outline the distinct techniques developed and the theories underlying them.

8.2.1 Agent Interaction Analysis

We have aided the creation of justified, opportunistic agent-based applications by supplying a methodology which can be used to provide a consistent approach. Informal, structured methodologies provide ease of use and acceptability for designers, but the informality may disguise assumptions that arbitrarily restrict the design, as we suggest is true of existing methodologies in Chapter 3. We have solved this problem by developing an approach in which the designer explicitly assumes flexibility in the system unless the requirements suggest restrictions. Our methodology is called *agent interaction analysis*.

8.2.2 Agent Interactions as Analysis Abstractions

We have developed the use of abstract *agent interactions* to specify a multi-agent system's functionality, allowing people to describe such systems without having to limit which subsystems are well-defined or between which subsystems operations will be distributed. Agent interactions are particularly useful for designing agent-based systems as they provide a flexible basic form to start from, regardless of whether agent interaction analysis is then used to develop the final design. The approach we have created is used in our methodology. It can more generally be applied wherever functionality in an open dynamic system is to be specified without restricting where the functionality will reside.

8.2.3 Flexibility Bias in Designing Multi-Agent Systems

We have enabled designers to reconcile opportunistic behaviour of agents in a dynamic system with achieving high worth solutions in a worth-oriented domain, i.e. satisfying quantitative preferences as well as possible in opportunistic cooperation. To achieve this, we tailor agents in a multi-agent system to the preferences of different system goals and then provide coordination mechanisms to ensure the best tailored agents are those most likely,

but not exclusively, chosen as collaborators when a goal needs to be achieved. This theory ensures that restrictions are made to the cooperative abilities of agents only where necessary, and so the design process has a *flexibility bias*. The theory of flexibility bias that we have defined is used in our methodology. It can more generally be applied wherever a development process that balances priorities with the desire to allow for opportunistic use of unknown resources is required.

8.2.4 Assurance Analysis

We have developed techniques for tailoring the coordination of agents in an open system to the functionality they are most likely to be active in attempting. Coordination can have a large impact on whether preferences (non-functional requirements) are satisfied or not. The mechanisms to achieve coordination between agents may be complex. We have developed *assurance analysis* to separate concerns in the use of a coordination mechanism, providing the designer with a consistent approach for analysis and comparison.

8.2.5 Infrastructure Part Modelling

We have provided a design pattern-based language, *IPML*, for flexibility and informally highlighting the issues of importance for agent-based system infrastructures in open systems. Using IPML, a designer can specify and re-use parts of a multi-agent system infrastructure.

8.3 Most Suitable Applications

Our approach aims to be suitable for a wide range of open system applications but it is, of course, more suited to the design of some than others. Primarily, we argue that our methodology is well suited to the design of applications with a high number of non-functional goals (*preferences*). This is first because the preferences are used to *justify* design decisions by providing a reason why one choice, e.g. infrastructure part model, is better than another. Secondly, the preferences are used to determine when agents should *opportunistically* make use of functionality that has been added to the open system. Conversely, an application with relatively few preferences is not well suited to our approach. This is because, without preferences, there is no way for a designer to justify one design decision over another or for

an agent to choose one functionally equivalent service over another. In this case, restrictions imposed by arbitrarily chosen application structures are of no consequence.

8.3.1 Well-Suited Case Study

To illustrate the argument above we provide the requirements for an application well-suited to our approach.

A music company wishes to provide customers on the Internet with a randomised jukebox application. The jukebox will play songs drawn from any of the company's databases. There are several databases existing each of which overlap in the songs they contain but songs are sometimes added to one database before others, e.g. for songs released earlier in one country. Also, mistakes are sometimes made in the song information available in a database so an entity in one database can be more up to date than for the same song in another database. The customer should be able to grade the artists and genres of music according to their taste in their jukebox. A customer's jukebox should aim to play more songs likely to appeal to the customer as judged by these gradings compared against the song information in the databases. New songs should be included in those available to be played as soon as possible after they appear in a database, and new databases should be included as sources for a jukebox when the company creates them. The jukebox should be secure in its retrieval and storage of songs to prevent piracy, and fast enough to make it enjoyable to use.

The application described above is well-suited to our approach because it allows for some opportunism and contains many non-functional goals, e.g. playing of songs graded higher by the customer, using database entries that are the most up to date, using the most databases of those available, security and speed in retrieval of songs etc.

8.4 Relation to Other Methodologies

As the concepts used in the design of other methodologies are undoubtably useful for expressing behaviour in multi-agent systems, it is worthwhile seeing how they relate to or could be incorporated into agent interaction analysis.

8.4.1 Organisational Roles

Dynamic organisational roles, i.e. roles that can be played by different agents at different times, could be incorporated into our methodology as part of a coordination mechanism. The roles could act as transferable commitments to benevolently accept particular goals with associated restrictions and permissions. Alternatively, the set of roles to be taken up by agents could be stored by a broker from which agents can collect roles to adopt. By treating roles as a coordination mechanism, we can justify their use as part of a wider justified design.

8.4.2 Workflows and Decision Points

In agent interaction analysis, goal decomposition provides enumeration of the decision points in the application, and workflow modelling may be useful in aiding the decomposition by the designer. However, workflows pre-suppose entities between which the process can flow and agent interaction analysis deliberately avoids identifying entities until the end of the methodological process, as the methodology's aim is to provide justification of those entities' designs. This will limit the applicability of workflow modelling.

8.4.3 Societies

Societies of agents which coordinate using a single model, as in SODA, and perhaps employ other joint mechanisms, can encapsulate several agent interactions into a single set. Societies could therefore be usefully employed as a *design pattern* for agent interactions. If a set of goals are commonly identified in applications which consistently suggest a shared coordination mechanism, this could be described as a society of agents and re-used without the extensive analysis being replicated. It may also be appropriate to store societies of agents for common goal sets in an implemented form to be included into applications as components.

8.4.4 Organisational Rules

As with workflows, organisational rules require entities to define the relation between, so are perhaps not appropriate for inclusion in *agent interaction analysis*.

8.5 Problems and Further Work

There are several clear areas in which our work could usefully be extended, perhaps taking the form of extensions to the *agent interaction analysis* methodology. These are topics that are not directly relevant to the aim of designing justified, opportunistic applications, but are important in raising the usefulness of the methodology.

8.5.1 Reducing Analysis Volume

For an application to be opportunistic, it must make frequent decisions on which part of the open system will achieve each piece of functionality required. If these decisions are not made, the application cannot take opportunity of newly available services or take account of changes in the system that cause one part to become more suitable than another. However, each decision may depend on different criteria for judging justifiability, and, therefore, will have different analyses in the design process. The volume of information produced for a justified, opportunistic design will, therefore, be large. This can be seen in the case study example in Appendix A, as the number of goals and preferences grows fairly large for the simple application by the end of goal decomposition, which is where the designer determines the points at which the application makes decisions.

If an application is to be both opportunistic and justified, regardless of the methodology, the volume of analysis produced will be large compared to the complexity of the application. However, there exist techniques to reduce the amount of analysis and design in software engineering that are also applicable to our methodology. One of the techniques is *approximation*, whereby the designer sacrifices some justification for the reduction of information. This can be seen where, for example, the designer chooses to decide on a coordination mechanism for a goal without assurance analysis.

Another technique is *re-use*, where the designer applies the analysis from one application design to another application. *Agent interaction analysis* aids re-use by making the detailed analyses application-independent where possible. This can be seen in assurance analysis where generally applicable coordination mechanisms can be compared to preferences. The preferences in the comparison are also reduced to only that part that is relevant to coordination. For example, in the assurance analysis of the Accuracy Viewed goal in Appendix A, each of the six coordination mechanisms is assessed for its ability to allow the

agent using it to ensure the high quality of the solution, corresponding to high accuracy of prediction. This analysis can be reapplied to other situations and applications, such as the Redistributed goal in Appendix A, even though the specific types of quality are different.

8.5.2 Implementation Paths

The translation from a design created in *agent interaction analysis* to implementations in particular language concepts, e.g. objects, needs to be developed. All implemented components will be models described in IPML, so it would be useful to have a mapping from IPML properties to an object-oriented design. Such a mapping would provide consistency in implementation to accompany consistency in design. A different approach would be to extend agent development platforms to better express the products of our methodology. In both cases, CASE tools may usefully support the entire development process.

8.5.3 Formal Verification

It may be useful to be able to connect the informal traceability of a design to formal verifications where this is appropriate and possible. The definition of some infrastructure part models could include verified formal descriptions providing the designer with extra information for deciding between models, i.e. some applications' requirements may have included the preference for verified design.

8.6 Concluding Remarks

Users' increasing willingness to use applications which take advantage of the benefits of dynamic open systems, while exposing themselves to the inherent risks of that approach, will allow those benefits to constantly increase as applications make use of each other's functionality. It also provides an obvious application for agent-based systems. However, users will want the risks to be mitigated as much as possible, ensuring that the application will perform to the requirements. In order to prove this, a potential application should be *justified*.

In the preceding chapters, we have described a way in which justification and opportunism can be balanced by designing the system in terms of agent interactions. When an open system can contain many agents of varying functionality that disappear to later

be replaced with new and different agents, the interaction between agents becomes more important than the agents themselves. A user will initiate many of the actions that an application should perform, and this interaction with the application will be translated into many interactions between agents. By designing in terms of interactions, developers create applications that are more easily extended. As each interaction role, the infrastructure and the application organisation as a whole is tied directly to the requirements, any change in the requirements can be easily translated into a new design. Depending on whether the application can be safely altered at run-time or not, new agents may replace or be added to the existing ones.

In the end, the various techniques we have developed to realise design in terms of agent interactions are aimed at enabling *re-use*. By allowing opportunism in open systems, re-use of existing and future functionality is created. By modularising applications into infrastructure parts with models described in an application-independent way we promote re-use of design. We believe that *agent interaction analysis* can be used both as a useful agent-oriented methodology and an illustration of techniques for developing justified designs of opportunistic applications.

We argue that existing agent-oriented software engineering methodologies do not guide the designer in the creation of justified designs for opportunistic applications. In this thesis we have presented a methodology that achieves this aim and so will be useful for the development of open system applications in the future.

Appendix A

Case Study Results

This appendix contains all diagrams and tables resulting from the example application summarised and explained throughout the thesis. For ease of examination, the terms used in this appendix are defined in Section A.1, though they also occur in the relevant thesis chapters. Section A.2 contains the example application requirements, as in Chapter 1, which are then analysed in Section A.3 to determine the goals and preferences included in the requirements following the method described in Chapter 4. In Section A.4, we decompose the goals into subgoals. Sections A.5, A.6 and A.7 describe the design phase where infrastructure part models are analysed and selected by the designer to create the final form of the design drawing on the techniques discussed in Chapters 5, 6 and 7 respectively.

A.1 Definitions

To clarify the use of language in the example application, we give definitions for commonly used terms.

User By user, we refer to any person using the application. Before implementation, the users are likely to be involved in drafting and refining the application requirements.

Requirements The requirements are an informal description of a desired application. The requirements for the example application are given below.

Application The application is the implemented collective functionality that provides the user with a solution to the application requirements.

Designer The person creating the design from the application requirements and implementing it is called the designer.

System The system is the dynamic, open software environment in which the application will be deployed. It potentially includes many services available for use by applications.

Goals Goals are descriptions of states of the system that the application is intended to bring about.

Goal Instance A goal instance is an explicit statement of a goal instantiated in the system, signifying that the goal should be actively pursued at that moment. A goal instance will be possessed and acted towards by agents in the system.

Goal Triggers Activity in or on the system that causes a goal instance to be created are called goal triggers.

Preferences Preferences are priorities or restrictions on how goals are most suitably achieved according to application requirements.

Opportunism Opportunism is the ability of the agents comprising an application to use the most suitable functionality available within the system at any time.

Originator As described in Chapter 3, the originator of a goal instance is the agent that possesses the goal instance before it is interacted (cooperated) over. The originator will have to initiate the interaction.

Capability Agents that are able to directly act (make changes to the system not involving agents) to achieve a goal are said to have the *capability* to achieve the goal.

Local Actors A local actor is an agent created by the application designer that has the capability to achieve a goal. The existence of local actors in the design, for some goals, is often desirable to ensure that the application can function, even if to a minimal quality, when the system has no third-party services available for a particular goal.

A.2 Requirements

We require a collaborative weather mapping application. Using the application, a global weather map giving the current state is accessed and edited by various collaborating organ-

isations. Contributors can add data they have gathered locally to the map in authorised locations. For example, one contributor organisation may be authorised to add data to a small local area, while another can add to any location within a country. Authorisation is enforced to prevent accidental changes. Contributors and other (paying) organisations can access the weather data and be provided with predictions of weather at specified locations in the future.

For speed and robustness in a system where organisations' software services may become accessible at any time, and later stop being so, the weather data is distributed among the contributors. As different organisations require access to different local areas, the data should be distributed to try to ensure each organisation has as rapid access as possible to the data they need. This distribution should be updated regularly to reflect the current demands.

Several services offer predictions based on the data, and the number of predicters available at any one time may vary, partly due to the load they each have on them. The predicter services vary in speed and in accuracy. They must have access to the weather data to make the prediction, either by moving onto the system on which relevant data is stored or by repeated requests to the relevant sources.

Prediction requests specify location, time (in the future) and whether speed or accuracy should be the priority in producing results. A prediction command, from a user, that prioritises speed of completion is called a *speed view* and one prioritising accuracy of prediction is called an *accuracy view*. On a speed or accuracy view, the application should always report back to the user within 10 seconds either with the prediction or with a 'time out' warning. All currently known predicters offer a time out service whereby the prediction is halted and a warning returned after a specified interval.

As any software service may become inaccessible to the rest of the system at any time, the functionality of the application should be available locally on the computer system of each contributor organisation, and preferably the same software will be loaded at each node for ease of deployment.

A.2.1 Variations

To illustrate the effects of different application priorities, we give two variations on the application requirements. The preferences below are ones which could have been given by

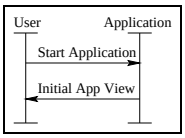


Figure A.1: Event trace for starting the application

the people drafting the requirements in addition to the text above.

Speed Variation After the priorities of the particular operations of the application, e.g. accuracy in the predictions produced from an accuracy view command, speed should be the most important factor in considering how the application is designed and implemented.

Interoperability Variation After the priorities of the particular operations of the application, the ability of the application to interoperate with and take advantage of many services in the open system should be the most important factor in considering how the application is designed and implemented.

A.3 Requirements Analysis

Requirements analysis is used to clarify the functionality required and to clearly express the requirements in terms of the analysis concepts of the methodology, in this case *goals* to be interacted over and *preferences* to be followed when attempting to achieve the goals.

A.3.1 Scenario analysis

Event traces were analysed for the following scenarios (interactions between the user and the local application). Exchange of commands, queries for information and information are shown as arrows from the sending entity (either the user or the application) to the receiving entity (either the user or the application). The scenario starts at the top of each diagram and a series of exchanges (events) occurs as time progresses to the bottom of the diagram.

- Starting the application (Figure A.1)
- Modifying the weather map (Figure A.2)
- Selecting a map location for a specified time to view prediction (Figure A.3)

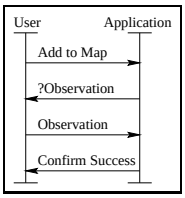


Figure A.2: Event trace for modifying the weather map

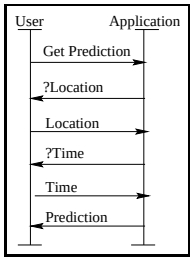


Figure A.3: Event trace for choosing a map location prediction to speed view

- Selecting a map location for viewing but times out after 10 seconds (Figure A.4)
- Changing the access rights of another user (Figure A.5)
- Stopping the application (Figure A.6)

The designer must decide which of these traces represents unique goals to be given to the agents, which are out of the agents' control and which are equivalent to other goals but possibly with different parameters. In the cases above, most events suggest an obvious goal to be achieved. Starting and stopping the local application is for the operating environment to achieve rather than the agents.

The goals identified from scenario analysis are presented in Table A.1.

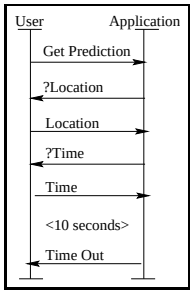


Figure A.4: Event trace for viewing a prediction but operation exceeds 10 seconds

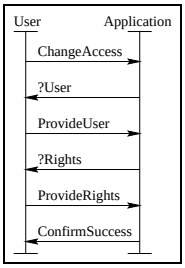


Figure A.5: Event trace for changing the access rights of another user

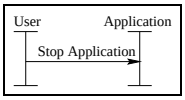


Figure A.6: Event trace for stopping the application

Type	*Name*	*Description*
Goal	Contributed	User contributed data to the weather map
Goal	Speed Viewed	User presented with view of map location prediction of specified time as fast as possible
Goal	Accuracy Viewed	User presented with map location prediction of specified time, as accurate as possible
Goal	Set Access	User has set access rights of another user

Table A.1: Data dictionary after scenario analysis

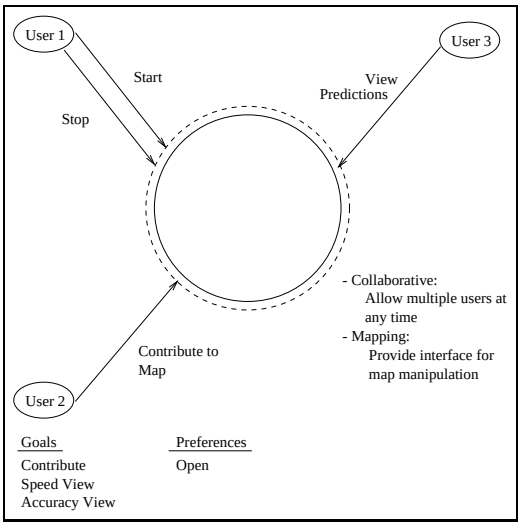


Figure A.7: Entity analysis for a collaborative weather mapping package

A.3.2 Entity Analyses

The following entities were analysed to determine the goals and preferences implicit in their requirements usage, adding to the data dictionary as shown in Table A.2. We analysed the application entities mentioned in the requires to determine the goals and preferences their presence implied. In the analysis diagrams for the entities listed below below, the application is shown as a solid circle, with the entity in question shown as a dashed shape positioned inside or outside the application (to show the relative position implied by the requirements). To determine the implications of the entity's existence, we examine its suggested interactions with other entities, including the user, and the properties it is described to have.

- Collaborative weather mapping package (Figure A.7)
- Weather map (Figure A.8)
- Access Rights (Figure A.9)
- Predicter (Figure A.10)

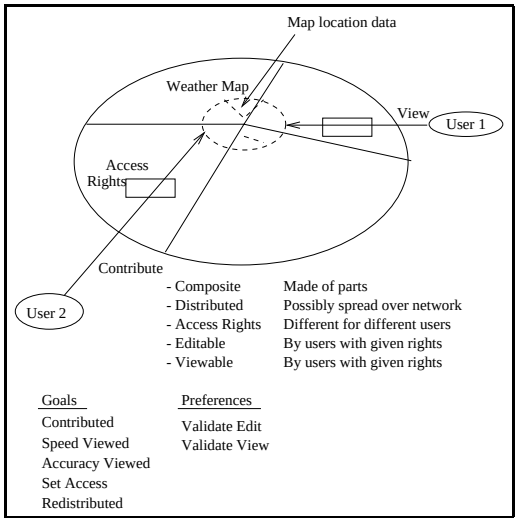


Figure A.8: Entity analysis for weather map

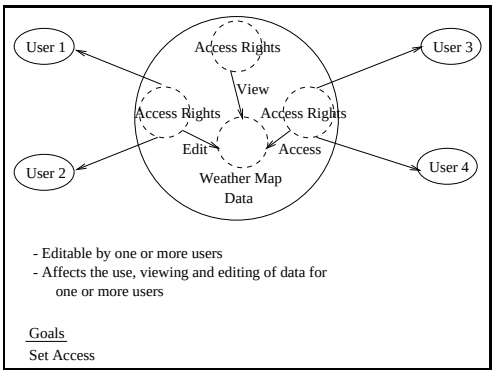


Figure A.9: Entity analysis for access rights

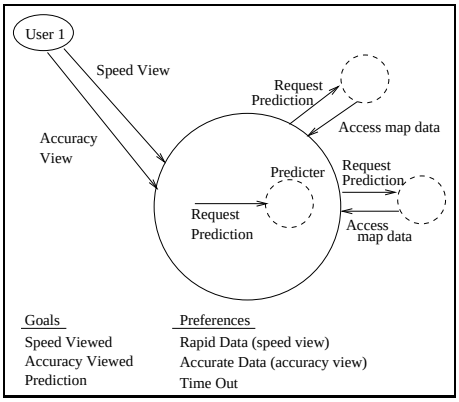


Figure A.10: Entity analysis for predicters

Type	*Name*	*Description*
Goal	Contributed	User contributed data to the weather map
Goal	Speed Viewed	User presented with view of map location prediction of specified time as fast as possible
Goal	Accuracy Viewed	User presented with map location prediction of specified time, as accurate as possible
Goal	Set Access	User has set access rights of another user
Goal	Redistributed	Map data redistributed for high access speed
Preference	Open	No limit on number of users
Preference	Validate Edit	Only authorised users can contribute to map location data
Preference	Validate View	Only authorised users can view predictions on map location data
Preference	Time Out	If prediction takes longer than 10 seconds then stop and warn user

Table A.2: Data dictionary after entity analysis

A.3.3 Goal Analysis

The system goals were analysed to determine the actions that trigger the addition of instances to the application and the preferences associated with each. The analysis adds preferences to the final data dictionary at the end of the requirements analysis stage, as shown in Table A.3. In the analysis diagrams for the goals analysed below, we specify the trigger (as an interaction with other entities) and the end state of the goal divided into those entities affected. When showing information being passed between entities, we use the UML convention of an arrow tailed by an empty circle for the passing of parameters required for goal achievement, e.g. Contribution in Figure A.11, and an arrow tailed by a filled circle for the passing of feedback on a goal's achievement, e.g. the acknowledgement and warning shown in Figure A.11.

- Contributed (Figure A.11)
- Speed Viewed (Figure A.12)
- Accuracy Viewed (Figure A.13)
- Set Access (Figure A.14)
- Redistributed (Figure A.15)

Goals are desired end states so refer to the consequences of an action rather than an action itself, e..g 'User *contributed* data' rather than 'User *contributes* data'. It may seem strange

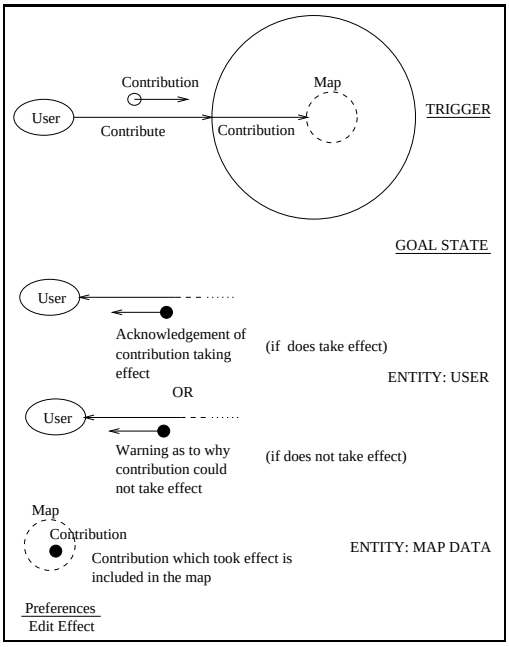


Figure A.11: Contributed goal analysis

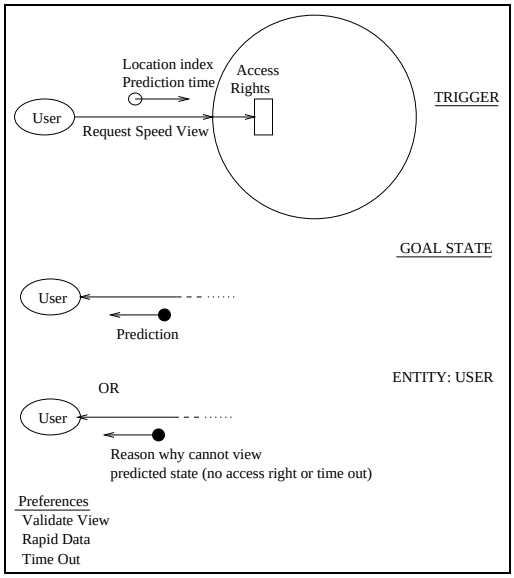


Figure A.12: Speed Viewed goal analysis

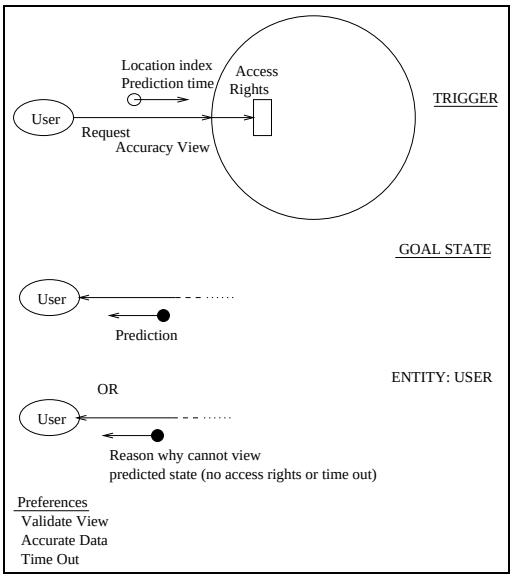


Figure A.13: Accuracy Viewed goal analysis

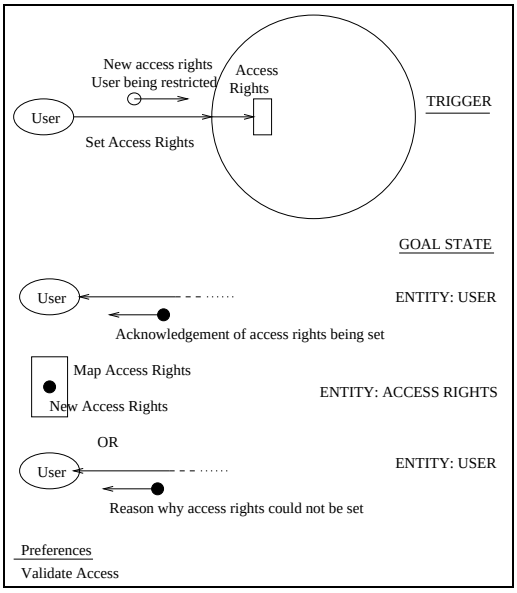


Figure A.14: Set Access goal analysis

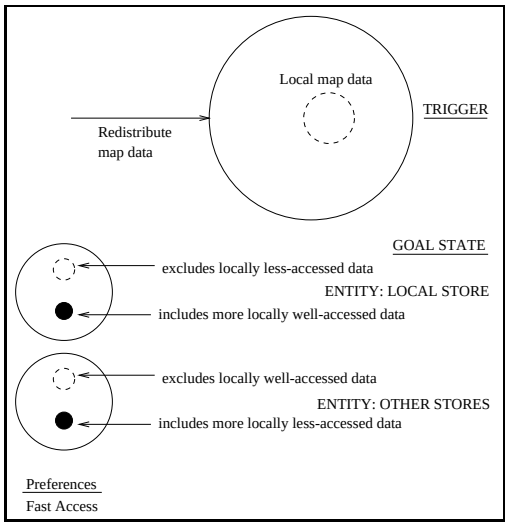


Figure A.15: Redistributed goal analysis

many of the goals refer to the user triggering the goal, when the goal state does not involve the user, e.g. User contributed data. There are two reasons for referring to external entities not obviously within the goal state. First, it is useful for the goal description to refer to all information that the designer knows has to be provided from external sources in order for the goal to be achieved. The user must be known in order for the agents to examine the access rights of the user and decide whether the action is allowed to take effect. Second, the user should be informed of the success or failure of the operation. Therefore the goal 'User contributed data to the weather map' is an abbreviation for 'User is given feedback on goal failure, or access rights of user allows contribution to be integrated into the weather map, contribution is integrated into the weather map and user is given feedback on the goal success'.

Type	*Name*	*Description*
Goal	Contributed	User contributed data to the weather map
Goal	Speed Viewed	User presented with view of map location prediction of specified time (as fast as possible)
Goal	Accuracy Viewed	User presented with map location prediction of specified time (as accurate as possible)
Goal	Set Access	User has set access rights of another user
Goal	Redistributed	Map data redistributed for high access speed
Preference	Open	No limit on number of users
Preference	Validate Edit	Only authorised users can contribute to map location data
Preference	Validate View	Only authorised users can view predictions on map location data
Preference	Time Out	If prediction takes longer than 10 seconds then stop and warn user
Preference	Validate Access	Only authorised users can edit access rights
Preference	Edit Effect	As many contributions as possible take effect
Preference	Rapid Data	Predictions are presented as quickly as possible
Preference	Accurate Data	Predictions are as accurate as possible
Preference	Fast Access	Map data should be distributed to give as rapid access by users as possible
Preference	Opportunism	The application should prioritise taking advantage of the most suitable functionalityavailable in the system.

Table A.3: Data dictionary after goal analysis

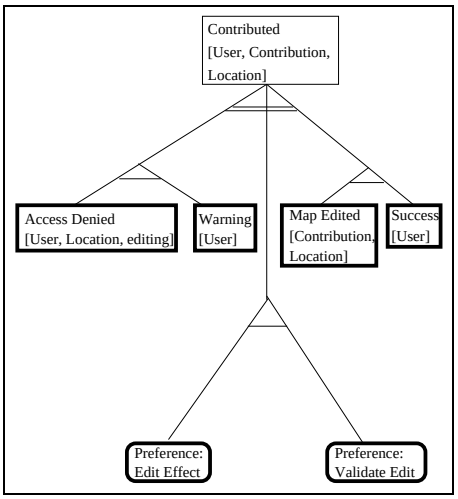


Figure A.16: Decompostion of the Contributed goal

A.4 Goal Decomposition

Goal decomposition is the process of dividing goals into several subgoals each of which can be coordinated over, discussed in Chapter 4. The following goals were decomposed to separate concerns in the application functionality and to enable implementation. In the goal decomposition diagrams, a goal is given by name in a box along with the information (parameters) needed to achieve the goal. The subgoals and preferences of the goals are shown at the end of lines below the goal. Preferences are marked with the word 'Preference'. Where a single horizontal line draws across the set of lines leading to subgoals, all of the subgoals must be achieved for the goal to be achieved. Where a double horizontal line draws across there is a choice between alternative sets of subgoals. For example, in the Contributed decomposition in Figure A.16, the goal may be completed either by checking that access is denied for the user to contribute data *and* a warning is given to the user, *or* the map is edited *and* the success of the goal is reported to the user. Where goals are drawn in a heavy-edged box, there are no further decompositions given for that goal, e.g. Access Denied, Map Edited etc. in Figure A.16.

- Contributed (Figure A.16)
- Speed Viewed (Figure A.17)
- Accuracy Viewed (Figure A.18)

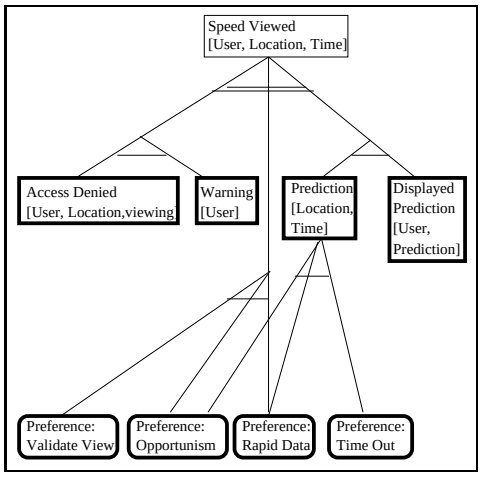


Figure A.17: Decomposition for Speed Viewed goal

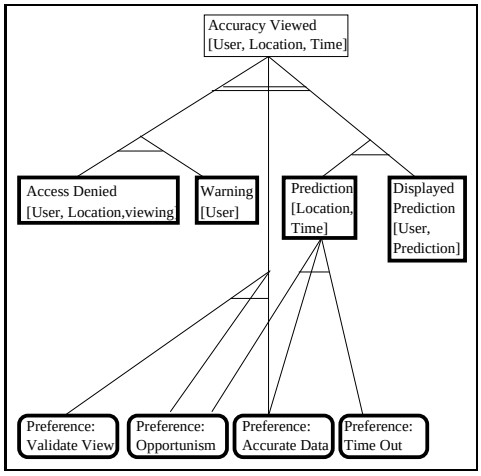


Figure A.18: Decomposition for Accuracy Viewed goal

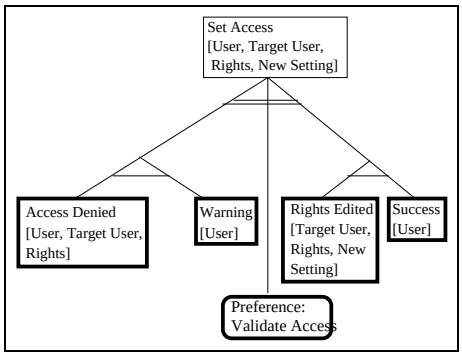


Figure A.19: Decomposition for Set Access goal

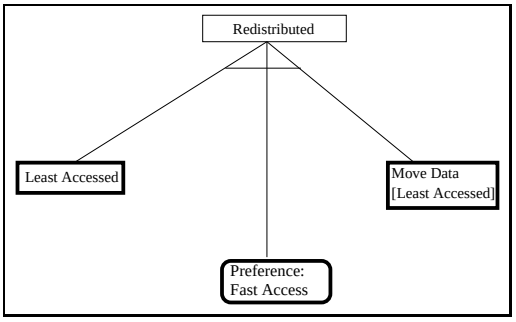


Figure A.20: Decomposition for Redistributed goal

- Set Access (Figure A.19)
- Redistributed (Figure A.20)

After decomposition, we include the resulting subgoals into our set and regard them in the same way as the goals originally derived from the requirements. The full list of goals and preferences at the end of the analysis phase is given in Table A.4.

A.5 Preference Analysis

The infrastructure needs to support the agent operation. In order to identify the components of the infrastructure required for agents to interact over the system goals, the designer can examine various aspects of the application operation. In particular, the designer can refer back to the analyses of goals made earlier to determine the functionality required for goals to be triggered, examine the infrastructure required to allow the agents to interact and also examine existing infrastructures (as discussed in Chapter 5).

Type	*Name*	*Description*
Goal	Contributed	User contributed data to the weather map
Goal	Speed Viewed	User presented with view of map location prediction of specified time (as fast as possible)
Goal	Accuracy Viewed	User presented with map location prediction of specified time (as accurate as possible)
Goal	Set Access	User has set access rights of another user
Goal	Redistributed	Map data redistributed for high access speed
Goal	Access Denied	It has been checked that the user is not authorised to perform the specified action
Goal	Warning	User has been warned of action failure
Goal	Map Edited	Map location data has been changed to a new value
Goal	Success	User has been informed of success of action
Goal	Prediction	Prediction regarding map location at given time has been made
Goal	Displayed Prediction	User views prediction
Goal	Rights Edited	Access rights for user have been changed
Goal	Least Accessed	The least accessed local map data has been identified
Goal	Move Data	Map data stored locally is moved to a remote store
Preference	Open	No limit on number of users
Preference	Validate Edit	Only authorised users can contribute to map location data
Preference	Validate View	Only authorised users can view predictions on map location data
Preference	Time Out	If prediction takes longer than 10 seconds then stop and warn user
Preference	Validate Access	Only authorised users can edit access rights
Preference	Edit Effect	As many contributions as possible take effect
Preference	Rapid Data	Predictions are presented as quickly as possible
Preference	Accurate Data	Predictions are as accurate as possible
Preference	Fast Access	Map data should be distributed to give as rapid access by users as possible
Preference	Opportunism	The application should prioritise taking advantage of the most suitable functionality

Table A.4: Data dictionary after goal decomposition

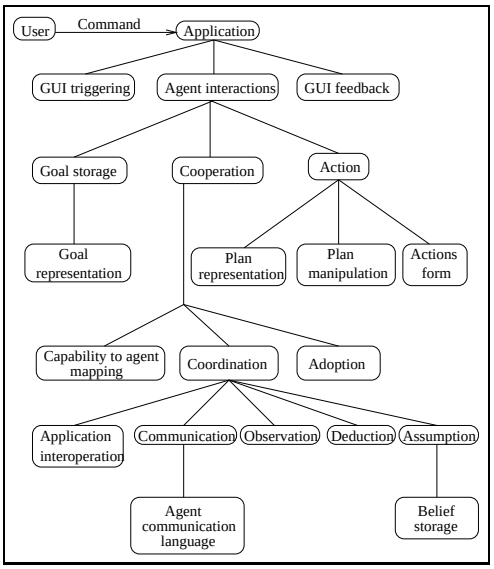


Figure A.21: Infrastructure modularisation showing the support required for goal triggering via the GUI and for agent interactions

Figure A.21 shows a decomposition of a generalised application in which the user issues commands through a graphical user interface (GUI). The functionality of the application is achieved by interacting agents that possess goals. The goals are caused to exist (triggered) by the user interacting with the GUI, and the feedback from any goal achievement or failure) is presented on the GUI. This is shown in the diagram as a decomposition of the application. The agent interactions are further modularised into the possession of a goal by an agent (the *originator* agent for the interaction), cooperation in achieving the goal and action arising from the cooperation. Goals possessed by an agent must be represented in a pre-specified form, as must the plans (plans are decompositions of action in this case). Cooperation is shown to require infrastructure to enable agent capabilities to be identified, agents to be coordinated to best achieve the goal and adoption by agents of the goal over which cooperation takes place (agreement to cooperate). Coordination between applications may require infrastructure to translate between them (as identified by Sycara et al. [94], described in Chapter 4). On an individual agent level, coordination depends on the coordination mechanisms used but belief acquisition by communication, observation, deduction and assumption (as described in Chapter 6) must be supported by the infrastructure. Communication requires a language in which to communicate and assumption may

require some way of storing those assumptions explicitly (as beliefs).

As described in Chapter 5, infrastructure parts can be divided into those that are individual to each agent, *agent infrastructure parts*, and those that are application-wide and support the operation of all agents, *application infrastructure parts*. Agent infrastructure parts are most closely tailored to individual goals, as each part is chosen based on its applicability for a particular goal. Therefore, where it is possible to choose between individual agent infrastructure parts and universally-used application infrastructure parts, the former is preferred due to its high applicability.

Each of the infrastructure parts can be implemented in one or more ways. We describe these possible models using the *infrastructure part model language* (IPML). IPML is a set of properties that can be described for each model, similar to a design pattern langauge. The properties highlight the significant ways in which the model would be compatible or conflict with application preferences, interactions over goals and choices of models for other infrastructure parts. For brevity and realism, we provide only one option for some infrastructure parts and several choices for others.

A.5.1 Parts of the Application

The following high-level parts were identified as fundamental to the functioning application. We refer to the graphical user interface below as the GUI.

GUI Triggering When users interact with the interface, by clicking on buttons for example, they issue commands to the agents making up the application. The mechanism by which the interface and agents interact is an infrastructure part. Models that the designer could choose from for this infrastructure part are shown in Tables A.5 and A.6.

Agent Interactions In agent interaction analysis, goals are described as being achieved by groups of cooperating agents. This requires infrastructure to enable the agents to interact. Agent interactions are abstractions and, therefore, are not implemented as an infrastructure part themselves. Instead we decompose the infrastructure parts involved in interaction below.

GUI Feedback The user will receive interface feedback on a GUI triggered goal, such as the progress, success or failure of the goal. The suggested model that the designer has

Part Name	GUI Triggering
Model Name	GUI Trigger Agent
Description	A single agent accepts all goals triggered from the local GUI and discovers other agents tocooperate with over each.
Algorithms	For each goal triggered from the GUI, the agent takes the following steps:1. Adopt the goal benevolently (see Benevolent model for Adoption part).2. As the originator for the goal, coordinate with other agents to achieve it.
Priorities	The model has the potential to be flexible through being agent-based andallows for coordination to be tailored to the goal from the time it is triggered.
Resource Use	There must be a reference to the GUI trigger agent from the GUI to allowgoals to be communicated to the agent.
Support Required	A benevolent goal adoption mechanism is required in the agent.
Scaling	With a large number of goals passing from the GUI, the trigger agents maybecome a bottle-neck.
Applicability	Applicable where the goals from the GUI are to remain low in frequency, similarin preferences but with complex activities required to achieve each.
Problems	The agent may not be able to be tailored to all goals triggered from the GUI and maybecome a bottleneck with increased frequency.

Table A.5: An IP model for the adoption of goals of a specific type.

available for this infrastructure part is shown in Table A.7.

Part Name	GUI Triggering
Model Name	Designated Local Adopters
Description	When goals are triggered, the local infrastructure is queried to discoverwhich agent is designated to deal with the goal. An agent is chosenfor each possible goal triggered.
Algorithms	On triggering a goal, the GUI does the following.1. Query the local infrastructure to determine the agent designatedto address this goal; 2. Offer the goal to the agent;3. The agent must accept the goal and cooperate to achieve it.
Priorities	The model prioritises division of goal execution between agents and a centralised,reliable point at which to query the agents currently available.
Resource Use	Requires a local store that the GUI can query for references to the designated agents.
Support Required	Designated agents must be compelled to adopt the designated goals on demand (seethe Benevolence model for Adoption and the Forced Cooperation model forCoordination).
Scaling	The number of agents may increase as the number of triggered goals does. The effortrequired to extend the application by adding GUI triggered goals is therefore higherthan with some other models.
Applicability	Most applicable where the frequency of triggered goals is high and diverse, the localresources are large and the goals can be communicated quickly so that the queriesfor local adopters do not overwhelm the local infrastructure.
Problems	As the local infrastructure is not agent-based itself, extra effort is required to maintainit and the triggering may be less flexible in this regard.

Table A.6: An IP model for the adoption of goals of a specific type.

Part Name	GUI Feedback
Model Name	Local GUI Access
Description	Agents can directly access and send commands to the GUI of the local infrastructure.
Algorithms	When an agent needs to communicate with the user it must send acommand message to the local infrastructure.
Priorities	The model prioritises simplicity and speed.
Resource Use	The local infrastructure must have a standard interface for commoncommands to the GUI. Agents must possess knowledge of the commandsinterface. As several commands may be sent to the local infrastructureat once, a queue is used to process the commands.
Support Required	No other infrastructure parts are affected as this is purely an interfacebetween agents wishing to provide feedback and the user.
Scaling	The model allows communication between agents and the interface tobe as direct as possible so processing is minimised. However, if thenumber of commands becomes too large, the local infrastructure couldbe overwhelmed.
Applicability	As this infrastructure part supports the agents in the trivial task ofaddressing the user, this model will be applicable in most situations.
Problems	As the interface is not agent based, it is more difficult to extend.

Table A.7: An IP model for the adoption of goals of a specific type.

A.5.2 Parts of Agent Interaction

Goal Storage Goals can potentially appear and disappear from the application, and are possessed by agents. To explicitly represent this varying number of goals, the agents need suitable storage mechanisms. The suggested model that the designer has available for this infrastructure part is shown in Table A.8.

Goal Representation In order to communicate goals, the format of the goals must be decided. This is related to the properties of goals identified by Logan [68] and discussed in Chapter 2. The suggested model that the designer has available for this infrastructure part is shown in Table A.9.

Cooperation Agents cooperate to achieve goals. As with agent interactions, this is a fairly abstract idea that we believe can be adequately captured by the processes involved in cooperation without the need for a separate infrastructure part describing it. Agents wishing to cooperate need to be able to find capable agents, persuade them to cooperate and coordinate activity with them for best effect. This decomposition is described below.

Part Name	Goal Storage
Model Name	Queue
Description	Store the goals possessed by an agent in a standard queue structure. Inconstrast to a set, this allows the same goal to be possessed in morethan one instance by an agent. The least recently adopted goal is themost recently considered.
Algorithms	Single operations allow the agent to add adopted goals to the tail ofthe queue and remove the goal from the head of the queue.
Priorities	Allows multiple instances of the same goal, prioritises speed andsimplicity over attempting more important goals first.
Resource Use	A standard queue data structure is required within the agent usingthis mechanism.
Support Required	Goal representations must be able to fit within a queue.
Scaling	If multiple instances of the same goal should be ignored, this modelis wasteful at large scales, otherwise it is as succinct as any other model.
Applicability	The model is applicable where multiple instances of the same goal aretreated as separate commands.
Problems	If the goal state is reached for a single goal, it is possibly wasteful toattempt it again if the context has not changed significantly.

Table A.8: An IP model for the storage of goals.

Part Name	Goal Representation
Model Name	Predicate Logic
Description	Uses predicate logic to represent a desired state of the environment stored inside agents.
Algorithms	Logic parsing, unification and matching algorithms are standard.
Priorities	This representation is well established, clear and has known manipulation algorithms.
Resource Use	The goal store of an agent must be able to hold multiple predicate logic statements.
Support Required	Agents must be able to match the predicate logic goals to the observed environmental state.
Scaling	Succinct descriptions by predicate logic allow for good scaling.
Applicability	In most agent-based applications, predicate logic is an adequate and known standard.
Problems	The form is not as easily read or written by people as other structures.

Table A.9: An IP model for the representation of goals.

Part Name	Actions
Model Name	Local Access Acting
Description	Agents act by sending commands to the local infrastructure.
Algorithms	When an agent acts it sends a command directly to the local infrastructurewhich instantiates the action.
Priorities	The model prioritises simplicity and speed through directness of communication.
Resource Use	The local infrastructure must be able to interpret and act on several action commands.As several commands may be sent at once, a queue is used to store the commandsbefore processing.
Support Required	No other infrastucture parts are affected as this is purely an interface between agentswishing to act and the other system state.
Scaling	The model allows communication between agents and the interface tobe as direct as possible so processing is minimised. However, if thenumber of commands becomes too large, the local infrastructure couldbe overwhelmed.
Applicability	As this infrastructure part directly supports the agents in performing actions on thesystem state, it will be applicable in most situations.
Problems	As the interface is not agent-based, it may be more difficult to extend than othermodels.

Table A.10: An IP model for the processing of actions.

Actions The application environment determines how the agents act, but the form of the interface for agents to make changes to the environment may be determined by the application designer. The suggested model that the designer has available for this infrastructure part is shown in Table A.10.

Part Name	Capability to Agent Mapping
Model Name	None / Broadcast
Description	No idenitification of agents by capability is made. Requests of cooperation are either directed at knownindividuals or broadcast to all agents in the open system, as dictated by the coordination mechanisms.
Algorithms	No algorithms are required.
Priorities	This model prioritises lower use of information storage, less reliance on a particular representationof capabilities and less reliance on agents registering their capabilities.
Resource Use	No resources used.
Support Required	The model relies on the communication mechanisms to provide discovery of an adequate range ofagents so that the required capabilities can be found from among them. The communication mechanismsmust allow broadcast of requests.
Scaling	The communication required for broadcast increases as the number of agents in the system increases.
Applicability	Applicable where the number of agents in the system is likely to remain low, where storage space islow and where capability registration is difficult.
Problems	The amount of communication is high as agents are contacteed regardless of capability.

Table A.11: An IP model for mapping capablities required to agents possessing them.

A.5.3 Parts of Cooperation

Capability to Agent Mapping This infrastructure part allows an agent requiring a particular capability (ability to achieve a goal) to discover an agent that possesses that capability. Models that the designer could choose from for this infrastructure part are shown in Tables A.11 and A.12.

Coordination An agent uses a coordination mechanism to ensure its actions and those of other agents are combined for greatest benefit. Models that the designer could choose from for this infrastructure part are shown in Tables A.13, A.14, A.15, A.16, A.17 and A.18.

Adoption An agent can be benevolent and accept all goals offered to it, accept them only from certain other agents, accept only a limited number at a time or only accept goals along with some reward offered for doing so. The way in which agents act when offered a goal to achieve is the adoption mechanism of the agent. Models that the designer could choose from for this infrastructure part are shown in Tables A.19, A.20 and A.21.

Part Name	Capability to Agent Mapping
Model Name	Broker Agent
Description	Agents pass on requests for goals to be achieved through a broker agents in this model. Agentsregister their capabilities with the broker agent. The broker agent is an identified model for thecoordination infrastructure part.
Algorithms	See the IPML description of the Broker Agent model for the Coordination infrastructure part.
Priorities	The model prioritises extensibility through being agent-based, centralisation of capabilitymapping and speed by use of a relatively simple mechanism.
Resource Use	The broker agent requires all the infrastructure needed in order for agents to register capabilitiesand make requests.
Support Required	Agents must use a boker agent as a coordination mechanism using this approach.
Scaling	The broker agent may become a bottleneck if the frequency and size of capability requestsbecomes large.
Applicability	Applicable where service requests are small or infrequent, the number of capable agents is largeand where it is easier for external agents to register their services with an agent than bediscovered.
Problems	The broker agent adds complexity to the design and the requirement that it be used potentiallyreduces the design's match to the preferences by restricting the choice of coordinationmechanisms.

Table A.12: An IP model for the adoption of goals of a specific type.

Part Name	Coordination
Model Name	Forced Cooperation
Description	In this model, certain agents are required to cooperate over a goalon demand and are known to the agent employing this mechanism.The mechanism is approximately the same as message passing in (concurrent) objects.
Algorithms	To request cooperation from a forced agent, there is only one step.1. Demand cooperation over the goal
Priorities	The model prioritises speed, reliability that the cooperators will becapable (through explicit design) and security in knowing thecooperator is pre-determined to be trustworthy.
Resource Use	Local information regarding the forced cooperation in agents isrequired by the agent employing this mechanism.
Support Required	The model requires mandatory adoption of goals in some agentsproviding the capability for this goal.
Scaling	Scales easily as it requires no communication beyond the minimumdemand for cooperation, but does require suitable functionality tocontinue to be available within the application over time.
Applicability	This model is most applicable where security or reliability are ofmuch greater importance than opportunism and where it is knownthat the forced functionality will always be available within theapplication.
Problems	Forced cooperation allows no flexibility in choosing cooperatorsso provides for no opportunism in exploiting the open system.

Table A.13: An IP model for a coordination mechanism.

Part Name	Coordination
Model Name	Commitments
Description	Agents using this coordination mechanism send requests for goal achievement and request offers, accepting the first offer that is-received in reply that is acceptable by their adoption mecha-nism. An agentoffering to achieve the goal and subsequently being accepted is committing iscommitting themselves to the goal's achievement.
Algorithms	1. Send out requests to viable agents (see Capability to Agent infrastructure part);2. Wait for offers; 3. Pass all offers to the adoption mechanism to consider;4. Send acceptance of first vi-able offer.
Priorities	The model minimises processing and so is fast but still able to take opportunityof all services in the system.
Resource Use	The model requires communication of requests and acceptances.
Support Required	The agent communication language needs to be able to express requests, offersand acceptances in a standard way.
Scaling	As the mechanism does not require a large amount of resources or any degreeof centralisation, it should scale well in most cases.
Applicability	The model is applicable where ensuring the speed of coordina-tion is of greater importance than the quality or reliability of solutions. It is also useful where theapplication needs to avoid centralisation.
Problems	As the commitment offers are not assessed to determine their suitability, theagent using the mechanism has no guarantee of achieving the highest qualitysolution, where this is a concern.

Table A.14: An IP model for a coordination mechanism.

Part Name	Coordination
Model Name	Trust
Description	The agent using this mechanism checks the quality of any solutionprovided by an agent and uses these assessments to decide whichoffers to accept in the future. The checks can be either by observationof the state achieved, if the goal attempts to achieve a particular observablestate, or by an independent production of the same information, if thegoal attempts to derive some information.
Algorithms	1. Send requests to agents; 2. Wait for a suitable duration to receive offers;3. If no offers received, resend requests; 4. If some (one or more) offers arereceived, assess them to determine which comes from the most trustworthyagent; 5. Accept the offer from the most trustworthy agent.
Priorities	A trust-based mechanism prioritises quality of solution and reliability inobtaining a solution.
Resource Use	The agent using the mechanism will need to possess quantitative assessmentsof the trustworthiness of other agents, which rise and fall depending onobserved quality of solution. The agent will make observations or requestextra information for each goal.
Support Required	As the agent using the trust mechanism algorithm must wait a specifiedduration, it requires a scheduling mechanism capable of this.
Scaling	With a large number of possible collaborators for a goal, the number ofmodels possessed by the agent will also be large. The observational checkswill add to the time taken to process each goal by the agent.
Applicability	Where the quality of the goal is able to be checked in some way and isof more importance than speed or the low use of resources.
Problems	A trust-based mechanism may add a significant amount of processing toeach goal the agent seeks cooperators for.

Table A.15: An IP model for a coordination mechanism.

Part Name	Coordination
Model Name	Matchmaker Broker Agent
Description	A matchmaker agent is a brokering solution in which requestingagents are given contact information of the providers. The matchmakermaintains a database of the services agents can perform (goals they canachieve) but uses this solely to identify a suitable provider for eachrequester. Requesters and providers then communicate without anyfurther intervention by the matchmaker, which allows for greaterflexibility.
Algorithms	The algorithm for an agent registering a provided service with thematchmaker requires only one step: 1. Communicate service provisionto matchmaker. An agent using a matchmaker to find a service mustalso get a commitment from the provider as the matchmaker agentdoes not do this. 1. Request communication channel to provider ofservice from matchmaker; 2. Receive provider channel from matchmaker;3. Communicate goal commitment from provider; 4. Receive commitmentto achieve goal.
Priorities	The model allows agents to find service providers, gives flexibility incommunication between provider and requester and has a relatively simpleimplementation.
Resource Use	The mechanism requires a data store recording the services that eachagent provides.
Support Required	The agent communication language used must support registering andrequesting services. Agents also need a reference to a matchmaker or to be ableto get one.
Scaling	Little communication is needed for making a request and, by transferringcommunication responsibility to the requester, the matchmaker has no furtherwork after matching requester with provider. However, the recorded servicesand the search of the data increases as the number of agents registering servicesincreases. If the adverts for services are to be large in size then the databasewill quickly expand.
Applicability	The mechanism is applicable where agents will and can easily register servicesto brokers accessible by those others that will need the services. The model allowsan agent-based model for the Capability to Agent Mapping part providing flexibilityand extensibility. It is also applicable where communication between requester andprovider must be flexible, complex or private.
Problems	The data store may become large with many agents or services. Agent identities mustbe revealed in order to provide services, so privacy and looseness of coupling arereduced. A requester must ask both the matchmaker and the provider for a singlegoal.

Table A.16: An IP model for a coordination mechanism.

Part Name	Coordination
Model Name	Intentions
Description	When an agent needs to find a suitable other to cooperate with over a goal,it may be that it would be best for the agent to attempt the goal itself. Thismechanism causes the agent to always choose this option for a goal.
Algorithms	On becoming originator for a goal, the agent adopts the goal itself.
Priorities	The mechanism prioritises speed in decision making and reliability incooperation over flexibility and opportunism in using external capabilitiesthat may provide a higher quality service.
Resource Use	No resources are required in addition to those necessary to achieve thegoal.
Support Required	The adoption mechanism must allow several goals to be adopted benevolentlyas there are no other agents to take them on.
Scaling	If the number of goals becomes large, the agent could become overloaded asit must adopt them all itself.
Applicability	The model is applicable where security, speed and reliability are of so muchimportance that the goal should be achieved by the same agent requiringassurance that it will be done.
Problems	The mechanism does not take opportunity of the potentially higher qualityservices available in the system.

Table A.17: An IP model for a coordination mechanism.

Part Name	Coordination
Model Name	Negotiation
Description	Broadly, we use negotiation to mean repeated communication between agents over one or more values ending in an agreement to action by the agents.
Algorithms	The value(s) negotiated over represents the quality of the goal solution and its meaning will, therefore, depend on the preferences. For this reason, we cannot provide a generic negotitation mechanism. In general terms, each communication will be a request for suggestions, a suggested allocation of values between agents or an acceptance.
Priorities	The model prioritises reaching a high quality solution through communication.
Resource Use	Agents engaging in negotiation must have a means of judging the next step of a negotiation given the current position.
Support Required	The agent communication language must be sophisticated enough to allow the expression of values being attached to domain concepts in order for those values to then be negotiated over.
Scaling	Negotiations need not involve many agents but can involve many communications between agents. Therefore, negotiation may not be suitable where this is a lot of communication occurring in the system for too little benefit.
Applicability	Negotiation is primarily applicable where there is something of value to be exchanged or distributed.
Problems	Negotiation can involve much complexity to work well.

Table A.18: An IP model for a coordination mechanism.

Part Name	Adoption
Model Name	Benevolence
Description	A benevolent agent agrees to cooperate over any request for a specific goal.
Algorithms	On receiving a request for cooperation over the goal, the agent responds with an offer to the sender
Priorities	The model priorities ease of finding agents to take on the goal when it is required,and also ease of implementation.
Resource Use	The model requires goals to be stored within and added to an agent.
Support Required	As benevolence could lead to a goal being adopted before another is achieved,the goal storage part of a benevolent agent must allow for multiple goals.
Scaling	If the agent always accepts new goals, it may not achieve the existing ones quickly.The application may not be most efficient with arbitrary acceptance of goals.
Applicability	Useful for goals that can be achieved relatively quickly but may still overlap in their existencein the application, and for goals for which it is necessary to accept them quickly.
Problems	An agent taking on goals benevolently may become a bottle-neck in the application.

Table A.19: An IP model for the adoption of goals.

Part Name	Adoption
Model Name	1-Goal Benevolence
Description	An agent using this mechanism will adopt only one goal(of a specific type) at a time but indiscriminately.
Algorithms	On receiving a request for cooperation over the goal, theagent will offer to accept it if it has no other goals or rejectit otherwise.
Priorities	The model prioritises wide distribution of goals, preventingagents from becoming overloaded.
Resource Use	The agent must be able to accept and store one goal at atime.
Support Required	The mechanism requires that agents using it can store a singlegoal.
Scaling	If the number of agents is large compared to the number ofgoals, then the application will continue to scale up well. Ifthe number of goals is larger than the number of agents thensome goals may not be accepted.
Applicability	The model is useful where a single goal takes a significantamount of time to complete and there are a large number ofagents able to take on the goal. It is also applicable wherewide distribution of goals is useful in itself.
Problems	If the number of goals existing at a time exceeds the numberof agents then the originators will not be able to find anyagent able to cooperate, leading to delays.

Table A.20: An IP model for the adoption of goals of a specific type.

Part Name	Adoption
Model Name	Payment
Description	Agents are offered some quantitative reward when requested to adopt a goal.Within an open system, an agent will need to confirm the validity of thepayment offer with a trusted source.
Algorithms	1. Receive goal request with payment offer; 2. Check validity of payment withtrusted banker; 3. Offer or refuse adoption of goal
Priorities	This model allows the appropriate matching of goal adoption to a quantitysuch as desired capital (if the reward represents real-life payment for services)or resource use (if the reward represents available resources, allowing for loadbalancing).
Resource Use	The agent should have some built-in criteria in order to assess whether to accepta payment or not (such as a valuation of goals offered).
Support Required	The model requires an infrastructure part offering banking facilities to check thevalidity of payments. It also requires the form of communicated goals to allow for payment offers to be included.
Scaling	This model may aid scaling if the reward is used to represent available resourcesand higher payments can only be offered by agents with access to more resources.
Applicability	Payments between agents are applicable where payments between humans mustbe represented, as well as where other quantities should be balanced.
Problems	The model requires a substantial amount of extra support in banking facilities,payment offers calculated and attached to goals and assessment of offers.

Table A.21: An IP model for the adoption of goals of a specific type.

Part Name	Plan Representation
Model Name	Tree
Description	A plan is stored within an agent as a tree of goals with subgoals.The tree nodes include information on whether their childrenare meant to be taken as disjunctives or conjunctives.
Algorithms	The tree is loaded in from file when the agent is initialised.
Priorities	The model prioritises data representation that closely matchesthe goal decompositions. This allows for simple translationbetween the design and the implementation.
Resource Use	A tree data structure within each plan-manipluating agent isrequired.
Support Required	The agent must have suitable plan manipulation algorithms.
Scaling	As the model closely represents the goal decomposition, itshould not produce significant problems at large scales.
Applicability	The tree representation would be applicable under mostconditions.
Problems	The model does not allow the preferences of goals to beexplicitly represented, which could be useful in someapplications.

Table A.22: An IP model for the representation of plans.

A.5.4 Parts of Action

Plan Representation An agent interacting over a goal may explicitly decompose the goal into several parts for execution. The way in which this is done may be tailored to the goal. The suggested model that the designer has available for this infrastructure part is shown in Table A.22.

Plan Manipulation When executing a plan, an agent must be able to keep track of their position in it. The suggested model that the designer has available for this infrastructure part is shown in Table A.23.

Actions Form The way in which actions are represented affects how they are contained in plans. The suggested model that the designer has available for this infrastructure part is shown in Table A.24.

Part Name	Plan Manipulation
Model Name	Tree Position
Description	This plan manipulation model is associated with the Treemodel for plan representation. The current state of the planis indicated solely by a position in the tree. An agentprocessing a node of the tree becomes the originator forthe goals of the child nodes and coordinates over them.
Algorithms	On adopting a goal, the agent examines the tree containingthe goal and finds cooperators for each of the goal'schildren, or acts directly to achieve the goal if it has nochildren.
Priorities	The model is simple and prioritises opportunism by keepingall subgoals open to cooperation.
Resource Use	A status of the goals and subgoals must be maintained toprevent the agent from repeatedly acting on the same goal.
Support Required	The plan representation must be a tree structure.
Scaling	The model should have no significant impact at small orlarge scale.
Applicability	The model is applicable wherever the tree representation isin use.
Problems	The algorithm may lead to unnecessary interaction betweenagents but this will otherwise have to be determined atdesign-time and so would only be applicable to morestatic environments.

Table A.23: An IP model for the manipulation of plans.

Part Name	Actions Form
Model Name	Predicate Logic
Description	This model uses predicate logic to represent actions by agents.
Algorithms	The algorithms required are those standard for logic parsing, unification and matching.
Priorities	This representation is well established, clear and has known manipulation algorithms.
Resource Use	The interface by which agents perform actions (change the environmental state) must be able toprocess predicate logic actions.
Support Required	The infrastructure part for processing actions must accept predicate logic form actions.
Scaling	Succinct descriptions by predicate logic allow for good scaling.
Applicability	In most agent-based applications, predicate logic is an adequate and known standard.
Problems	Predicate logic descriptions may be complex and difficult to read for some type of actions,e.g. graphical drawing instructions.

Table A.24: An IP model for the form of representation for agent actions.

Part Name	Application Interoperation
Model Name	Uniform Standard
Description	All applications wishing to interact with agents in an application using this modelmust implement communication communication language and protocols.
Algorithms	None required.
Priorities	This model prioritises speed, accuracy of application requests (as no interpretationis required), and ease and accuracy in agents requesting functionality from otherfunctionality. Also, any agent in the open system can, in principle, cooperate directlywith any other using this model.
Resource Use	No internal resources but the standard must be widely available to developers.
Support Required	A broad, general agent communication language is required to allow a range offunctionality to be accessible to all agents.
Scaling	No extra resources will be required as the application interoperation increases, as noadditional processing is necessary.
Applicability	The model is most applicable where speed, ease of interoperation and scaling are ofhighest importance and where the published standard is likely to beimplemented in other systems useful to the application.
Problems	A uniform standard for interoperation requires a standard that is likely to be adaptedand be useful in constructing other applications, and continue to be so in the future.This may be difficult to achieve. If the standard is not widely used, the applicationmay be limited to using only a few applications, which reduces the chances ofopportunistic behaviour by agents.

Table A.25: An IP model for interoperation between applications.

A.5.5 Parts of Coordination

Application Interoperation In order for the application to function fully in an open system, it must have mechanisms to allow it to communicate and interoperate with other applications. This infrastructure part is one of the layers identified by Sycara et al. in [94] and discussed in Chapter 4. Models that the designer could choose from for this infrastructure part are shown in Tables A.25 and A.26.

Communication Modules In order for agents to communicate, the infrastructure needs to provide mechanisms for the transmission of messages between them. The suggested model that the designer has available for this infrastructure part is shown in Table A.27.

Agent Communication Language The agent communication language (ACL) defines the form of the messages passed between agents. Models that the designer could

Part Name	Application Interoperation
Model Name	Translators
Description	Between applications, translating agents attempt to convert fromone communication model to another.
Algorithms	An algorithm is required for each external communication modeland depends on that model. As not all external models may beknown in advance, the translator agents may have to integrateddynamically with newly implemented algorithms.
Priorities	The model prioritises complete flexibility in accessingpotentially all other applications, lessening demand on developercollaboration. It also allows the application not to be tied to asingle communication model itself.
Resource Use	The model requires translation information and processes withintranslator agents to make use of this information.
Support Required	The model is based on extra agents, translators, that can convertbetween communication models.
Scaling	Extra translating agents are required for every new communicationmodel to be connected to the application.
Applicability	The model is applicable where flexibility in interacting with awide range of application communication models is of highestimportance. It is also useful where the functionality of oneapplication is not be easily expressed in terms of the functionalityof a connected application, as the translator may provide additionalcontext-sensitive approximation.
Problems	The model needs to be updated with each communicationmodel, it uses more resources and translation slows down theprocess of interoperation.

Table A.26: An IP model for interoperation between applications.

Part Name	Communication
Model Name	Stubs
Description	Agents communicate with remote others by sending messages toagent *stubs* which then pass on the messages to the agentsthemselves. The stubs, and agents should be able to acceptmessages in the agent communication language chosen,
Algorithms	For agent A to send a message to agent B, A passes the messageto B's local stub and the stub transmits the message to B.
Priorities	The model prioritises flexibility, in that the stub can be alteredto accept other forms of communication. The model also aidsdistribution by separating the points to which agent communicationsare sent. Finally, it simplifies the local operation because the methodof referencing remote agents is the same as for local ones, requiringno additional addressing mechanism.
Resource Use	Each remote agent must have a stub capable of sending onmessages in the agent communication language.
Support Required	Agent communication language messages must have a transferableform.
Scaling	As the number of remote agents accessible to a local node grows,the number of stubs required to represent them grows too.
Applicability	The model is useful where the number of remote agents necessaryto directly communicate from any local node should not grow toolarge and where simplicity and speed in passing message isthe priority.
Problems	The number of stubs required could become unmanageably largeas the number of remote agents grows.

Table A.27: An IP model for communication.

Part Name	Agent Communication Language
Model Name	FIPA-ACL (subset)
Description	The communication language structure and semantics described by the FIPA standards body.It is based on speech acts with a strictly defined syntax.
Algorithms	Algorithms are required for parsing messages in FIPA-ACL and to construct them. These depend on the performatives and so will not be given here for reasons of space.
Priorities	FIPA-ACL is a known standard so prioritises interoperability into the future. It has been tested through use and is widely expressive.
Resource Use	FIPA-ACL requires a semantic language, such as FIPA-SL, to describe content. The content language used can be stated as a property field of FIPA-ACL.Modules to parse and construct messages are also required.
Support Required	Communication modules must be able to transmit FIPA-ACL encoded messages.
Scaling	FIPA-ACL messages can be quite large so the resources required increases morequickly with communication than for smaller languages.
Applicability	Applicable where high interoperability will be required into the future and in domains wherehigh expressivity is necessary.
Problems	Complex message semantics, such as FIPA-ACL's, require complex parsers.

Table A.28: An IP model for an agent communication language.

choose from for this infrastructure part are shown in Tables A.28, A.29 and A.30.

Observation This infrastructure part defines the interface that agents access the environment through. Models that the designer could choose from for this infrastructure part are shown in Tables A.31 and A.32.

Deduction Agents may be able to transform one set of knowledge to derive further knowledge. The suggested model that the designer has available for this infrastructure part is shown in Table A.33.

Assumption Agents may have built-in knowledge represented explicitly or implicitly. This does not really need a choice of models as the knowledge should always be possible to integrate as part of the agent's implementation or as an initial belief.

Belief Storage The knowledge of agents can be stored within the agents in several ways. The suggested model that the designer has available for this infrastructure part is shown in Table A.34.

Part Name	Agent Communication Language
Model Name	KQML
Description	KQML is a commonly implemented agent communication languagerepresenting speech acts by a set of performatives.
Algorithms	Algorithms for parsing KQML statements are not presented herefor reasons of space.
Priorities	The advantage of KQML is that it is implemented in severalexisting systems, allowing the application to potentially interact with them.
Resource Use	KQML uses the KIF language to represent content of messages so parsers are required for KQML and KIF.
Support Required	Communication modules must be able to transmit KQML messages.
Scaling	KQML messages are of a similar size to FIPA-ACL messages withsymbols kept to a minimum but several properties to each message.
Applicability	Applicable where interoperation with existing agent-based applications is particularly useful.
Problems	FIPA-ACL has mostly superceded KQML.

Table A.29: An IP model for an agent communication language.

Part Name	Agent Communication Language
Model Name	Application Specific
Description	An application-specific language can have messages with a succinctstructure only able to express the simple commands relevant tothe application, e.g. OfferPrediction or View(<location>, <time>)
Algorithms	Parsing the messages is mostly a matter of matching the tokensand responding accordingly.
Priorities	Simplicity and speed are priorities for this model.
Resource Use	The agents need modules to parse the messages.
Support Required	Communication modules must be able to transmit messages inthe application-specific language.
Scaling	Application specific ACLs can be as small as possible to fit theapplication requirements.
Applicability	The model is applicable where the application is tightly defined inits functionality and interoperability is not significant.
Problems	Using this model allows no opportunism beyond the applicationsfollowing the standard application-specific language and allowsfor no extension to the application.

Table A.30: An IP model for an agent communication language.

Part Name	Observation
Model Name	Event driven
Description	Using an event model, agents aquire observational information by waitingfor the local infrastructure to inform them of changes in the system state.
Algorithms	The agents receive each event and process it.
Priorities	The event-driven model prioritises speed in agents response to changesin the system state.
Resource Use	The local infrastructure must be able to discover and broadcast changes to the systemstate.
Support Required	A suitable form for representing events must be decided upon, and the agents musthave a standard method of receiving events.
Scaling	With a large number of agents or system changes, the number of events may becomeunmanageably large. Registering the events of interest to each agent (*interestmanagement)* could help reduce the load to each inidividual but adds processingfurther requirements to the local infrastructure.
Applicability	This model is most applicable where state changes are readily available as eventsfor another reason and can be exploited to inform agents. The application shouldalso have a slowly changing state to reduce the amount of events.
Problems	Scaling event-driven observation may be difficult, requiring either filtering or fastprocessing of the events.

Table A.31: An IP model for observation.

Part Name	Observation
Model Name	Polling
Description	Agents interrogate the local infrastructure toobserve system state (or ask other agents todo so).
Algorithms	On requiring a piece of information, the agentsends a request to the local infrastructure.
Priorities	The model prioritises less communication due totargetted information retrieval.
Resource Use	The local infrastructure requires an interface able toquery information.
Support Required	The form of beliefs should allow for queries to be madeconcerning pieces of information.
Scaling	As the only information transmitted will be that needed,the model is very scalable.
Applicability	The model is applicable where the agents drive theneed for information rather than having to react quicklyto changes in the system state.
Problems	The information the agents possess may not representthe most up to date state of the system.

Table A.32: An IP model for observation.

Part Name	Deduction
Model Name	No Additional Deduction.
Description	Using this model, the agent does not transform or combine itscurrent beliefs to form new ones. However, transformationsoccur by other infrastructure parts such as Plan Manipulation.
Algorithms	None.
Priorities	The model prioritises conserving resources by not processingthe beliefs held by an agent.
Resource Use	No resources are used.
Support Required	No support is required.
Scaling	No additional processing means that scaling is ideal.
Applicability	This is applicable in applications where transformations ofbeliefs is and will continue to be meaningless and/or unnecessary.
Problems	If this model is adopted and deduction is later required, it maybe difficult to integrate.

Table A.33: An IP model for deduction.

Part Name	Belief Storage
Model Name	Predicate Logic
Description	Uses predicate logic to represent knowledge about the environment and stored inside agents.
Algorithms	Logic parsing, unification and matching algorithms are standard.
Priorities	This representation is well established, clear and has known manipulation algorithms.
Resource Use	The belief data store must be able to hold multiple predicate logic statements.
Support Required	Agents must be able to match the predicate logic beliefs to the observed environmental state.
Scaling	Succinct descriptions by predicate logic allow for good scaling.
Applicability	In most agent-based applications, predicate logic is an adequate and known standard.
Problems	Some knowledge is better, or only, represented in other forms, e.g. neural networks or higher order logic.

Table A.34: An IP model for the storage of beliefs in an agent.

A.5.6 Other Parts

Organisation The organisation is the set of agents making up the application and the ways in which they are connected. Models for this infrastructure part are derived from the collection of agent infrastructure parts and is discussed further below (Section A.7).

Data Storage for Weather Map The obvious data structure for storing the current weather map is as a mapping from locations to appropriate values. This will be distributed in accordance with the requirements.

A.5.7 Application Infrastructure Part Decisions

For most infrastructure parts, the designer can choose a model by comparing the priorities of the models with the application preferences. For application infrastructure parts, it should also be ensured that all identified goals can be interacted over and achieved. The designer needs to check that the resources used by each model (from the IPML Resource Use property) are available, that the other infrastructure parts required by the model (from the IPML Support Required property) do not conflict with decisions made on those parts and that the model will scale well enough for the application (accordining to the IPML Scaling property). After choosing a model for each of the application infrastructure parts not requiring further analysis, in this case all parts but *coordination, adoption, capability-to-agent mapping* and *organisation*, the models shown in Table A.35 have been chosen by the designer. Justifications are given relating the properties given in the IPML descriptions above to the application requirements in Section A.2, wherever a choice of models exists. Different models may be chosen for the Speed Variation and the Interoperability Variation of the requirements (see Section A.2). The variations are abbreviated to SV and IV, respectively, in Table A.35 for reasons of space.

A.6 Assurance Analysis

Coordination is a complex process and there are several possible models given in the section above. Assurance analysis attempts to match the coordination mechanism appropriate for the originator of a goal to the preferences of the goal. To make the analysis clearer, we briefly summarise the coordination mechanisms that were described in full IPML descrip-

Part	Model	Justification
GUI Triggering	Designated Local Adopters	The GUI triggered goals vary widely in their preferences.
GUI Feedback	Local GUI Access	
Goal Storage	Queue	
Actions	Local Access Acting	
Goal Representation	Predicate Logic	
Plan Representation	Tree	
Plan Manipulation	Tree Position	
Actions Form	Predicate Logic	
Application Interoperation	SV: Uniform Standard, IV: Translators	Where interoperation is paramount, translators are necessary,but otherwise the extra resource requirements means that a uniform standard is more applicable to applications with tightlydefined functionality.
Communication	Stubs	
Observation	Polling	Agents are driven by the need for weather map data rather thanbeing driven by current system state.
Deduction	No Additional Deduction	
Agent Communication Language	FIPA-ACL	Interoperation in the future is important to the opportunism ofthe application and using a subset of FIPA-ACL shouldnot be significantly slower than other models.
Belief Storage	Predicate Logic	

Table A.35: Infrastructure part models chosen for application infrastructure parts.

tions above.

Commitments A fast, simple mechanism where an request is made for collaborators to commit to a goal and the first one(s) to reply are chosen.

Trust A mechanism whereby agents offering to cooperate are assessed on the previous quality of solutions they have produced and the most *trustworthy* chosen.

Broker Agent An agent-based mechanism where a broker agent (facilitator) records registrations of agents offering to attempt a goal. The broker can then be asked to supply cooperator references for the originator to coordinate with.

Forced Cooperation Object-oriented style cooperation where an agent is known to always accept requests for cooperation on command.

Intention A mechanism whereby the originator does not cooperate but achieves the goal itself.

Negotiation A mechanism allowing a group of agents to send a series of exchanges until they have agreed on assigning particular values to objects or tasks which enables the goal to then be best achieved.

The preferences as given in the requirements may be too application-specific to easily be identified as obeyed or otherwise by the coordination mechanisms. Therefore, the designer should examine the requirement preferences to determine the priorities that they require in coordination mechanisms. See Table A.4 for the descriptions of the preferences.

Edit Effect requires *reliability* of each goal being completed successfully.

Validate Edit requires *access to access rights* and also *security* in the acquisition of the rights.

Validate View requires *access to access rights* and also *security* in the acquisition of the rights.

Validate Access requires *access to access rights* and also *security* in the acquisition of the rights.

Rapid Data requires *speed* in obtaining results.

Accurate Data required high *quality* of results (accuracy of predictions).

Fast Access requires high *quality* of results (fast access through suitable distribution).

Opportunism requires high *flexibility* in the choices of agent cooperators.

Time Out requires nothing in terms of coordination as it is monitored by the agent possessing the goal.

Speed Variance requires *speed* in obtaining results.

Flexibility Variance requires high *flexibility* in the choices of agents.

The priorities identified (*reliability, access to access rights, security, speed, quality* and *flexibility*) are all preferences but expressed in a form more suitable for comparison with application-independent coordination mechanism models, as they are more application-independent themselves. The final two preferences in the list above are those application-wide preferences derived from the variations on the requirements given in Section A.2.

First, the influence of the application variances, Speed Variance and Interoperability Variance, were analysed to discover the influence they have on the choice of coordination mechanism. In Figure A.22, we analyse how well each of the coordination mechanisms match the preference to prioritise speed and to prioritise flexibility. A full explanation of the process is given in Chapter 5. In Figure A.22, the analysis is annotated where a decision is made to state that a coordination mechanism is not as suitable for a preference. The annotation references (marked F, G, H and I) refer to notes in Table A.36.

The following goals were analysed using the same method.

- Contributed goal (see Figure A.23)
- Speed Viewed goal (see Figure A.24)
- Accuracy Viewed goal (see Figure A.25)
- Set Access goal (see Figure A.26)
- Redistributed goal (see Figure A.27)
- Prediction goal (see Figure A.28)

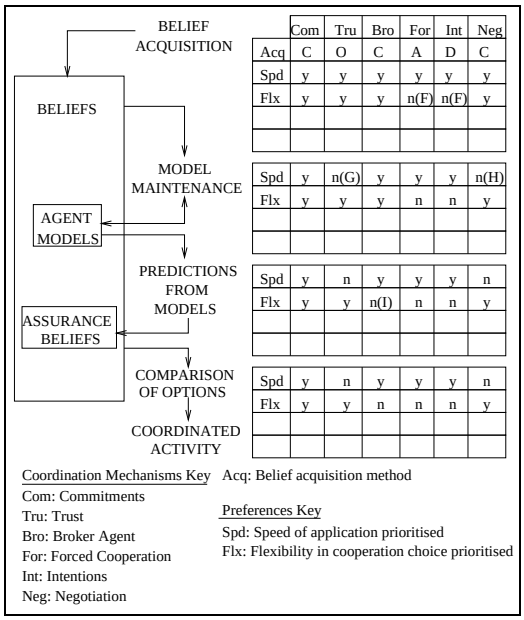


Figure A.22: Assurance analysis for application-wide preferences.

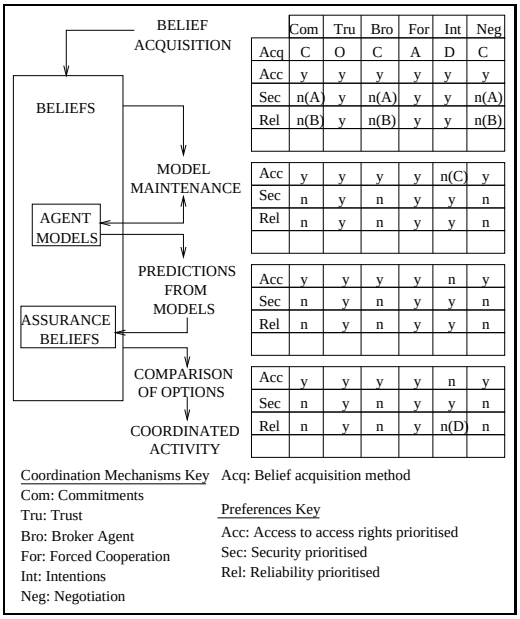
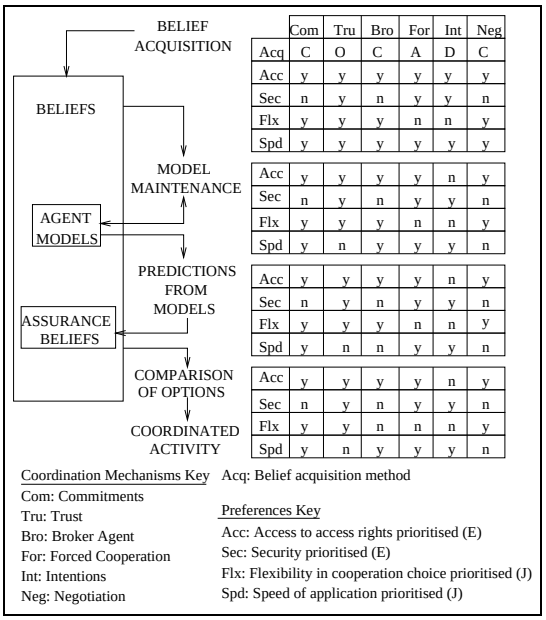


Figure A.23: Assurance analysis of Contributed goal



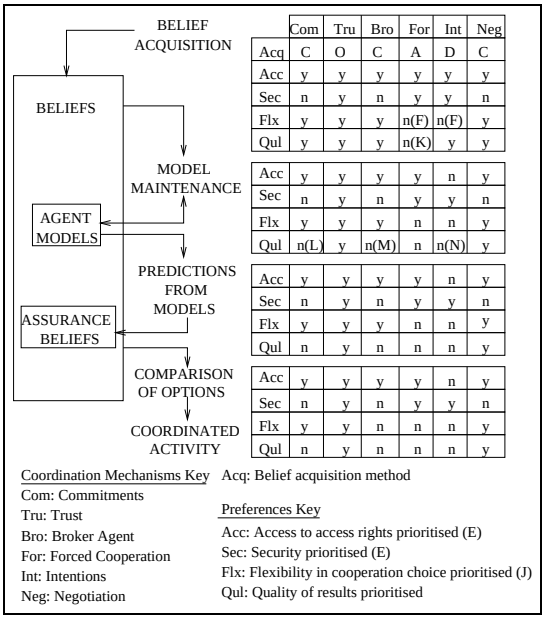
	Com	Tru	Bro	For	Int	Neg
Acq	C	O	C	A	D	C
Acc	y	y	y	y	y	y
Sec	n	y	n	y	y	n
Flx	y	y	y	n	n	y
Spd	y	y	y	y	y	y
Acc	y	y	y	y	n	y
Sec	n	y	n	y	y	n
Flx	y	y	y	n	n	y
Spd	y	n	y	y	y	n
Acc	y	y	y	y	n	y
Sec	n	y	n	y	y	n
Flx	y	y	y	n	n	y
Spd	y	n	n	y	y	n
Acc	y	y	y	y	n	y
Sec	n	y	n	y	y	n
Flx	y	y	n	n	n	y
Spd	y	n	y	y	y	n

Coordination Mechanisms Key
Com: Commitments
Tru: Trust
Bro: Broker Agent
For: Forced Cooperation
Int: Intentions
Neg: Negotiation

Acq: Belief acquisition method

Preferences Key
Acc: Access to access rights prioritised (E)
Sec: Security prioritised (E)
Flx: Flexibility in cooperation choice prioritised (J)
Spd: Speed of application prioritised (J)

Figure A.24: Assurance analysis of Speed Viewed goal.



	Com	Tru	Bro	For	Int	Neg
Acq	C	O	C	A	D	C
Acc	y	y	y	y	y	y
Sec	n	y	n	y	y	n
Flx	y	y	y	n(F)	n(F)	y
Qul	y	y	y	n(K)	y	y
Acc	y	y	y	y	n	y
Sec	n	y	n	y	y	n
Flx	y	y	y	n	n	y
Qul	n(L)	y	n(M)	n	n(N)	y
Acc	y	y	y	y	n	y
Sec	n	y	n	y	y	n
Flx	y	y	y	n	n	y
Qul	n	y	n	n	n	y
Acc	y	y	y	y	n	y
Sec	n	y	n	y	y	n
Flx	y	y	n	n	n	y
Qul	n	y	n	n	n	y

Coordination Mechanisms Key
Com: Commitments
Tru: Trust
Bro: Broker Agent
For: Forced Cooperation
Int: Intentions
Neg: Negotiation

Acq: Belief acquisition method

Preferences Key
Acc: Access to access rights prioritised (E)
Sec: Security prioritised (E)
Flx: Flexibility in cooperation choice prioritised (J)
Qul: Quality of results prioritised

Figure A.25: Assurance analysis of Accuracy Viewed goal.

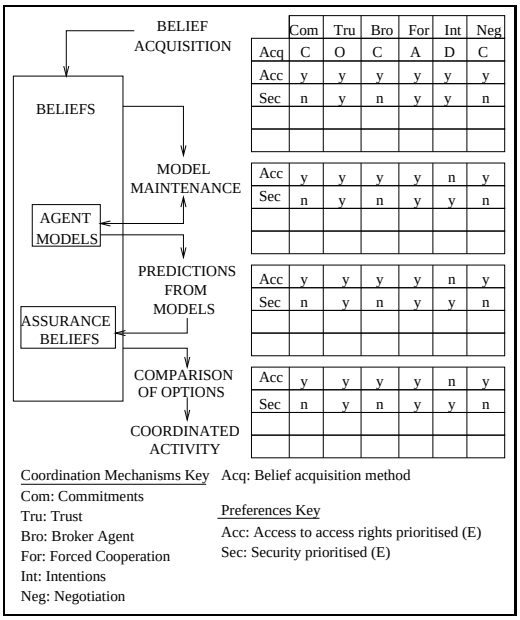


Figure A.26: Assurance Analysis for Set Access goal.

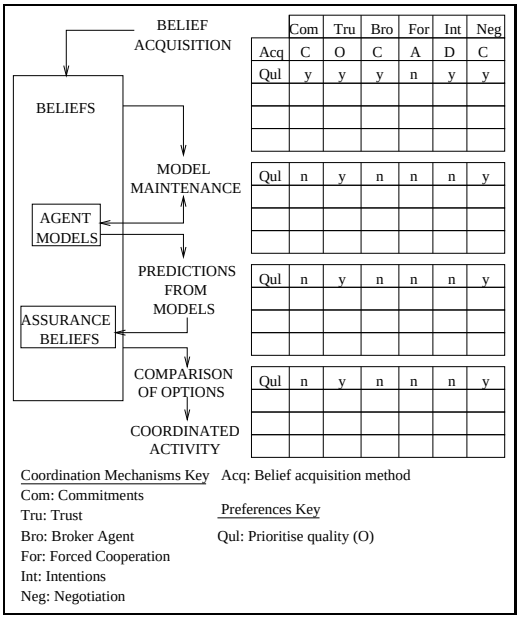


Figure A.27: Assurance Analysis for Redistributed goal.

Code	Note
A	Security in communication-based coordination is worse than for observation, assumption ordeduction as there is another stage (the communicating agent) that has to pass through, beyond the securityprovided by the infrastructure.
B	Communication-based mechanisms rely on accurate information in messages passed to them, which leavesan agent using the mechanism unable to reliably achieve the goal. Assumption is also unreliable in general,but not in the case of forced cooperation, where the reliability is explicitly determined by design.
C	The agent achieving the goal does not necessarily have access to the rights itself.
D	The agent does not reliably have the necessary capability to act on the goal.
E	See the Contributed goal diagram (Figure A.23) for annotated analysis of this preference.
F	Assumption and deduction limit knowledge to products of that already known, so are not as flexible.
G	Building up trust models may take significant amounts of time.
H	Negotiating may be significantly time consuming.
I	Broker agents take on the choice of collaborators and so limit the flexibility of choice in the originator.
J	See the application preferences diagram (Figure A.22) for annotated analysis of this preference.
K	Assumption will not offer the highest quality options if the options in the open system improve beyondthose assumed.
L	Commitments provide no quality information.
M	Brokers provide no quality information.
N	Intention only models the agent itself so it does not address quality in other agents.
O	See the Accuracy Viewed goal diagram (Figure A.25) for annotated analysis of this preference.

Table A.36: Explanatory notes for the assurance analysis diagrams.

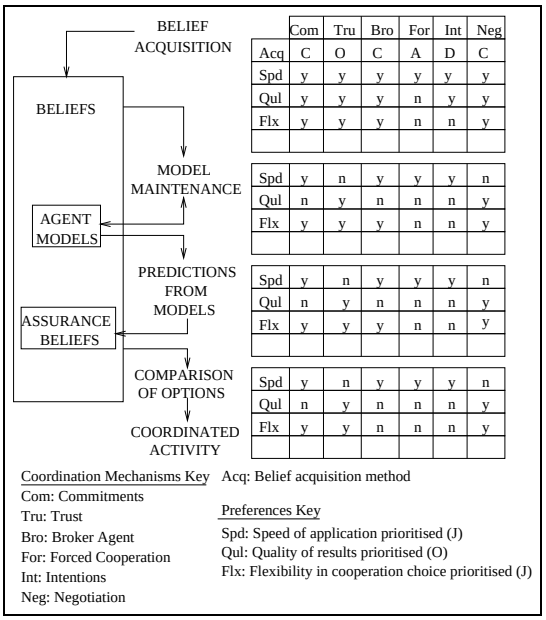


Figure A.28: Assurance Analysis for Prediction goal.

A.6.1 Agent Infrastructure Part Decisions

Each goal instance is the result of interaction between an originator agent and at least one cooperating agent. The originator agent must initiate the coordination of agents in order to best match the preferences of the goal. The choices of mechanism for coordination, adoption and capability to agent mapping (agent infrastructure parts) are given below for the originator of each goal. The choices for coordination mechanism are based on the assurance analyses in Figures A.22 to A.28. In general, the mechanism chosen for a goal is one of those most likely to allow the goal's preferences to be followed, i.e. the one with the most 'y's at the end of the analysis. Further justification of the choice is given where appropriate. For the adoption mechanisms, *benevolence* is chosen unless limiting the agent to one goal at a time is justified, in which case *1-goal benevolence* is chosen and a justification given. For the capability-to-agent mapping mechanisms, *broadcast* is chosen for simplicity unless the particular goal is more easily achieved by agents registering their capabilities, in which case the *broker* model is chosen. See the IPML descriptions in Section A.5 for more details on the priorities of these models.

As well as originators, agents within the application set may also be required to have the ability to achieve goals directly by acting on the system, even if they will not do

so as preferably as other agents within the system. These *local actors* are also described below for all those goals that are not decomposed into subgoals (see Section A.4). There does not have to be local actors for every undecomposed goal in an application design. For instance, it would be reasonable for the example application to rely on external agents to make weather predictions, but in this case the requirements explicitly state that the local application could be able to make, possibly poor, predictions.

Contributed (Originator)

Coordination Speed Variation: Forced cooperation (fast engagement of known co-operators), Interoperation Variation: Trust (does not limit choices)

Adoption Benevolence

Capability to Agent Mapping Broadcast

Speed Viewed (Originator)

Coordination Commitments (speed is the most important preference to this goal, so commitments are preferable to trust or forced cooperation)

Adoption Benevolence

Capability to Agent Mapping Broadcast

Accuracy Viewed (Originator)

Coordination Trust

Adoption Benevolence

Capability to Agent Mapping Broadcast

Set Access (Originator)

Coordination Speed Variation: Forced cooperation (fast engagement of known co-operators), Interoperation Variation: Trust (does not limit choices)

Adoption Benevolence

Capability to Agent Mapping Broadcast

Redistributed (Originator)

Coordination Negotiation (other contributor nodes are trusted and their are valued objects to be negotiated over, so negotiation is preferable to trust)

Adoption 1-Goal Benevolence (only one redistribution of data should occur at a time)

Capability to Agent Mapping Broadcast

Access Denied (Originator) Coordination mechanisms for Access Denied and all other goals with no associated preferences are chosen by examining the application preferences analysed in Figure A.22.

Coordination Commitments (commitments are applicable to both variations and are a simple, fast solution for interacting with other trusted agents)

Adoption Benevolence

Capability to Agent Mapping Broadcast

Access Denied (Local Actor)

Capability Ability to check access rights for authorisation

Warned (Originator)

Coordination Commitments

Adoption Benevolence

Capability to Agent Mapping Broadcast

Warned (Local Actor)

Capability Ability to provide the user with a warning that a triggered goal was unsuccessful.

Map Edited (Originator)

Coordination Commitments

Adoption Benevolence

Capability to Agent Mapping Broadcast

Map Edited (Local Actor)

Capability Ability to edit the current weather map.

Success (Originator)

Coordination Commitments

Adoption Benevolence

Capability to Agent Mapping Broadcast

Success (Local Actor)

Capability Ability to provide the user with an acknowledgement of an operation's success.

Prediction (Originator)

Coordination Commitments

Adoption 1-Goal Benevolence (predictions may take a substantial amount of time so should be forced to be distributed)

Capability to Agent Mapping Broker (predictors will be mostly external so discovery should be encouraged by allowing registration)

Prediction (Local Actor)

Capability Ability to make a weather prediction for a given location and time.

Displayed Prediction (Originator)

Coordination Commitments

Adoption Benevolence

Capability to Agent Mapping Broadcast

Displayed Prediction (Local Actor)

Capability Ability to provide the user with the results of a prediction in a suitable form.

Rights Edited (Originator)

Coordination Commitments

Adoption Benevolence

Capability to Agent Mapping Broadcast

Rights Edited (Local Actor)

Capability Ability to edit the access rights for a user.

Least Accessed (Originator)

Coordination Commitments

Adoption Benevolence

Capability to Agent Mapping Broadcast

Least Accessed (Local Actor)

Capability Ability to determine that section of the weather map data stored locally that is accessed least by local agents.

Moved Data (Originator)

Coordination Commitments

Adoption Benevolence

Capability to Agent Mapping Broadcast

Moved Data (Local Actor)

Capability Ability to move a part of the locally stored weather map data to another point in the system.

A.6.2 Support Required for Chosen Mechanisms

As we have chosen to use a broker agent to find capable agents for the Prediction goal, the broker becomes a third agent in the interaction over that goal and is specified below.

Prediction (Broker)

Coordination Broker (self)

Adoption Benevolence

Capability to Agent Mapping Broker (self)

Negotiation has been chosen for the Redistributed goal and, as the algorithm implementing this coordination mechanism is dependent on the goal (see IPML description in Table A.18), we need to provide a suitable algorithm. The approach chosen in goal decomposition (Figure A.20) is to move the least accessed part of the weather map data at a node to a more suitable node. The preference for the Redistributed goal is to prioritise fast access to data, which means ensuring the data stored on a node is that which is most accessed by that node. The algorithm is as follows.

1. Identify the least accessed parts (locations) of the weather map data (Least Accessed goal).
2. Offer one of the least accessed part to other nodes.
3. Receive numeric bids for the data part. The bids are proportional to the number of accesses made on the data by the bidding node. Bidders may include in their bid for one part, the rejection of another part they previously accepted in this redistribution, which is then returned to those parts to be distributed. This will happen if the latest part is of more use (more accesses) than the previously accepted part.
4. The highest bid is accepted and the part transferred to the node containing the bidding agent.
5. If more parts are to be redistributed, the agent returns to step 2 to negotiate on another.

Depending on the requirements variation, up to three goals may be best served by a trust-based mechanism, where agents are assessed to judge their competence and reliability for the future. The trust mechanism IPML description (Table A.15) notes that a scheduler is required to allow the agent to wait for several offers before one is chosen. A model therefore has to be chosen for the 'scheduler' infrastructure part. For brevity, we will not give a full IPML description but simply state here that the scheduler model available allows the agent to postpone continuing the algorithm for a specified duration, during which the agent can take part in other activities.

A.7 Collation

As described in Chapter 3, we do not assume that the agents taking part in interactions (originators, local actors, brokers etc.) are not actual implemented agents within the application set as the agents actually taking part in the interaction may be external to the application set. Also, the same agent may act as originator for one goal interaction and a cooperator in another interaction. Therefore, originators, brokers and other interaction roles are said to be filled by *place-holders* for agents. We identified 24 place-holders for agents in interaction roles in the section above (originators, local actors and a broker).

When all the agent place-holders have been identified and agent infrastructure parts have been chosen, the designer must determine how they comprise the final organisation of the application set. As described in Chapter 5, the designer examines how well different organisations best fit the requirements by examining three factors. *Low replication* is the priority to reduce the amount of unjustified redundancy in the design by merging agents. *Close approximation* is the priority to keep agents close to the models chosen in the previous section by ensuring very disparate agent models are not merged, as then the infrastructure part models would have to be approximated. *Ease of integration* is the priority to only merge agents that can easily be merged. The third factor, ease of integration, is of more importance to maintenance and extension, so we do not discuss it further here. In the next chapter we will show how it is useful as part of the methodology's approach to maintenance and extension.

For low replication the following organisation models are identified, starting with the lowest replication and becoming less suitable.

1. Merge all the place-holders into one agent.
2. Merge the place-holders into two agents:
   (a) An originator agent acting as the originator for all goals and coordinating accordingly.
   (b) An local actor agent able with the capability of all local actors.
3. Merge the place-holders into ten agents:
   (a) An originator agent for the Contributed and Set Access goals (as they share all mechanisms).
   (b) An originator agent for the Accuracy Viewed goal.
   (c) An originator agent for the Redistributed and Prediction goals (as they share adoption mechanisms).
   (d) An originator agent for all other goals (as they share all mechanisms).
   (e) A local actor agent for the Access Denied and Rights Edited capabilities (as they share resources required).
   (f) A local actor agent for the Warned, Success and Displayed Prediction capabilities (as they share resources required).
   (g) A local actor agent for the Least Accessed and Moved Data capabilities (as they share resources required).
   (h) A local actor agent for the Map Edited capability.
   (i) A local actor agent for the Prediction capability.
   (j) A broker agent for the Prediction goal .
4. Merge the place-holders as above but separate the originator for the Redistributed and Prediction goals.
5. Separate all place-holders into different agents.

The originators and local actors are separated above because they provide different functionality (the former adds coordination, the latter gives capability when it is lacking in the rest of the system). Decisions to merge or separate originators are based on the similarity

of decisions on infrastructure parts given in the previous section. Decisions to merge or separate local actors are based on the similarity of local resources required. A decision to merge an originator with an local actor would only be taken where it was clearly justified to keep the coordination and action on a goal together in one agent, as otherwise we are unnecessarily binding the coordination needed in the future of the application to the possibly temporary capability to act locally. The obvious justification for merging an originator with a local actor would be if the coordination mechanism for an originator was chosen to be to use *intentions*, where the agent acts on the goal itself.

For close approximation the following organisation models are identified, starting with the lowest replication and becoming less suitable.

1. Separate all place-holders into different agents.
2. Merge only the originators for the place-holders using exactly the same coordination, adoption and capability-to-agent mapping mechanisms (results in six originator agents, one broker, nine local actor agents).
3. As above, but also merge local actor agents in the way described by suggestion 3 for low replication above, i.e. Access Denied capability merged with Rights Edited etc.
4. As above, but also merge the Contributed orginator with the Set Access originator (equivalent to suggestion 4 for low replication).
5. Merge all the place-holders into one agent.

The lists of possible models for low replication and close approximation exclude many others that are obviously no better than the ones given. To be more exact, those excluded do not significantly aid in achieving one priority or the other. For example, if the Contributed originator was merged with the Access Denied originator, some approximation would have to be made so that the merged agent had one coordination mechanism, one adoption mechanism and one mapping mechanism, but there would be no reason to not merge that agent with the originators for Warned, Map Edited etc. as nothing further is approximated in merging them and there is lower replication if it is done.

On examining the suggestions above, the designer may choose suggestion 4 in the two lists, as this ensures no approximation of mechanisms is required but the place-holders are merged where possible otherwise. This results in an multi-agent system with eleven agents

	Coordination	Adoption	C-A Mapping	Capability	GUI Triggering	Will Adopt
1	Forced Cooperation	Benevolence	Broadcast	None	Contributed, Set Access	Contributed, Set Access
2	Trust	Benevolence	Broadcast	None	Accuracy Viewed	All goals
3	Commitments	Benevolence	Broadcast	None	All other triggered goals	All goals
4	Negotiation	1-Goal Benevolence	Broadcast	None	None	Redistributed
5	Commitments	1-Goal Benevolence	Broker (11)	None	None	Prediction
6	Not originator	Benevolence	Broadcast	Warned, Success, Displayed Prediction	None	Warned, Success, Displayed Prediction
7	Not originator	Benevolence	Broadcast	Map Edited	None	Map Edited
8	Not originator	Benevolence	Broadcast	Prediction	None	Prediction
9	Not originator	Benevolence	Broadcast	Access Denied,	None	Access Denied, Rights Edited
10	Not originator	Benevolence	Broadcast	Least Accessed,	None	Least Accessed, Moved Data
11	Broker (self)	Benevolence	Broker (self)	None	None	No goals

Table A.37: Agents produced by collation with cardinality (speed variation)

(five originators, one broker and five local actors) shown in Table A.37. In the table we include the which agent will act as a designated local adopter for the GUI to pass triggered goal instances to (this requires a benevolent adoption mechanism). These are chosen based on the suitability of the agent (how close it approximates the infrastructure parts needed). We also include which goals the agent will adopt if offered. For flexibility, this is kept as wide as possible but may be limited by the appropriateness of the coordination mechanism or capability-to-agent mapping mechanism, e.g. forced cooperation requires agents that will be forced to adopt the goal.

A.7.1 Agent Types

If the application is likely to process several goal instances at once, particularly if they are of one type of goal, the designer may decide to use the results of the collation as a set of *agent types*. The designer would then build useful redundancy and replication into the application by instantiating several agents from each type, e.g. eight Redistribution originators in the initial design.

If this option was taken with the example application, the designer may reason as follows.

- The local user is likely to trigger goals at a reasonably slow pace, so there is only need for one agent of each of the Contributed/Set Access and Accuracy Viewed originator agent types.
- Only one redistribution of local data should be occurring at a time to prevent conflicts, so there only needs to be one Redistributed originator, and one local actor for Least Accessed / Moved Data.
- The Prediction orginator and the Prediction, Map Edited and Access Denied/Rights Edited local actors may be used by external sources so the number should be chosen according to the expected number of collaborating weather institutions, to prevent overloading a single agent.
- There should only be one broker for the Prediction goal, as otherwise predictors may not all be registered in one place.
- The originator for other goals may be used several times during the attempt of one triggered goal instance. For example, when Set Access is triggered, the same agent type acts as originator for Access Denied, Warning, Rights Edited and Success. It may be best to have three or four of this agent type to prevent overloading and distribute goals.
- The local actor for Warning, Success and Displayed Prediction is only used once per local triggered goal and displays information to the local GUI, so only one agent is needed.

We add numbers of each agent type in Table A.38.

	Coordination	Adoption	C-A Mapping	Capability	GUI Triggering	Will Adopt	No.
1	Forced Cooperation	Benevolence	Broadcast	None	Contributed, Set Access	Contributed, Set Access	1
2	Trust	Benevolence	Broadcast	None	Accuracy Viewed	All goals	1
3	Commitments	Benevolence	Broadcast	None	All other triggered goals	All goals	4
4	Negotiation	1-Goal Benevolence	Broadcast	None	None	Redistributed	1
5	Commitments	1-Goal Benevolence	Broker (11)	None	None	Prediction	10
6	Not originator	Benevolence	Broadcast	Warned, Success, Displayed Prediction	None	Warned, Success, Displayed Prediction	1
7	Not originator	Benevolence	Broadcast	Map Edited	None	Map Edited	10
8	Not originator	Benevolence	Broadcast	Prediction	None	Prediction	10
9	Not originator	Benevolence	Broadcast	Access Denied,	None	Access Denied, Rights Edited	10
10	Not originator	Benevolence	Broadcast	Least Accessed,	None	Least Accessed, Moved Data	1
11	Broker (self)	Benevolence	Broker (self)	None	None	No goals	1

Table A.38: Agents produced by collation (speed variation)

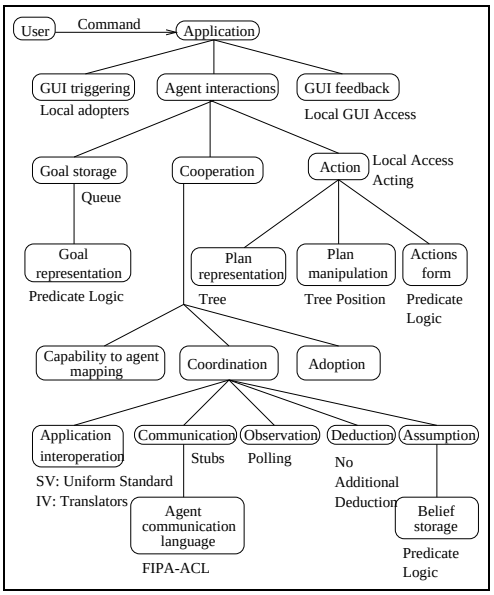


Figure A.29: Infrastructure modularisation annotated with application infrastructure part model decisions.

A.8 Results

The application infrastructure parts are annotated to the modularisation diagram shown in Figure A.29 and the agent infrastructure parts are collated into agents to be implemented in Table A.38. In the next chapter, we highlight the points in the example application that illustrate how this methodology has justified the entire design from the requirements, and how this compares to the approaches of other methods. We also consider the issues involved in implementing the application and agent infrastructure parts.

Bibliography

[1] H. H. Adelsberger and W. Conen. Economic Coordination Mechanisms for Holonic Multi Agent Systems . In *Workshop on Industrial Applications of Holonic and Multi-Agent Systems (HoloMAS 2000)*, 2000. Also available at http://nestroy.wi-inf.uni-essen.de/ conen/conen/publications.html.

[2] D. Amyot, L. Logrippo, R. J. A. Buhr, and T. Gray. Use Case Maps for the capture and validation of distributed systems requirements. In *Fourth International Symposium on Requirements Engineering (RE-99)*, 1999.

[3] Y. Aridor and D. B. Lange. Agent design patterns: Elements of agent application design. In *Proceedings of the Second International Conference on Autonomous Agents (Agents-98)*, Minneapolis, USA, 1998.

[4] Unspecified authors. Unified modeling language (UML) notation guide. Technical report, Various institutions, September 1997.

[5] K. S. Barber, D. C. Han, and T. Liu. Strategy Selection-Based Meta-level Reasoning for Multi-agent Problem-Solving. In P. Ciancarini and M. J. Wooldridge, editors, *Proceedings of Agent-Oriented Software Engineering 2000 (AOSE 2000)*, pages 269–283. Springer-Verlag.

[6] M. Barbuceanu and M. S. Fox. The design of a coordination language for multi-agent systems. In J.P. Müller, M.J. Wooldridge, and N.R. Jennings, editors, *Intelligent Agents III: Proceedings of the Third International Workshop on Agent Theories, Architectures and Languages (ATAL-96)*, pages 341–355. Springer-Verlag, 1996.

[7] B. Bauer, J. P. Müller, and J. Odell. Agent UML: A Formalism for Specifying Multi-agent Software Systems. In P. Ciancarini and M. J. Wooldridge, editors, *Proceed-*

ings of Agent-Oriented Software Engineering 2000 (AOSE 2000), pages 91–104. Springer-Verlag, 2000.

[8] M. Beer, M. d'Inverno, M. Luck, N. Jennings, C. Priest, and M. Schroeder. Negotiation in multi-agent systems. *Knowledge Engineering Review*, 14(3):285–290, 1999.

[9] F. Bellifemine, A. Poggi, and G. Rimassa. JADE — a FIPA-compliant agent framework. In *Proceedings of the Fourth International Conference and Exhibition on The Practical Application of Intelligent Agents and Multi-Agents*, 1999.

[10] T. J. M. Bench-Capon and P. E. Dunne. No agent is an island: A framework for the study of inter-agent behaviour. In C. Castelfranchi and W. L. Johnson, editors, *Proceedings of the First International Joint Conference on Autonomous Agents and MultiAgent Systems 2002 (AAMAS-2002)*, pages 690–691, Bologna, Italy, 2002.

[11] F. Bergenti. On agentware: Ruminations on why we should use agents. In *Proceedings of the Third International Workshop on Engineering Societies in the Agents World (ESAW-2002)*, Madrid, Spain, 2002.

[12] A. Bonarini and M. Restelli. An architecture to implement agents co-operating in dynamic environments. In C. Castelfranchi and W. L. Johnson, editors, *Proceedings of the First International Joint Conference on Autonomous Agents and MultiAgent Systems 2002 (AAMAS-2002)*, pages 1143–1144, Bologna, Italy, 2002.

[13] M. E. Bratman. Shared cooperative activity. *Philosphical Review*, 101(2):327–341, April 1992.

[14] M. E. Bratman, D. Israel, and M. Pollack. Plans and resource-bounded practical reasoning. *Computational Intelligence*, 4:349–355, 1988.

[15] W. Brauer, M. Nickles, G. Weiss, and K. F. Lorentzen. Expectation-oriented analysis and design. In M. Wooldridge, P. Ciancarini, and G. Weiss, editors, *Proceedings of the Second International Workshop on Agent-Oriented Software Engineering (AOSE-2001)*, pages 226–244, Montreal, Canada, 2001.

[16] P. Bresciani, A. Perini, P. Giogini, F. Giunchiglia, and J. Mylopolous. Modeling early requirements in Tropos: A transformation based approach. In M. Wooldridge, P. Ciancarini, and G. Weiss, editors, *Proceedings of the Second International Workshop on Agent-Oriented Software Engineering (AOSE-2001)*, pages 151–168, Montreal, Canada, 2001.

[17] R. J. A. Buhr, D. Amyot, E. Elammari, D. Quesnel, T. Gray, and S. Mankovski. Feature-interaction visualization and resolution in an agent environment. In *Proceedings of the Fifth International Workshop on Feature Interactions in Telecommunications and Software Systems*, pages 135–149, 1998.

[18] S. Bussmann, N. R. Jennings, and M. Wooldridge. On the identification of agents in the design of production control systems. In P. Ciancarini and M. J. Wooldridge, editors, *Proceedings of Agent-Oriented Software Engineering 2000 (AOSE 2000)*, pages 141–162. Springer-Verlag, 2000.

[19] C. Castelfranchi and R. Falcone. Principles of trust in MAS: Cognitive autonomy, social importance, and quantification. In *Proceedings of the Third International Conference on Multi-Agent Systems (ICMAS-98)*, pages 72–79, 1998.

[20] J. Castro, M. Kolp, and J. Mylopoulos. A requirements-driven development methodology. In *Proceedings of the 13th International Conference on Advanced Information Systems Engineering (CAiSE-01)*, Interlaken, Switzerland, 2001.

[21] P. Ciancarini, A. Omicini, and F. Zambonelli. Multiagent system engineering: The coordination viewpoint. In N. R. Jennings and Y. Lesperance, editors, *Intelligent Agents VI: Proceedings of the Sixth International Workshop on Agent Theories, Architectures and Languages (ATAL-99)*, Orlando, Florida, USA, 2000. Also available at http://sirio.dsi.unimo.it/Zambonelli/pubblica.html.

[22] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes. *Object-Oriented Development: The Fusion Method*. Prentice Hall, 1994.

[23] A. Dardenne, A. van Lamsweerde, and S. Fickas. Goal-directed requirements analysis. *Science of Computer Programming*, 20:3–50, 1993.

[24] P. Davidsson. Categories of artificial societies. In A. Omicini, P. Petta, and R. Tolksdorf, editors, *Proceedings of the Second International Workshop on Engineering Societies in the Agents World (ESAW-2001)*, pages 1–9, Prague, Czech Republic, 2001. Springer-Verlag.

[25] P. Davidsson and S. Johansson. Evaluating multi-agent system architectures: A case study concerning dynamic resource allocation. In *Proceedings of the Third International Workshop on Engineering Societies in the Agents World (ESAW-2002)*, Madrid, Spain, 2002.

[26] A. M. Davis. *Software Requirements: Objects, States and Functions.* Prentice Hall, 1993.

[27] K. S. Decker and V. R. Lesser. Designing a family of coordination algorithms. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95)*, pages 73–80, 1995.

[28] R. Depke, R. Heckel, and J. M. Küster. Agent-oriented modeling with graph transformations. In P. Ciancarini and M. J. Wooldridge, editors, *Proceedings of Agent-Oriented Software Engineering 2000 (AOSE 2000)*, pages 105–120. Springer-Verlag, 2000.

[29] R. Deters. Developing and deploying a multi-agent system. In *Proceedings of the Forth International Conference on Autonomous Agents (Agents-00)*, 2000.

[30] F. Dignum and M. Greaves. Technology for agent communication. In F. Dignum and M. Greaves, editors, *Issues in Agent Communication (LNCS-1916)*, pages 1–16. Springer-Verlag, 2000.

[31] V. Dignum, H. Weigand, and L. Xu. Agent Societies: Toward Frameworks-Based Design. In M. Wooldridge, P. Ciancarini, and G. Weiss, editors, *Proceedings of the Second International Workshop on Agent-Oriented Software Engineering (AOSE-2001)*, pages 25–32, Montreal, Canada, 2001.

[32] E. H. Durfee and V. R. Lesser. Using partial global plans to coordinate distributed problem solvers. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence (IJCAI-87)*, Milan, Italy, 1987.

[33] C. B. Excelente-Toledo, R. A. Bourne, and N. R. Jennings. Reasoning about Commitments and Penalties for Coordination between Autonomous Agents . In M. Wooldridge, P. Ciancarini, and G. Weiss, editors, *Proceedings of the Second International Workshop on Agent-Oriented Software Engineering (AOSE-2001)*, Montreal, Canada, 2001.

[34] C. B. Excelente-Toledo and N. R. Jennings. Learning to select a coordination mechanism. In C. Castelfranchi and W. L. Johnson, editors, *Proceedings of the First International Joint Conference on Autonomous Agents and MultiAgent Systems 2002 (AAMAS-2002)*, pages 1106–1113, Bologna, Italy, 2002.

[35] R. Falcone and B. S. Firozabadi. The challenge of trust: The Autonomous Agents '98 Workshop on Deception, Fraud and Trust in Agent Societies. *Knowledge Engineering Review*, 14(1):81–89, 1999.

[36] T. Finin, R. Fritzson, D. McKay, and R. McEntire. KQML — A Language and Protocol for Knowledge and Information Exchange. In K. Fuchi and T. Yokoi, editors, *Knowledge Building and Knowledge Sharing*. IOS Press, 1994.

[37] T. Finin, R. Fritzson, D. McKay, and R. McEntire. KQML as an Agent Communication Language. In N. Adam, B. Bhargava, and Y. Yesha, editors, *Proceedings of the 3rd International Conference on Information and Knowledge Management (CIKM'94)*, pages 456–463, Gaithersburg, MD, USA, 1994. ACM Press.

[38] M. Fisher and W. Wooldridge. On the formal specification and verification of multi-agent systems. *International Journal of Cooperative Information Systems*, 6(1):37–65, 1997. Also available at http://www.csc.liv.ac.uk/ mjw/pubs/.

[39] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns: Abstraction and reuse of object oriented design. In *Proceedings of ECOOP'93, Kaiserslautern, Germany*, 1993.

[40] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software* . Addison Wesley, Reading, Mass., 1995.

[41] L. Gasser. MAS Infrastructure Definitions, Needs and Prospects. In *Proceedings of the Forth International Conference on Autonomous Agents (Agents-00)*, 2000.

[42] M. Georgeff, B. Pell, M. Pollack, M. Tambe, and M. Wooldridge. The belief-desire-intention model of agency. In *Intelligent Agents V: Proceedings of the Fifth International Workshop on Agent Theories, Architectures and Languages (ATAL-98)*, 1998.

[43] M.P. Georgeff and A.L. Lansky. Reactive reasoning and planning. In *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87)*, pages 677–682, Seattle, WA, 1987.

[44] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering*. Prentice-Hall International, 1991.

[45] F. Giunchiglia, J. Mylopoulos, and A. Perini. The Tropos software development

methodology: Processes, models and diagrams. In C. Castelfranchi and W. L. Johnson, editors, *Proceedings of the First International Joint Conference on Autonomous Agents and MultiAgent Systems 2002 (AAMAS-2002)*, pages 35–36, Bologna, Italy, 2002.

[46] P. J. Gmytrasiewicz and E. H. Durfee. Toward a theory of honesty and trust among communicating autonomous agents. *Group Decision and Negotiation*, 2:237–258, 1993.

[47] P. J. Gmytrasiewicz and E. H. Durfee. A rigorous, operational formalization of recursive modeling. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95)*, pages 125–132, San Francisco, CA, June 1995.

[48] N. Griffiths and M. Luck. Cooperative plan selection through trust. In *Proceedings of the European Workshop on Modelling Autonomous Agents in a Multi-Agent World 1999 (MAAMAW-99)*, 1999.

[49] F. Guerin and J. Pitt. Proving properties of open agent systems. In C. Castelfranchi and W. L. Johnson, editors, *Proceedings of the First International Joint Conference on Autonomous Agents and MultiAgent Systems 2002 (AAMAS-2002)*, Bologna, Italy, 2002.

[50] P. Haumer, K. Pohl, and K. Weidenhaupt. Requirements elicitation and validation with real world scenes. *IEEE Transactions on Software Engineering*, 24(12), December 1998.

[51] S. Hayden, C. Carrick, and Q. Yang. Architectural design patterns for multi-agent coordination. In *Proceedings of the International Conference on Agent Systems 1999 (Agents'99)*, Seattle, WA, 1999.

[52] C. E. Hewitt and P. de Jong. Analyzing the roles of descriptions and actions in open systems. In *Proceedings of the Third National Conference on Artificial Intelligence (AAAI-83)*, pages 162–166, 1983.

[53] M. N. Huhns. Interaction-oriented programming. In P. Ciancarini and M. J. Wooldridge, editors, *Proceedings of Agent-Oriented Software Engineering 2000 (AOSE 2000)*, 2000.

[54] C. A. Iglesias, M. Garijo, and J. C. Gonzalez. A survey of agent-oriented methodologies. In J. P. Müller, M. P. Singh, and A. S. Rao, editors, *Intelligent Agents V:*

Proceedings of the Fifth International Workshop on Agent Theories, Architectures and Languages (ATAL-98), 1998.

[55] N. Jennings. On argumentation-based negotiation. In *Proceedings of the International Workshop on Multi-Agent Systems (IWMAS-1998)*, 1998.

[56] N. R. Jennings. Commitments and conventions: The foundation of coordination in multi-agent systems. *Knowledge Engineering Review*, 8(3):223–250, 1993.

[57] N. R. Jennings. On agent-based software engineering. *Artificial Intelligence*, 117:277–296, 2000.

[58] N. R Jennings. Building Complex Software Systems: The Case for an Agent-Based Approach. *Communications of the ACM*, 2001.

[59] N. R. Jennings, E. H. Mamdani, I. Laresgotti, J. Perez, and J. Corera. Grate: A general framework for cooperative problem solving. *IEE-BCS Journal of Intelligent Systems Engineering*, 1(2), 1992.

[60] N. R. Jennings, K. Sycara, and M. Wooldridge. A roadmap of agent research and development. *Autonomous Agents and Multi-Agent Systems*, 1(1):275–306, 1998.

[61] N. R. Jennings and M. Wooldridge. Agent-oriented software engineering. In *Proceedings of the European Workshop on Modelling Autonomous Agents in a Multi-Agent World 1999 (MAAMAW-99)*, 1999. Also available at http://www.ecs.soton.ac.uk/ nrj/pubs.html.

[62] T. Juan, A. Pearce, and L. Sterling. ROADMAP: Extending the Gaia methodology for complex open systems. In C. Castelfranchi and W. L. Johnson, editors, *Proceedings of the First International Joint Conference on Autonomous Agents and MultiAgent Systems 2002 (AAMAS-2002)*, pages 3–10, Bologna, Italy, 2002.

[63] E. A. Kendall. Agent software engineering with role modelling. In P. Ciancarini and M. J. Wooldridge, editors, *Proceedings of Agent-Oriented Software Engineering 2000 (AOSE 2000)*, pages 163–170. Springer-Verlag, 2000.

[64] E. A. Kendall, M. T. Malkoun, and C. Jiang. A methodology for developing agent based systems for enterprise integration. In *First Australian Workshop on Distributed Artificial Intelligence*, 1995.

[65] M. Klusch and K. Sycara. Brokering and matchmaking for coordination of agent societies: A survey. In A. Omicini, F. Zambonelli, M. Klusch, and R. Tolksdorf, editors, *Coordination of Internet Agents*, pages 197–224. Springer-Verlag, 2001.

[66] B. Langley, M. Paolucci, and K. Sycara. Discovery of infrastructure in multi-agent systems. In *Agents 2001 Workshop on Infrastructure for Agents, MAS, and Scalable MAS*, 2001.

[67] Y. Larou, T. Finin, and Y. Peng. Agent communication languages: The current landscape. *IEEE Intelligent Systems*, 14(2):45–52, 1999.

[68] B. Logan. Classifying agent systems. Technical Report WS-98-10, School of Computer Science, University of Birmingham, 1998.

[69] F. Lopez y Lopez, M. Luck, and M. d'Inverno. Constraining autonomy through norms. In C. Castelfranchi and W. L. Johnson, editors, *Proceedings of the First International Joint Conference on Autonomous Agents and MultiAgent Systems 2002 (AAMAS-2002)*, pages 674–681, Bologna, Italy, 2002.

[70] M. Luck and M. d'Inverno. Structuring a Z specification to provide a formal framework for autonomous agent systems. In J. P. Bowen and M. G. Hinchey, editors, *Proceedings of the Ninth International Conference of Z Users (ZUM-95)*, pages 47–62, Heidelberg, 1995. Springer-Verlag.

[71] M. Luck and M. d'Inverno. Engagement and cooperation in motivated agent modelling. In C. Zhang and D. Lukose, editors, *Distributed Artificial Intelligence Architecture and Modelling: Proceedings of the First Australian DAI Workshop*, pages 70–84. Springer-Verlag, 1996.

[72] M. Luck, N. Griffiths, and M. d'Inverno. From agent theory to agent construction: A case study. In J.P. Müller, M.J. Wooldridge, and N.R. Jennings, editors, *Intelligent Agents III: Proceedings of the Third International Workshop on Agent Theories, Architectures and Languages (ATAL-96)*, pages 49–63. Springer-Verlag, 1997.

[73] S. P. Marsh. *Formalising Trust as a Computational Concept*. PhD thesis, Department of Computing Science and Mathematics, University of Stirling, 1994.

[74] D. L. Martin, A. J. Cheyer, and D. B. Moran. The open agent architecture: A framework for building distributed software systems. *Applied Artificial Intelligence*, 13(1-2), January 1999.

[75] J. Mayfield, Y. Labrou, and T. Finin. Evaluation of kqml as an agent communication language. In M. J. Wooldridge, J. P. Müller, and M. Tambe, editors, *Intelligent Agents II: Proceedings of the Second International Workshop on Theories, Architectures and Languages (ATAL-95)*, pages 347–360. Springer-Verlag, 1995.

[76] S. Miles, M. Joy, and M. Luck. Designing agent-oriented systems by analysing agent interactions. In P. Ciancarini and M. J. Wooldridge, editors, *Proceedings of Agent-Oriented Software Engineering 2000 (AOSE 2000)*, pages 171–184, 2000.

[77] S. Miles, M. Joy, and M. Luck. Towards a methodology for coordination mechanism selection in open systems. In *Proceedings of the Third International Workshop on Engineering Societies in the Agents World (ESAW-2002)*, Madrid, Spain, 2002.

[78] L. Moreau, N. Gibbins, D. DeRoure, S. El-Beltagy, W. Hall, G. Hughes, D. Joyce, S. Kim, S. Michaelides, S. Millard, S. Reich, R. Tansley, and M. Weal. Sofar with dim agents: An agent framework for distributed information management. In *Proceedings of The Fifth International Conference and Exhibition on The Practical Application of Intelligent Agents and Multi-Agents*, pages 369–388, Manchester, UK, 2000.

[79] J. Odell, H. V. D. Parunak, and B. Bauer. Extending UML for agents. In G. Wagner, Y. Lesperance, and E. Yu, editors, *Proceedings of the Agent-Oriented Information Systems Workshop at the 17th National conference on Artificial Intelligence (AOIS-00)*, pages 3–17, Austin, TX, 2000.

[80] J. Odell, H. V. D. Parunak, and B. Bauer. Representing agent interaction protocols in UML. In *Proceedings of Agent-Oriented Software Engineering 2000 (AOSE 2000)*, 2000.

[81] Foundation of Intelligent Physical Agents. http://www.fipa.org.

[82] A. Omicini. From objects to agent societies: Abstractions and methodologies for the engineering of open systems. In *WOA-00*, Parma, Italy, 2000. Also available at http://lia.deis.unibo.it/ ao/.

[83] A. Omicini. SODA: Societies and infastructures in the analysis and design of agent-based systems. In P. Ciancarini and M. J. Wooldridge, editors, *Proceedings of Agent-Oriented Software Engineering 2000 (AOSE 2000)*, 2000.

[84] L. Padgham and M. Winikoff. Prometheus: A methodology for developing intelligent agents. In C. Castelfranchi and W. L. Johnson, editors, *Proceedings of the First International Joint Conference on Autonomous Agents and MultiAgent Systems 2002 (AAMAS-2002)*, pages 37–38, Bologna, Italy, 2002.

[85] P. Panzarasa and N. R. Jennings. The organisation of sociality: A manifesto for a new science of multi-agent systems. In *Proceedings of the Tenth European Workshop on Multi-Agent Systems (MAAMAW-01)*, 2001.

[86] H. V. D. Parunak and J. J. Odell. Representing social structures in UML. In M. Wooldridge, P. Ciancarini, and G. Weiss, editors, *Proceedings of the Second International Workshop on Agent-Oriented Software Engineering (AOSE-2001)*, pages 1–16, Montreal, Canada, 2001.

[87] C. Petrie. Agent-based software engineering. In P. Ciancarini and M. J. Wooldridge, editors, *Proceedings of Agent-Oriented Software Engineering 2000 (AOSE 2000)*, pages 59–75. Springer-Verlag, 2000.

[88] D. Poutakidis, L. Padgham, and M. Winikoff. Debugging multi-agent systems using design artifacts: The case of interaction protocols. In C. Castelfranchi and W. L. Johnson, editors, *Proceedings of the First International Joint Conference on Autonomous Agents and MultiAgent Systems 2002 (AAMAS-2002)*, Bologna, Italy, 2002.

[89] L. Rising. Pattern writing. In L. Rising, editor, *The Patterns Handbook: Techniques, Strategies and Applications*, pages 69–82. Cambridge University Press, 1998.

[90] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.

[91] O. Shehory. Software architecture attributes of multi-agent systems. In P. Ciancarini and M. J. Wooldridge, editors, *Proceedings of Agent-Oriented Software Engineering 2000 (AOSE 2000)*, pages 77–90. Springer-Verlag, 2000.

[92] M. P. Singh. Agent communication languages: Rethinking the principles. *IEEE Computer*, 31(12):40–47, December 1998.

[93] I. Sommerville. *Software Engineering.* Addison-Wesley, fifth edition edition, 1995.

[94] K. Sycara. The RETSINA MAS infrastructure. *Journal of Autonomous Agents and Multi-Agent Systems*, 2001.

[95] A. van Lamsweerde. Requirements engineering in the year 00: A research perspective. In *Proceeding of the Twenty-Second International Conference on Software Engineering (ICSE-00)*, 2000.

[96] A. van Lamsweerde and L. Willemet. Inferring declarative requirements specifications from operational scenarios. *IEEE Transactions on Software Engineering*, 1998.

[97] W. W. Vasconcelos, J. Sabater, C. Sierra, and J. Querol. Skeleton-based agent development for electronic institutions. In C. Castelfranchi and W. L. Johnson, editors, *Proceedings of the First International Joint Conference on Autonomous Agents and MultiAgent Systems 2002 (AAMAS-2002)*, pages 696–703, Bologna, Italy, 2002.

[98] G. Wagner. Towards Agent-Oriented Information Systems. 2000.

[99] F. Wang. Self-organising communities formed by middle agents. In C. Castelfranchi and W. L. Johnson, editors, *Proceedings of the First International Joint Conference on Autonomous Agents and MultiAgent Systems 2002 (AAMAS-2002)*, pages 1333–1339, Bologna, Italy, 2002.

[100] G. Weiß. Agent orientation in software engineering. *Knowledge Engineering Review*, 16(4), December 2001.

[101] M. Wooldridge and P. Ciancarini. Agent-oriented software engineering: The state of the art. In P. Ciancarini and M. J. Wooldridge, editors, *Proceedings of Agent-Oriented Software Engineering 2000 (AOSE 2000)*, 2000.

[102] M. Wooldridge, N. R. Jennings, and D. Kinny. The Gaia methodology for agent-oriented analysis and design. *Journal of Autonomous Agents and Multi-Agent Systems*, (3), 2000.

[103] P. Yolum and M. P. Singh. Synthesizing Finite State Machines for Communication Protocols . Technical report, Department of Computer Science, North Carolina State University, USA, June 2001.

[104] E. Yu. Towards modelling and reasoning support for early-phase requirements engineering, washington d.c., usa. In *Proceedings of the 3rd IEEE Int. Symp. on Requirements Engineering (RE'97)*, pages 226–235, January 1997.

[105] E. Yu. Agent-oriented modelling: Software versus the world. In M. Wooldridge, P. Ciancarini, and G. Weiss, editors, *Proceedings of the Second International Workshop on Agent-Oriented Software Engineering (AOSE-2001)*, pages 206–225, Montreal, Canada, 2001.

[106] L. Yu and B. F. Schmid. A conceptual framework for agent oriented and role based workflow modeling. In *Proceedings of Agent-Oriented Information Systems 1999 (AOIS-99)*, 1999.

[107] F. Zambonelli, N. R. Jennings, and M. Wooldridge. Organisational abstractions for the analysis and design of multi-agent systems. In P. Ciancarini and M. J. Wooldridge, editors, *Proceedings of Agent-Oriented Software Engineering 2000 (AOSE 2000)*, 2000.

[108] F. Zambonelli, N. R. Jennings, and M. Wooldridge. Organisational rules as an abstraction for the analysis and design of multi-agent systems. *International Journal of Software Engineering and Knowledge Engineering*, 2000.

[109] F. Zambonelli and H. V. D. Parunak. Signs of a revolution in computer science and software engineering. In *Proceedings of the Third International Workshop on Engineering Societies in the Agents World (ESAW-2002)*, Madrid, Spain, 2002.