UNIVERSITY OF SOUTHAMPTON

Models for Agent-Based Infrastructures

by
Ronald Ashri

A thesis submitted in partial fulfillment for the
degree of Doctor of Philosophy

in the
School of Electronics and Computer Science

September 2004

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

SCHOOL OF ELECTRONICS AND COMPUTER SCIENCE

Doctor of Philosophy

MODELS FOR AGENT-BASED INFRASTRUCTURES

by Ronald Ashri

Computing environments are undergoing a drastic transformation with the introduction of diverse devices with varying capabilities into networked environments and infrastructure that enables the exchange of information and the collaboration between devices in a number of modes. While it is becoming easier to connect practically any computing device through a network and embed computing devices unobtrusively in a wide range of real-world artifacts, it is becoming more difficult to develop software systems that can deal with the inherent dynamics and complex interactions of the resulting distributed computing environments.

Agent-based systems have a key role to play in the effort to provide and support such applications, since agents embody several of the required characteristics for effective and robust operation in dynamic and heterogenous computing environments. However, there are a number of shortcomings relating to the use of the agent approach to application development. In particular, in this thesis we deal with the lack of clarity in existing agent models and address the need for models that can directly support practical application development. These are widely-accepted shortcomings that have been identified by a number of researchers in recent years [8, 32, 136, 189, 227, 231]. This thesis addresses these shortcomings with relation to the basic infrastructural concerns that are common to practically all significant agent-based applications in dynamic, heterogeneous environments. We develop *principled* and *reusable* models in support of agent-based systems construction, dealing both within individual agent construction and support for relationship identification and characterisation.

In this thesis we make three main contributions. Firstly, through an abstract agent model we enable the characterisation of the wide range of agent types that can exist within a heterogenous environment. This facilitates development by ensuring that the underlying theory adequately models the actual application environment and provides indications as to where designers must focus their efforts. Secondly, we develop a model for agent construction which links the abstract agent model to practical application concerns and enables the specification of a range of agent architectures while also facilitating their run-time reconfiguration. This bridges the gap between abstract models and practical implementation, allows developers to choose the type of agent architecture that best suits the application at hand, and provide the flexibility for adapting architectures to changing application needs. Finally, we develop a model of agent interaction and use it to comprehensively identify all the possible relationships between two agents, as well as to relate agent goals to the abilities of agents to achieve those goals given their individual capabilities. This enables the effective identification and characterisation of agent relationships in dynamic environments, so as to guide the choice of appropriate relationship management mechanisms.

Contents

List of Figures

Acknowledgements

Writing a PhD is an unforgivably solitary task, which makes the people that supported me throughout the period all the more important.

Firstly, a big thanks to Michael Luck who more than being an excellent supervisor throughout the process, is a true mentor and friend. His dedication to his work and, more importantly, to the people he works with sets an example that I can only hope to be able to live up to.

My many thanks also to Mark d'Inverno – when things just seemed to stall he was always there to plant the seed for another idea.

The work in this thesis would not have been possible without a grant from BT Exact – my thanks to Simon Thompson for handling the relationship with BT Exact.

A big *salute*, *salud*, *a votre sante*, *cheers*, *prosit*, *l'chaim*, *fisehatak* and *eis igeian* to the many friends at Southampton and all over the world which I made during the years of the PhD: Steve, Serena, Claudia, Fabiola, Iyad, Gopal, Raj, Jordi, Vijay, Martina, Georgia, Cora and Alejandro, Marco and Fariba, Grit, Jorge, Arturo, Roxana, Darren, Talal and Shaza, Craig, Colm. My thanks also to Alex, Christos, Nikolas, Olivia, Richard, Sergio, Lorenzo, Demetris, my brother Daniel, and to the ever so wonderful Carla and Titta (and a ginger dog named Lilly, who sat for countless hours next to me wondering why I was staring at a screen instead of taking her out for walk).

Finally, my love and thanks to Katia. No words can express how important her support has been. Having shared the journey of both working for a PhD we decided somewhere along the way to share the journey of the rest of our lives.

To my parents, Andreas and Photini.

Chapter 1

Introduction

"There is nothing like a dream to create the future."

Victor Hugo (1802-1885); writer.

1.1 The Need for Agent-Based Computing

Continual advances in basic networking technologies, processing capabilities and device miniaturisation have, over recent years, allowed computing devices to pervade every aspect of society. Through unifying infrastructures such as the Internet, and the establishment of standards for access to, and presentation of, information such as HTTP [93], HTML [108] and XML [34], devices as diverse as mobile phones, personal digital assistants (PDAs) and desktop computers can be interconnected to share information and services. The combination of these hardware and software advances has created an environment in which the ability to communicate and access online services using networked computing devices, at any time and irrespective of geographic location, is fast being realised.

As a result, new opportunities for innovative applications are being created, but new challenges are also being posed relating to the appropriate development of tools that will enable us to best exploit the available technology. Because of the increasing drive for creating new *kinds* of applications for users, combining the underlying networking and processing capabilities with powerful software tools, concerns with issues of development are becoming correspondingly important. Examples of the kinds of applications envisaged are dynamic online environments for providing e-services to users [187], the formation of *virtual organisations* through the dynamically-determined cooperation of existing organisations [159], the use of mobile devices

interacting with enterprise systems [149], more powerful and flexible mechanisms for scientific computation [68], and so on.

All these applications share the need to support interaction between disparate components that typically operate independently of each other, in dynamic and heterogeneous environments. However, in order to facilitate such interactions, developers must address several issues such as the great variety in computing device capabilities, the range of operating systems and network protocols, users moving and accessing services through changing geographical locations while dealing with a number of different organisational domains, and the possible loss of power and network connectivity for mobile devices.

In response to these challenges, agent-based computing has been suggested as a paradigm that can provide the conceptual grounding to enable application developers to effectively deal with the problems they raise [99, 120, 138, 140]. The basic concept of agents capable of individual, independent action, working towards their design goals, while at the same time able to interact with other agents through automated means in order to resolve issues relevant to individual or common goals, represents an intuitive and natural starting point for solutions. However, current work on agent-based computing has several shortcomings relating both to its foundations and to its use in the design of applications. This thesis aims to address some of these shortcomings by providing *principled* and *reusable* models for supporting the development and analysis of agent systems, focusing on the basic issues of individual agent construction and support for cooperation between agents.

Before providing a more detailed description of our aims, we briefly examine the various technological changes, and the new demands from users, that have driven the move towards adopting an agent perspective in developing applications. Subsequently, we present arguments for the suitability of the agent paradigm and discuss its advantages and limitations. Based on this discussion, we introduce our aims and then, finally, the main contributions are presented. We conclude with an overview of the rest of the thesis.

1.2 Emerging Computing Environments

A defining characteristic of computing environments of the past ten years has been the level of interconnectedness between computers. More recent developments have broadened these characteristics to include the diversification of computing devices from powerful workstations to

mobile devices and embedded devices, and the improvement of wireless communications. Not only are we now able to interconnect computing devices, but we are able to do so with unprecedented flexibility. In this section, we examine how these developments affect the different spaces in which we as humans operate, and the challenges they place on application developers attempting to make the best use of these emerging computing environments. Throughout the discussion, we provide examples of suggested agent solutions, illustrating the suitability of the approach to these problems.

1.2.1 Work Environments

Large organisations, in both the private and public sector, have, predictably, been the first to build applications that take advantage of network technologies in an attempt to integrate their information processing systems. However, their use of computers over a relatively large number of years has resulted in an unavoidable reliance on older (*legacy*) systems which, for a number of reasons, cannot simply be replaced by new ones, but must be integrated into existing structures [39]. This integration of legacy systems with new infrastructure is one of the classic problems for which an agent-based approach has been suggested as an appropriate solution [100, 123].

Furthermore, the proliferation of desktop computers created the possibility, and subsequently the need, to network these computers in order to improve access to information and collaboration between individuals, in turn creating three basic requirements. Firstly, such networks need to be supported and administered in the face of increasing complexity and heterogeneity. Secondly, users require secure access to a number of different information sources and applications. Finally, the users of these systems also require specific applications to support more direct modes of collaboration across networked computers that reflect the more global structure of organisations, with resources dispersed across a number of different locations and a need to access them at any time. This not only adds a new level of heterogeneity to the system but also challenges traditionally held ideas about software engineering for distributed systems. Developers can no longer abstract out location, reliability and bandwidth issues, since networks can no longer be administered as if they were a reliable, homogeneous collection of resources. Such challenges provide good application areas for illustrating how the agent paradigm can be used to aid in complex information management tasks and a number of agent-based solutions demonstrate this (e.g. [43, 51, 74, 167, 211, 226]). All these examples take advantage of the benefits of decentralised and loosely coupled systems, which can deal with changing operating

conditions.

Finally, calls for enabling a more decentralised, team-based mode of operation *between* large, global organisations over the Internet are increasing [72, 147]. These types of organisations add yet another challenge to system design, since we now need to enable and control collaboration between domains that are likely to have very different ways of describing information as well as different administration policies. While basic networking capabilities can aid in forming such strategic collaborations between organisations [97], many have suggested that effective solutions can only come about through a significant shift from traditional object-oriented techniques for building distributed systems to the inclusion of artificial intelligence and agent-based approaches [1, 158, 183] that incorporate organisational and societal notions to regulate the interactions between the different parts of the organisation.

1.2.2 Social and Home Environments

Outside a purely commercial context, the Internet has enabled the creation of numerous online communities that share information and collaborate on issues of common interest, ranging from the development of open-source software to political and social movements. At home, we have managed to take advantage of the Internet through personal computers which, typically, use the telephone line to provide a connection. Broadband services that facilitate the flow of richer types of media are also, increasingly, entering homes.

In addition, embedded devices are becoming networked-enabled[1], providing a new kind of connection to the Internet through means that are less obvious. New models of common domestic devices, such as the washing machine, the refrigerator and the television are being marketed as *intelligent devices* that can send information about their status to the manufacturer for maintenance, or can allow us to control them remotely. The whole concept of *the space in which we live* is being reshaped into the notion of an *intelligent* or *ambient* environment in which every parameter, from the temperature to the decoration on walls, can be dynamically fine-tuned to suit our wishes. More significantly, we can also create intelligent *environments* that can provide assistance when it is required for health reasons through the dynamic interactions of devices that monitor the vital statistics of a patient, such as heart rate and blood pressure. In this context, agent-based approaches have been used to create a variety of such smart home

[1]Large consumer product manufacturers have already stated that their policy is to network-enable every device they produce as soon as possible.(Red Herring, March 1st, 2001)

services [110, 112, 180], since agents are well suited to acting as abstractions for the various devices in a home and managing their interactions with humans and other services outside the home.

At the societal level, the integration of networking technologies into our homes could lead to dramatic changes in the provision of public services and the modalities of participation in politics. Already, there is work being done on enabling voting in official elections to take place through mobile phones or the Internet [115], while the debate on the wider impact of the Internet on the way societies govern themselves is just beginning [46, 224]. In any case, it is clear that agent technologies will have a significant role to play in the development of applications at this level, as evidenced by some initial efforts [142].

1.2.3 Mobile Users

The spread, and rise in influence, of wireless mobile devices enables us to maintain connections to other networks while on the move. This capability is beginning to have a profound impact on the concept of the work-space, since *geographical location* is losing importance while the *virtual space* is becoming more and more significant. Workers are increasingly tempted and, sometimes, even encouraged, to abandon the daily commute to the office in favour of a virtual connection. However, as mobile users change geographical locations, they very often also have to change service providers for access to online services, raising further problems of interoperability and security. Significant challenges are raised here in terms of accommodating the free flow of information and devices between different administration domains. In these scenarios, the combination of intelligent agents and techniques from mobile agent research are providing effective solutions [19, 111, 169, 192].

1.3 Enabling Infrastructures

As should be apparent from the discussion above, the overarching feature of this changing landscape of networked computing environments is that of *heterogeneous* networks of devices and users in which communication and collaboration takes place at many different levels, and information and services need to be available in a number of different modes to satisfy user needs. It is also clear that certain issues, such as the support of mobile users, collaboration between

organisations, and providing customer services over the Internet, are problems on a scale unprecedented in software engineering. They require global structures and the cooperation of a number of service providers in order to function. In turn, this requires distributed systems in which the communication and coordination between the disparate components is able to operate over, and adapt to, changing needs and changing resources. At the same time, these components require some level of *individual* control to be able to adapt to changes when reliable centralised control can no longer be taken for granted, or when it cannot be as effective as decentralised control.

To develop these new types of systems, appropriate paradigms are required that will embrace the challenges set and deal with them directly. The *scale* and *diversity* of the challenges means that any paradigm needs to embrace a variety of different techniques in order to provide the appropriate set of tools for application development. Note that we do not aim to tackle all of these issues in the thesis but we discuss the relationships between them here so as to motivate the need for agent-based computing as the underlying paradigm that can provide a conceptual underpinning for applications that touch on all of these issues. We will discuss later on in the chapter which are the precise issues that we aim to tackle within the context of agent-based computing.

At the most basic level, we must be able to construct *dynamic networks* that can handle continuous changes in the number and types of devices available. Secondly, we need to deal with *mobility*, both of users and possibly of code. At the same time mobility in part, also *creates* the need for dynamic networking since the introduction or removal of devices inevitably causes changes to the configuration of a network.Finally, we require *intelligent* applications that are able to react to change and undertake tasks with a certain degree of independence from human guidance. These three basic issues are currently tackled in relative isolation, as research areas in their own right. However, the envisioned applications require all of them to operate in unison. We illustrate this in Figure 1.1, in which the Venn diagram illustrates the location of origin and the relationships between them.[2] The overlaps between the issues of intelligence, mobility and dynamic networking indicate that there are sub-fields that are common to different areas, while the unison of all three can provide the required enabling infrastructure. A description of each follows.

- *Dynamic networking* refers to technologies that allow the dynamic creation of communi-

[2] We note, however, that it is, of course, impossible to arrive at an organisation that is not contentious, and merely seek to indicate the diverse contribution to this area.

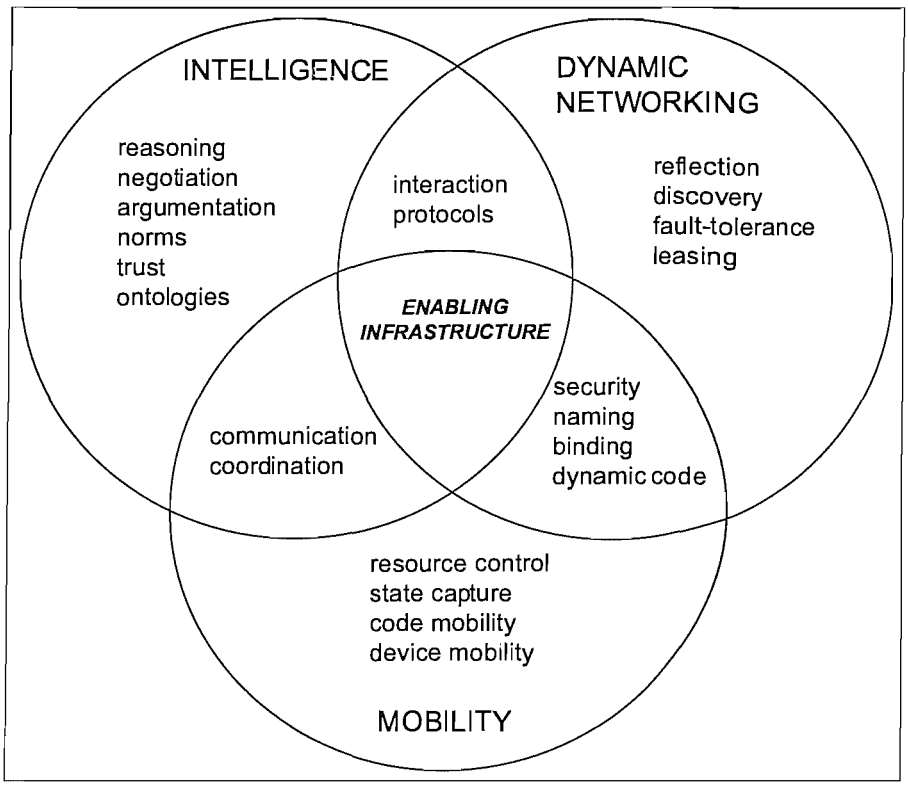

FIGURE 1.1: Required capabilities for enabling infrastructures

ties of networked devices. Since we need to deal with environments in which devices and users may come and go at any time, it is essential first to provide mechanisms that allow devices to join network communities automatically, and second to develop techniques to administer their access to other devices within that community.

- *Mobility* refers both to code and to user mobility. Code mobility aims to improve resource utilisation by moving code closer to the information it must process. Device mobility, enabled by wireless network technologies, allows users to access network resources while on the move. Although certain aspects of the problem are different, they share in common the need to address access control, authentication, security and privacy.

- *Intelligence* refers to attempts to automate complex processes. In this respect, providing mechanisms for programs to reason about their environment, and negotiate with other parts of the system at a higher semantic level, is paramount if applications are to be able to adjust to changes and pursue tasks without guidance by users or detailed control by network administrators.

A number of technologies have been developed to address some of these issues, especially

in relationship to *dynamic networking* and *mobility*. They range from new internet protocols (IPv6 [70], IPsec [131]), technologies enabling dynamic networks (Jini [5], UPnP [64]) and dynamic service discovery (Web Services [220], UDDI [23]) to higher-level standardisation efforts (RDF [85], Semantic Web [25], OWL-S [4]). Combined, these technologies can form a formidable toolkit, providing many of the pieces required for creating adaptive distributed systems.

At the *technological* level, therefore, some degree of integration has been achieved (although clearly more is needed). At the *conceptual* level, however, the issues relevant to dynamic networking, mobility and intelligence draw on diverse and distinct areas of research and development, and some overarching framework is necessary. For developers, this *conceptual* framework can make explicit the interconnections between the technologies described above. This is important, for without it there is likely to be a redundancy of approach, with technologies and applications in one subfield being reproduced (in very different terms, and with different solutions) in another. In the worst case, this can also lead to poorly-designed systems that are then hard to maintain and upgrade.

In our view, agent-based computing is well suited to provide this conceptual grounding. In the next section we examine the argument for its suitability and identify some of the challenges, both in general terms as well as with specific relevance to heterogeneous computing environments.

1.4 Agent-Based Computing

In Section 1.2, we mentioned several examples of agent-based applications that have already been developed in an attempt to tackle the problems posed by the emerging computing environments. In part, this is proof of the *appeal* of the paradigm as a basis for systems involving the full range of concepts described in Figure 1.1. As Jennings [120] argues, agent-based approaches offer several qualitative advantages over conventional approaches for dealing with complex systems.[3] The main abstraction offered by this paradigm is that of an *agent* as an encapsulated computer system, situated in an environment and capable of independent problem-solving action [120]. It is largely agreed that the predominant distinguishing characteristics of intelligent agents are autonomy (agents having some form of control over their actions), social ability (the ability to interact with other agents or humans), pro-activity (goal-directed behaviour) and reac-

[3] Nevertheless, Jennings also acknowledges that there is no *quantitative* data to back the claim.

tivity (the ability to sense and react to the environment) [232]. Large, distributed systems can then be decomposed in terms of *interacting* agents, leading to *multi-agent* systems. The aim is to make as few assumptions as possible about the state of the environment in which agents will operate, and enable agents to *dynamically* interact with other agents as needs arise. As such, the organisation and relationships between individual components of the system can much better reflect the true nature of dynamic, heterogeneous environments, and are thus better able to cope in them.

Despite its apparent suitability and limited adoption, agent-based computing has yet to mature as a mainstream paradigm on a par with object-oriented computing, which is now almost universally accepted as the *de-facto* paradigm for software development. The reasons for this are multi-faceted, ranging from issues directly related to the maturity of the technology to issues relating to the difficulty of organisations changing their established development processes. In the next subsection we discuss some of the general challenges facing agent researchers as they have been identified in the relevant literature.

1.4.1 Challenges for Agent-Based Development

A central problem for agent development is the lack of clarity in defining the exact relationship between agent systems and other software paradigms, most importantly object-oriented development [231]. This makes it harder for developers to understand exactly what agent development brings to their application development *toolbox* that could not already be achieved through a purely object-oriented approach. Coupled to this problem is the lack of a well-defined agent methodology. Although there are a number of methodologies available [138], none have managed to be adopted in a convincing fashion by industry. The two issues are interrelated because, for a convincing methodology, and a satisfying account of the difference between agent-based development and other approaches, there should be some agreement on the basic building blocks of agent development [231]. Sabater et al [189] and Lind [136] also identify this as a problem and argue that more conceptual frameworks are required that directly support the practical development of agent systems.

Wooldridge and Ciancarini argue that this lack of consensus may be further complicated by the use of UML [185] as a modelling language that was intended for object-oriented systems. Interestingly, recent developments within the agent community related to the FIPA standards body[4],

[4] http://www.fipa.org/

have led to more work on the use of UML for the design of agent systems[5], while related work is examining a variety of alternative notations[6] as well as meta-modelling notations[7]. However, such work is in its early stages and there is still a lively debate about the suitability of any single approach.

Both from the above observations and from similar observations of Bradshaw et al. [32] and, more recently, Winikoff et al. [227], it appears that one of the key problems is a lack of clarity relating to how *basic concepts* are understood and realised in practical agent systems, through the development process and in the systems themselves. By basic concepts, we mean the concepts that underpin all agent systems, such as how agents and the relationships between them are modelled at an abstract level. Such models are necessary for understanding the relationships between agents and objects, for understanding the underlying principles of agent-based computing, for providing methodologies and routes to implementation, and for offering well-founded development toolkits.

Although there are several examples of agent-based applications, and several underlying models for use in agent systems, there is a general lack of connection between the two, since applications are typically developed arbitrarily, providing mostly instance-specific knowledge. Without such links it becomes difficult to establish *reusable* models of agent systems, since there is no common foundation. Crucially, reusable models can act as a strong point of reference for the wider range of agent research, allowing developments in areas such as negotiation, coordination and intelligent reasoning to be built on top of them.

1.4.2 New Challenges

In addition to the problems described above, there is also a set of challenges relating to the development of agent-based systems in heterogeneous environments. These challenges are of a more practical nature, since they deal with issues of direct concern to any application development effort in such environments, irrespective of whether it adopts an agent approach.

Dealing with such practical challenges is one of the key motivating principles of our work. In development environments in which the culture of rapid application development is overpowering, abstract models are often seen as a hindering rather than facilitating influence. Thus, while

[5] http://www.auml.org/
[6] http://www.pa.icar.cnr.it/ cossentino/FIPAmeth/
[7] http://www.fipa.org/activities/modeling.html

we do propose such abstract models for agent systems, we recognise that we must also provide a clear path from the models to implementation in a coherent and structured fashion. In particular, we aim to develop models that address the following issues, which we consider to be central in this context.

- The wide variety of application domains, and the heterogeneity of operating platforms within any single application, creates the need for constructing a variety of different types of agents, each reflecting the particular application needs and operating environment constraints. In order to deal with this variety, application developers are forced to adopt several approaches at the same time. For example, integrating agent development for both mobile and desktop devices is not currently supported at the conceptual level, even though it is technologically feasible. Mobile devices are not able to support continuous operation, and have limited computational power, which makes the ability to save state information on the device, for later use to resume operation, a challenge. The design of agents for such devices must deal directly with such challenges.
- Dynamic environments and changing user demands create a need for applications to be able to easily adapt. This refers both to the need for reconfiguring individual agents and to the manner in which a multi-system as a whole operates. The ability to reconfigure agents at run-time should not remain just a possibility at an implementation level, but should also be supported at a conceptual level.
- The large number of devices and their diverse capabilities will inevitably lead to several ways in which agents could cooperate to solve common problems as well as to several areas where conflicts may occur as each agent attempts to achieve their own goals. The challenging aspect is effectively *identifying* such opportunities or conflicts as the system develops, and *applying management* to ensure that conflicts are avoided or opportunities for cooperation are exploited.

1.5 Research Aims

The shortcomings described above cover a wide range of issues, from the construction of individual agents to the way in which relationships between agents are modeled and understood. The unifying thread between them is that they can be characterised as issues relating to the *infrastructure*, i.e. the basic building blocks, required to develop multi-agent systems. By this we

mean that they are not problems that occur only within a specific application, but are weaknesses inherent in many existing models of multi-agent systems that are developed in *ad hoc* ways. We argue that future development of agent systems can only progress the state-of-the-art if it builds on current work and is supported by a technical infrastructure that corresponds to principled theoretical models (or *conceptual infrastructure*). These theoretical models should provide the necessary abstractions to support agent-based systems development, as well as explicating the relationships between models of individual agents and models of interaction between agents.

Such conceptual infrastructure should be based on two overarching principles. Firstly, the models developed should be applicable across a range of domains and the resulting artifacts should be *reusable* across application and domains. Not only does this enable alternative solutions to be described and contrasted through a common set of concepts, but it also benefits development, since experience gained during application development for one project can be transferred to other projects. Secondly, key to all models of multi-agent systems, and underpinning them is support for *cooperation*. The conceptual infrastructure should support the development of mechanisms for cooperation between agents through models that enable them to describe and analyse the interactions between agents while taking into account the dynamism and heterogeneity of the environment. Given these overarching principles, this thesis aims to achieve the following distinct and clear goals.

1. To provide conceptual infrastructure for building agent systems that is suitable for use in the conceptual elaboration and design of agent systems, and in the technical infrastructure for construction of agent systems. This facilitates both reasoning about agent systems and agent systems development. We can divide this aim into two parts.

   (a) To provide abstract models of agents that allow us to capture the wide range of different types of agents that exist in heterogenous environments. The ability to capture the *entire* range is important because it means that the theoretical models are sufficient to model applications, ensuring they will not become irrelevant to development.

   (b) To develop a *technical* framework (which we might also consider to be a development or construction toolkit) that provides a clear illustration of how the conceptual framework can find practical implementation.[8] This technical framework should sat-

[8] We clarify that our aim is not to develop a methodology for agent-system development but simply illustrate through a practical implementation how the abstract concepts introduced can be implemented.

isfy the following aims.

i. It should make use of the abstract models, and make explicit the links between the abstract models and their use within the technical framework.

ii. Similarly to the abstract models, it should provide a unifying way of specifying a range of agents architectures to suit the different domains and applications.

2. Cooperation involves the participation of multiple agents in achieving some overarching objectives, and there are already many mechanisms to facilitate this. However, hand-in-hand with facilitating cooperation is the need to *control* agent behaviour to prevent undesirable interactions. Although there is a variety of mechanisms for achieving this, they typically focus on either just the technical means or on restricted cases where some strong assumptions about the types of agents can be made. They do not provide an analysis to determine *when* such cooperation or interference is likely given the current state of the environment and the agents within it. Thus, we aim to develop tools to enable the analysis of potentially cooperative or undesirable situations, and of the possible configurations that might achieve the desired results. This can also be divided into two parts, as follows.

   (a) To develop a model for interactions between agents and relate such a model to the *goals* of individual agents so as to be able to reason about relationships and how cooperative interactions or undesirable interactions may be identified.

   (b) To use this model, to provide a complete characterisation of the possible relationships between agents that can act as a guide for enabling:

      i. system design and analysis by designers;

      ii. and run-time identification of, and reasoning about, relationships between agents so that agents can better adjust their behaviour to deal with changing conditions without external intervention.

3. As already mentioned, a central concern is that there must be clear links between abstract models and their practical implementation. Thus, throughout the thesis we aim to provide examples of the use of the models to solve practical problems so as to illustrate the utility these models have to offer application development in general.

1.6 Research Approach

In order to meet our aims, it is important to establish an approach to the task at hand that will ensure, as far as is feasible, that the results remain relevant. In this respect, we recognise that there are several possible paths to follow in devising models for agent infrastructures. For example, current implementations of applications could be studied and common patterns identified, or a theory could be developed from scratch or adopted from existing work and then refined following its application to real world problems.

In essence, our approach is a combination of those above. We begin with the belief that creating new models from scratch, without basing them on any existing work, is probably counter-intuitive since agent research has reached a certain level of maturity, and ignoring existing work would simply add to the current proliferation of *alternative* models. Rather, we aim to adopt an existing approach, and refine it by identifying the points at which it does not address our needs as outlined above. The criteria to determine which approach to choose will be based on how closely it can be aligned with our aims. At the same time, the manner in which we proceed to introduce new concepts or refine existing ones is informed by the experience gained through the numerous agent applications and toolkits that are available (several of which are reviewed in Chapter 2). Finally, the resulting models are themselves implemented to demonstrate how one can arrive from the abstract models at their actual implementation.

An overarching guideline for our work is that any attempt to provide abstract models of multi-agent systems must strike a careful balance between providing sufficiently practical models to aid implementation without, however, closing possible avenues in terms of agent architectures, communication and coordination mechanisms.

1.7 Thesis Overview

Our research is presented in 7 chapters, including this one, organised as follows. In Chapter 2 we review existing work that can contribute to the discussion of appropriate models to support agent-based development. The review ranges from an examination of architectures for individual agents, to issues central to multi-agent systems, such as models of interaction, discovery, and regulation.

A detailed analysis of the SMART framework is provided in Chapter 3, since it underpins all

subsequent work in the thesis. We identify SMART's shortcomings in relation to our aims, and outline how we extend and refine it to address them. As such, Chapter 3 sets the scene for the rest of the thesis.

Construction of individual agents is dealt with in Chapter 4. We present an agent construction model (*act*SMART, which is based on the abstract agent model of SMART), and justify our design choices through a discussion of the criteria we believe any agent construction model should fulfill in order to address our stated aims. In addition, we provide examples of the use of *act*SMART through the specification of an architecture for an agent participating in auctions, and generic architectures for negotiation and argumentation agents.

The issue of relationship identification and characterisation is examined in Chapter 5, providing the underpinning for supporting cooperation in multi-agent systems. We present a model of agent interaction and use it to derive a typology that describes all possible relationships between two agents. Furthermore, we provide a typology of goals in relation to an agent's capabilities and explain how that can be used to further enhance relationship analysis.

The work of Chapters 4 and 5 is evaluated in Chapter 6 through the development of an application based on a ubiquitous computing scenario. In order to support agent operation within such an application we develop architectures for middle agents performing capability brokering, as well as relationship analysis agents that make use of the tools of Chapter 5 to support cooperation between agents. We also provide examples of how *act*SMART enables architectures to adapt to their operating environment. Some of the details of the architecture specification developed in this context are provided in Appendix A.

Finally, Chapter 7 provides a summary of the work and our conclusions, outlines our key contributions, discusses the limitations of the work and its potential to underpin further research.

Chapter 2

Models for Agent Infrastructures

"The determination of shared paradigms is not, however,
the determination of shared rules."
Thomas S. Kuhn (1922-1996); Science philosopher and historian

2.1 Introduction

In this chapter we identify, review and relate work that can inform our stated aims of providing reusable models for individual agent construction and for identifying and reasoning about relationships between agents. In order to achieve this, the review spans several fields that are traditionally viewed separately. This is an inevitable side-effect of our attempt to provide a broad foundation for the subject of agent-based system construction that ranges, in breadth, from individual agent architectures to models of agent interaction and, in depth, from abstract theoretical concepts to practical implementations.

As a result, our first task is to identify a suitable schema that connects the various issues so as to impose some order in the method through which we proceed with the review. The task is not trivial as the richness and variety of agent research often defies clear categorisations [137]. We recognise, therefore, that any classification is significant as far as it fulfills the purpose of ordering the *presentation* of work, rather than identifying *true distinctions* between different types of research. The schema we introduce attempts to relate issues both in breadth and in depth. The former is achieved by adopting a division of issues as suggested by Zambonelli et al. [236], who propose a tripartite division between *intra-agent*, *inter-agent* and *organisational*

structures. The latter is done by clearly distinguishing between research that is based on abstract models and research that is based on empirical experience through practical implementation.

With the schema in place, we then examine each aspect in turn and relate them, both within each division, as well as between divisions. We conclude by identifying the missing links between the different strands of research and discuss how our own work attempts to address some of these shortcomings.

2.2 Review Schema

As mentioned in Chapter 1, we view agent infrastructure as providing the basic building blocks required to enable an operational multi-agent system in a heterogeneous and dynamic computing environment. These building blocks refer both to abstract concepts, and to practical implementations in existing computing environments. This section provides a classification of the various research issues that we view as relevant to providing such agent infrastructure within the limits of the aims of this thesis.

We begin by defining the breadth of research issues through Zambonelli et al.'s [236] division of issues into three related constituents, as described below, and illustrated in Figure 2.1.

Intra-agent At the intra-agent level the focus is on individual agents and their structure. We investigate the variety of proposals of how to characterise and construct agents, which range from specific agent architectures, such as the Belief-Desire-Intention architecture, to proposals relating to a more generalised understanding of agents, such as Luck and d'Inverno's SMART framework [82]. Both of these aspects inform the task of establishing the appropriate infrastructure for agent systems at this level.

Inter-agent At the inter-agent level we examine models of interactions between agents and how they can facilitate reasoning about relationships. We do not focus on specific communication or coordination mechanisms, as these go beyond the remit of this thesis and are issues that necessarily must be based on an underlying model of interaction, which should enable developers to reason about what are the most appropriate coordination mechanisms. In addition, we also consider the discovery of agents within dynamic environments, as it forms a fundamental aspect of agent infrastructure for dynamic and heterogeneous environments.

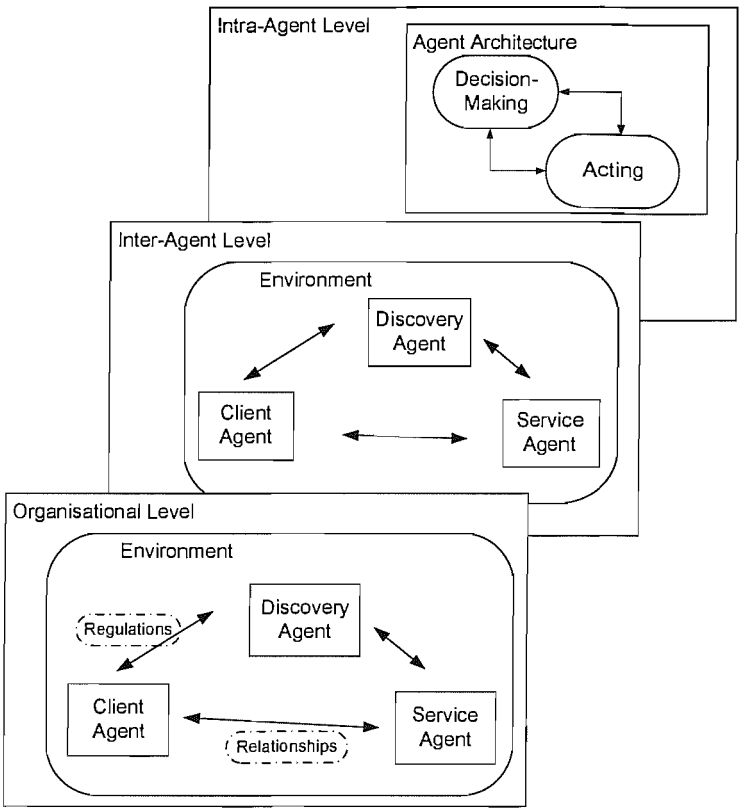

FIGURE 2.1: Intra-agent, inter-agent and organisational levels

Organisational structures Finally, at the organisational level we review work that enables the definition of appropriate structures for the control of agent systems beyond individual agent architectures and the emerging agent interactions. These issues cover work on policies, institutions and norms.

As indicated, the research contributions to these different areas range from abstract frameworks to practical tools. We distinguish abstract from practical approaches by the degree to which they are tied to, and arise from, the implementation of a particular system. Often the distinction is clear by the presentation of the work, where concepts from a practical implementation are described through a presentation of the implementation, while more abstract work is presented in isolation and, typically, with the aid of a formal mathematical language. Of course, there are some examples of work which offer both a formal presentation of the models used as well as a practical implementation.

We primarily examine research that falls under the broad headings of agent theories, agent methodologies and agent toolkits attempting to extract from each the issues highlighted above. Furthermore, with direct reference to the organisational level we review work on institutions,

norms and policies, which deals with the development of appropriate regulatory frameworks for agent-based systems.

Agent theory is considered to be anything that provides a conceptual model of agents and their operation that is clearly divorced from specific implementation technologies, such as programming languages, operating systems, and networking protocols. Relevant examples of such work are BDI-based agent models [177] or the SMART framework [81]. We consider agent methodologies as dealing with formalising the process of moving from a problem specification to the design of a solution. Now, in order to do this an agent methodology must either adopt or define some notions of agents and their operations, since these are the first order components of the design. It is this aspect of agent methodologies that we investigate. Relevant examples are the Gaia methodology [233] or DESIRE [37]. Finally, toolkits are essentially software designed to accelerate the task of developing agent-based systems by providing a large portion of the required lower-level infrastructural support, and in recent years, there has been significant work in this direction. [1] Although often not explicitly defined, most toolkits encompass some conceptual model relating to agent operation. As such, they are particularly relevant to our research since, by virtue of their purpose, they usually tackle a large portion of the issues we have highlighted in the introductory chapter as relevant to our research.

2.3 Intra-Agent Issues: Models of agents

In this section, we review work that can provide a suitable basis for the description and construction of single agents. We consider both abstract approaches and specific agent architectures, ranging from reactive to deliberative. We begin with a review of the work of Russel and Norvig [186] and Luck and d'Inverno [82], both of which take a broader view of the problem of modelling agents. Subsequently, we look at the BDI model and its various implementations, as it is one of the most influential approaches in agent design and a good example of deliberative approaches to agent architecture. Then, Brook's Subsumption architecture provides an example of a reactive architecture, while TouringMachines [92] and INTERRRAP [96] provide examples of hybrid architectures. Subsequently, we review the work of Sabater et al. [189] on engineering agents, which makes use of multi-context logics, and DESIRE [37] as examples of component-based approaches to agent architecture design. Finally, we examine architectures from agent toolkits, so as to gain an understanding of the state of the art at the level of im-

[1] Agentlink.org lists, at least, 100 different toolkits.

Architecturally-Neutral	Deliberative	Reactive	Hybrid
Rusell&Norvig SMART Sabater et al. DESIRE JADE	IRMA PRS dMARS AgentSpeak(L) ZEUS RETSINA IMPACT	Subsumption Architecture	TouringMachines InterRRaP

FIGURE 2.2: Schema of agent models reviewed

plemented systems. The various models reviewed can be categorised as shown in Figure 2.2. They are distinguished along the lines of architecturally-neutral models (which do not constrain development to specific architecture types), deliberative architectures (which make use of and reason over explicit representations of the environment), reactive architectures (which do not depend on explicit representations of the two), and hybrid architectures (which make use of both reactive and deliberative approaches).

2.3.1 Russel and Norvig

In the textbook *Artificial Intelligence: A Modern Approach*, Russel and Norvig describe agents as "anything that can be viewed as **perceiving** its environment through **sensors** and **acting** on that environment through **effectors**" [186]. They also differentiate between autonomous and non-autonomous agents by stating that if an agent acts without paying attention to its percepts, then it lacks autonomy [186]. The justification behind this form of distinction is based on the understanding that if an agent's actions are completely pre-determined by what Russel and Norvig call "build-in knowledge", then there is no space for a deviation from the instructions of the designer. Russel and Norvig view this as transferring the *ownership of intelligence* to the designer and not to the agent itself. Although such a distinction is appealing in its simplicity, it is also naive in its conception, since based on the same rationale one could say that the way an agent interprets environmental information depends on the rules placed by the designer within the agent, therefore enabling one to say that the intelligence *always* lies with the designer.

In any case, Russel and Norvig go on to state that the task of building useful agents is seen as synonymous to building *rational* agents, where rationality is described as the attempt to "do the right thing" [186]. In order to evaluate the extend to which the "right thing" has been done, they introduce the notion of a *performance measure* as the indicator of an agent's success in achieving a task.

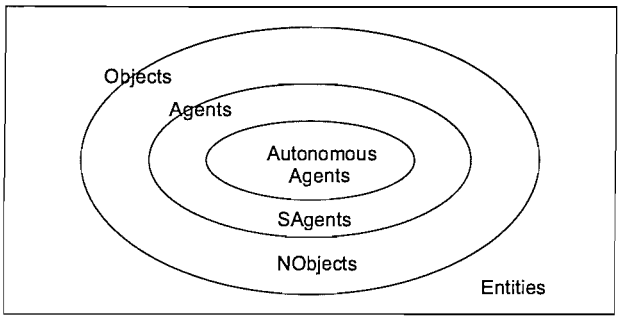

FIGURE 2.3: The entity hierarchy

With these basic principles in place, Russel and Norving, continue to define different types of agents, described below.

- *Reflex agents* are those that simply map stimuli to responses.
- *Agents with state* extend reflex agents by keeping a record of their actions.
- *Goal-based agents* act towards achieving a desirable situation by reasoning about the state of the world and attempting to determine which actions will change that state to bring it closer to the desired state.
- Finally, *utility-based agents* are able to determine which of a possible set of desirable goals would be *more* desirable based on a performance measure.

Based on this model, Russel and Norvig map out the relationships between different aspects of artificial intelligence research (planning, knowledge, learning and communicating) and agent-based systems. In essence, agents are viewed as the software engineering paradigm for the application of artificial intelligence techniques.

2.3.2 SMART

Luck and d'Inverno [82] propose a conceptual model of agent-based systems that lies at a high level of abstraction, making very few assumptions about the internal structure of agents, preferring to focus on providing means for describing different models within a common framework.

SMART (Structured, Modular Agent Relationships and Types) provides an encompassing structure that clearly differentiates between agent and non-agent entities in the environment, and

specifies agents in a compositional way. In essence, the framework proposes a four-tiered hierarchy that includes the generic and abstract notion of an *entity* from which *objects*, *agents* and *autonomous agents* are, in turn, derived. In Figure 3.3, the Venn diagram describes the different levels in the hierarchy, and outlines the ways in which they are related. Though we will not offer a detailed exposition of the framework, we review the key concepts below.

Entities are defined in terms of sets of attributes, where attributes are describable features of the environment. Objects are then simply entities with sets of capabilities, where capabilities are actions that objects can perform to change the state of the environment. In turn, agents are objects with sets of goals, where goals are defined as desirable environmental states, and autonomous agents are those agents able to generate their own goals through the motivations that drive them. Here, motivations can be regarded as preferences or desires of an autonomous agent that cause it to produce goals and execute plans in an attempt to satisfy those desires.

In addition to these basic levels, and in order to further explicate the consequences of their framework, Luck and d'Inverno introduce two additional refinements: *neutral objects* (Nobjects) are objects that are not agents, and *server agents* (SAgents) are agents that are not autonomous [139]. The relationship between neutral objects and server agents is complementary, since neutral objects give rise to server agents when they are ascribed goals by other agents in the environment. Once these goals are achieved or they are no longer feasible, server agents revert back to neutral objects.

Luck and d'Inverno have applied this model in a concerted effort to translate alternative approaches to a common set of ideas [75, 79, 76, 77], giving a strong indication of the value of a common high-level approach as a means of consolidating different strands of work within agent research.

2.3.3 Belief-Desire-Intention

The Belief-Desire-Intention (BDI) model has found widespread acceptance within the agent research community, and its adoption and use by a large number of researchers make it the most widely studied model for agent architecture design. The approach underpinning the BDI model is based on the explicit representation of an agent's beliefs (knowledge about the world), desires (what it would like to achieve), and intentions (what it will try to achieve) as data structures that determine the operation of the agent. These data structures form the core around which the BDI

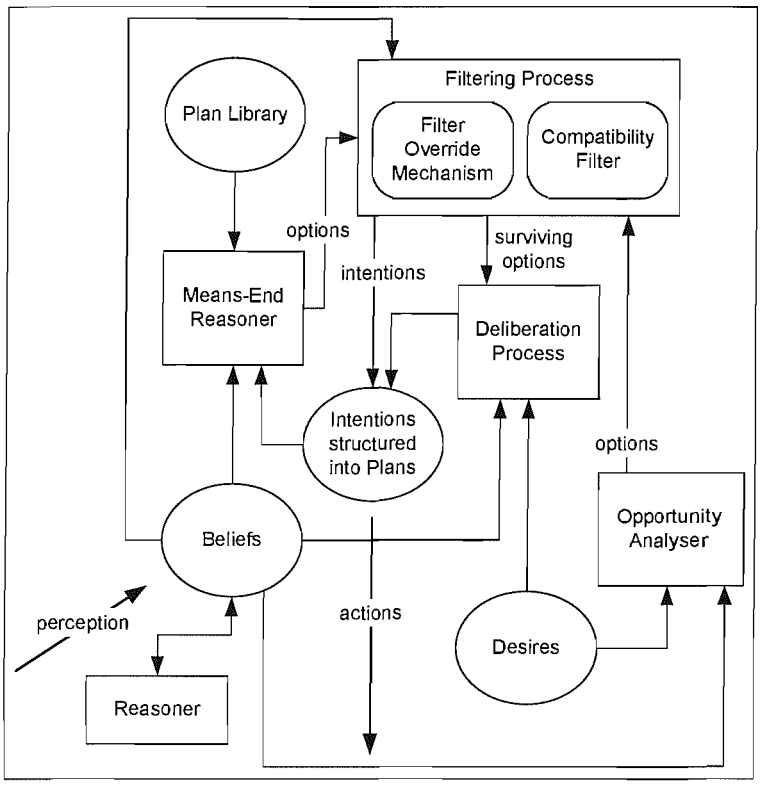

FIGURE 2.4: The IRMA agent architecture. (Based on [33])

architecture can be developed. The overarching goal of a BDI architecture can be summarised as connecting the data structures described above to appropriate decision-making algorithms that will determine which desires the agent will choose to achieve, transforming those desires into intentions, based on its beliefs. A BDI agent will attempt to achieve intentions until they are satisfied or they are no longer achievable [57]. Much work has gone into developing practical systems and formal models for BDI agents and in the following sections we briefly look at some of the most influential work in this area.

IRMA

Bratman et al. developed one of the first sophisticated examples of a BDI architecture, drawing from their work on intentions, plans and practical reasoning [33]. Their IRMA architecture (Intelligent Resource-bounded Machine Architecture), primarily deals with situations where agents may not always have the resources to make optimal decisions, and have to be able to choose sub-optimal ones. An overview of the IRMA architecture is shown in Figure 2.4. The operational cycle begins with perception of the environment and the update of *Beliefs*. The *Opportunity Analyser* then checks whether, based on the updated beliefs, any goals have been achieved,

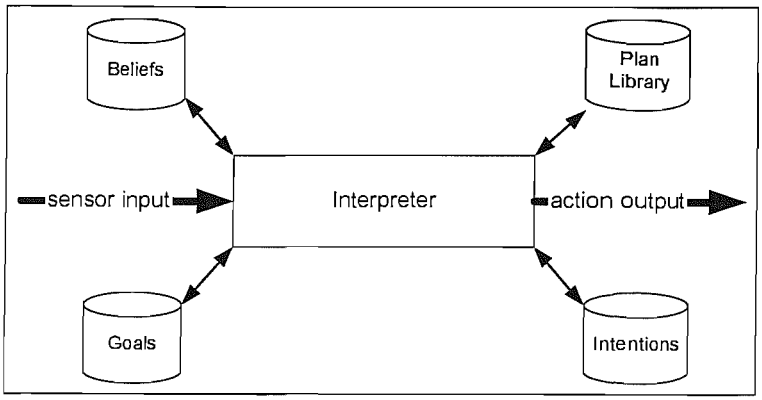

FIGURE 2.5: The PRS agent architecture

whether it can still pursue existing intentions, or whether alternative plans can be proposed. The *Means-End Reasoner*, uses the *Plan Library* and the current beliefs to evaluate what plans can be adopted. Now, plans from both the *Means-End Reasoner* and the *Opportunity Analyser* pass through a filtering process which is made up of the *Compatibility Filter* and the *Filter Override Mechanism*. The former, checks whether any new plans are consistent with the existing intentions, while the latter is used in circumstances where although plans are not compatible they may fulfil certain properties that may necessitate that they do go to the *Deliberation Process*. The *Deliberation Process* determines how the current intention structure is affected by plans that have gone through the filtering process. Finally, the *intentions structured into plans* are what the agent should attempt to achieve through actions.

The IRMA architecture provides several significant lessons in architecture design for agents. It illustrates the importance of ascertaining, through examination of beliefs, which intentions remain valid at each operational cycle and that certain overriding mechanisms, if chosen appropriately, can be very effective for agents in dynamic environments. The architecture was demonstrated and tested in a Tileworld simulation environment [168].

PRS, dMARS and AgentSpeak(L)

Rao and Georgeff, have developed extensive formal models for BDI agents [176, 177, 178], based on intention logics, as well as gained experience from practical implementation of BDI architectures such as the Procedural Reasoning System [102] (PRS). The PRS system, illustrated in Figure 2.5, is based around beliefs, goals, plans and intentions. Goals correspond to the agent's desires. Plans prescribe courses of action that an agent can follow in order to achieve its intentions. Plans have *triggers* or *invocation conditions* that stipulate which beliefs and/or goals

must be active for a plan to be relevant. Furthermore, plans have a *context* that states what the agent must believe about the world (as stated within an agent's beliefs) before a plan becomes applicable. Finally, the body of the plan prescribes which actions the agent should take. For example, a belief of *hungry*, may trigger a plan *make sandwich*, which is applicable under the conditions *have toast and butter* with a body indicating the action *spread butter on toast*.

This work led to the more sophisticated dMARS [101] (distributed Multi-Agent Reasoning System) implementation, which was used in a number of multi-agent systems applications [101]. D'Inverno et al. also provide a formal operational semantics for the dMARS systems, using the SMART framework [75].

Rao, went on to define AgentSpeak(L) [175], a programming language for BDI agents that aims to provide an operational and proof-theoretic semantics for a language that can be viewed as an abstraction of an implemented BDI system. AgentSpeak(L), is based on the experience of PRS and dMARS and was born out of a wish to provide a model for BDI agents that would unite theory and practice in the field. D'Inverno and Luck also provide a formalisation of this work, based on the SMART framework in [79]. Furthermore, Bordini et al. provide extensions to AgentSpeak(L), in order to enable the language to better deal with task scheduling in [29].

Other BDI-related work

Several other research efforts have dealt with various aspects of agent-based systems based on the BDI paradigm. In [57], Cohen and Levesque, develop a theory of intention which acts as a departure point for many BDI systems. Singh expands on that work, criticising Cohen and Levesque in [198], and developing his own approach on the subject in [199]. Jennings, in [118], provides the specification and implementation of a BDI agent system. Shoham, in [194], presents the AGENT0 programming language, which is one of the first agent programming languages and geared towards BDI agents. Wooldridge, in [230], presents a testbed for experimenting with agent-based systems, where agents are based on a BDI-paradigm, adopted from Shoham, and formally defined. Other systems for BDI agents are the JAM system [117] and the commercial JACK system [116, 44]. Finally, Kinny et al. suggest a methodology for BDI systems in [132] as do Padgham and Winikoff [163].

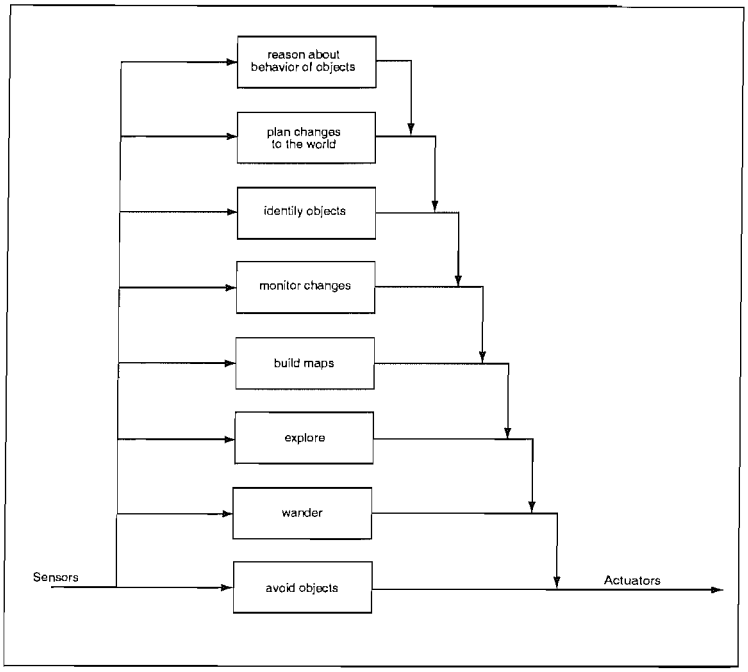

FIGURE 2.6: The Subsumption architecture

2.3.4 Subsumption Architecture

The *Subsumption* architecture, was proposed by Brooks [40], in an attempt to provide an agent architecture which did not depend on the explicit representation of the environment within the agent architecture. Deliberative approaches are criticised by Brooks [41] as not effective when having to deal with truly dynamic and complex environments, where the attempt to explicitly represent all the relevant information about the world within the agent architecture becomes an intractable task. Furthermore, based on his experience of development of mobile robots that need to react at real-time, Brooks identified three central requirements for agents [40].

1. Agents should be able to cope with multiple goals at the same time. Clearly, in a mobile robot scenario, where robots may need to navigate through a complex environment while trying to carry an object to a specific destination or while still being able to receive new instructions, the ability to deal with multiple, possibly conflicting goals is paramount.

2. Agents should have multiple sensors to be able to receive the various kinds of information the environment provides. In a mobile robot setting these could be infra-red cameras, acoustic sensors, and so forth. In purely software-based setting this may be the ability to receive messages from other agents, sensors for measuring network bandwidth, available disk-space, and so forth.

3. Agents should be robust. When relatively unexpected events take place, such as the failure of components or dramatic changes in the environment, the agent behaviour should degrade gracefully rather than bring about an abrupt general failure of the agent.

As an alternative, Brook's suggests that we consider agents as embodied entities whose behaviour is affected by the interactions of a series of, relatively, simple control layers, each one independent of the other and each one focusing on a specific task. The layers operate concurrently and all have access to sensor information and actuators. Layers can access information from layers below them and can suppress the action of layers below them, but are unaware of layers above them.

An example architecture using this approach is illustrated in Figure 2.6. Here, the different capabilities of a robot are decomposed into the layered structure, with more basic capabilities such as *avoid objects* or *wander* at the bottom of the layer stack, and more complex ones such as *reason about behaviour of objects* and *plan changes to the world* at the top.

Starting from this basis Brooks and his team at the MIT Artificial Intelligence lab have developed a series of robots that prove the claim that intelligent behaviour can exist without the use of internal representation [3]. One of the significant benefits of the approach is that each layer can be tested in isolation to the others, ensuring that it works appropriately before placing it within a more complex system. The agents produced are also robust since if complex layers fail, the more simple basic layers will still enable the robot to function. Nevertheless, this work has also shown that for certain types of actions, some level of internal representation is essential. For example, it is hard to see how long-term goal directed behaviour in a social environment, in which interaction with other agents can only take place through direct communication, can be achieved without some internal knowledge [66]. Furthermore, systems based on the subsumption architecture tend to be quite complex, and there are no clear guidelines of how layers should be stacked and the interaction between them managed.

2.3.5 Hybrid Architectures

The limitations of both deliberative and reactive approaches have led to the development of models that incorporate features from both, leading to hybrid designs. The basic tenants are very similar since agents are still considered as goal-directed entities that interact with the environment via sensors and effectors. The innovation lies at the architecture of these agents,

where both reactive and deliberative elements are present. We examine two systems that act as exemplars for the alternative approaches to hybrid architectures.

TouringMachines

TouringMachines was proposed by Ferguson [92], as an architecture for autonomous agents situated in dynamic environments. Ferguson recognises that agents in such environments must be able to effectively monitor both *expected* and *unexpected changes*, and be able to deal with them in the short-term as well as reason about how they will affect long-term goals.

As a response to these concerns, Ferguson proposes the TouringMachines architecture. It has three main layers, each handling one significant aspect of agent action. The *reactive* layer represents the agent's direct responses to stimuli, the *planning* layer handles the generation and execution of plans while the *modelling* layer handles higher level societal aspects such as modelling other agents in the environment. These layers all have access to sensors and actuators and can communicate with each other. In addition, there is a set of context-sensitive rules imposed on all layers in order to solve conflicts between decisions of different layers.

INTERRRAP

INTERRRAP [96] takes a similar approach to TouringMachines by mixing deliberative and reactive components, but while TouringMachines is a *horizontally* layered architecture, since every layer can communicate with every other, INTERRRAP is a *vertically* layered architecture more similar to Brooks's Subsumption architecture. There are three levels of control starting from the behaviour layer, moving up to the plan-based component and finally a cooperation component. Each level communicates with the one above and below it, while each level also has access to a specific knowledge base (KB). The world model KB represents knowledge about the agent's specific situation and corresponds with the behaviour layer, the planning KB represents goals and plans communicating with the planning layer while the cooperation KB represents the social knowledge, such as joint plans, and communicates with the cooperation layer. Actions are accessed via the behaviour layer while sensor information filters up from the world model KB.

2.3.6 Sabater et al.

Sabater et al. [189] have developed a model for constructing agent architectures in response to what they identify as a lack of proposals of methodologies that relate designs to agent architectures and their practical implementation. They attempt to address this problem within the context of logic-based agents that make use of multi-context logic reasoning [104].

Their model allows for the modular composition of agent architectures, where each *unit* (or *component*) may use a different logic to encode the problem-solving knowledge of the agent. Logics in this sense are defined as declarative languages, each with a set of axioms and rules of inference. Units are connected by *bridge rules* that translate one logic-based representation to another. Units and bridge rules can be grouped together to form *modules* which provide a further level of abstraction that makes it easier to handle large numbers of units, and modules can also be connected via bridge rules. Modules communicate by multicasting bridge rules along a communication bus.

This basic model is further refined with two *control elements* which are associated to bridge rules. The *consuming* element causes a rule to 'move' between units. This enables the modeling of changing state between units. The *time-out* element indicates that there is a delay between the instant where the conditions of a bridge rule are satisfied and the activation of the rule. The justification for this control element is that it increases the expressiveness of the construction model since it allows for rules not to be acted upon if the formulas are removed before the time-outs.

The Sabater et al. construction model, through its modular approach and simplistic framework that enables modules to communicate goes someway towards providing a viable generic agent construction model.

2.3.7 DESIRE

DESIRE (DEsign and Specification of Interacting Reasoning Components) [35, 36, 37] aims to aid in the specification of complex software systems when viewed as interacting components. The process of system design involves task specification, identification of what information is exchanged, sequencing of tasks and definition of appropriate knowledge structures.

Agent architectures are composed as sets of interacting processes that are expressed through

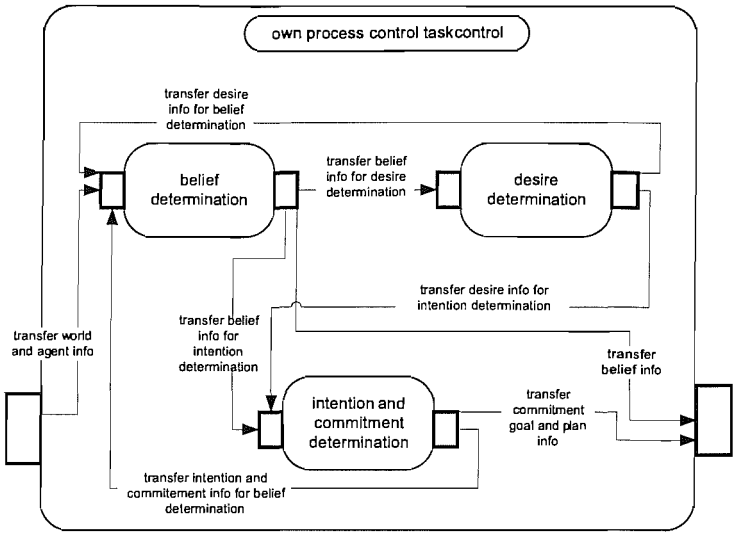

FIGURE 2.7: Generic BDI agent in DESIRE (taken from [36])

components. For each process the designer must identify the types of information used as input and resulting as output, which define the input and output *interfaces* of the components. Processes can be modeled at different abstraction levels and are implemented as *abstraction/specialisation* relations between components, leading to components being composed of other components. The most primitive types of components are considered to be components performing tasks such as calculation, information retrieval, and so forth. Processes can then be composed to form an architecture, where composition is described by the relationships between components, the possibilities of information exchange between processes and the task control knowledge used to control processes and information exchange.

Using this technique Brazier et al. then go on to define a number of possible agent architectures. In Figure 2.7 we illustrate the use of DESIRE to specify a generic BDI Agent. The main aspects of the BDI architecture (belief, desire and intention/commitment) are modelled as components with their inputs and outputs specified. These components can then be further refined; for example, the *intention and commitment* component is actually the result of the composition of a *goal determination* and *plan determination* component.

2.3.8 Agent Architectures in Toolkits

So far we have looked at abstract agent models and generic architectures, outside the context of any specific agent development environment. In this section we examine some of the most representative architectures developed within the context of practical agent toolkits and the un-

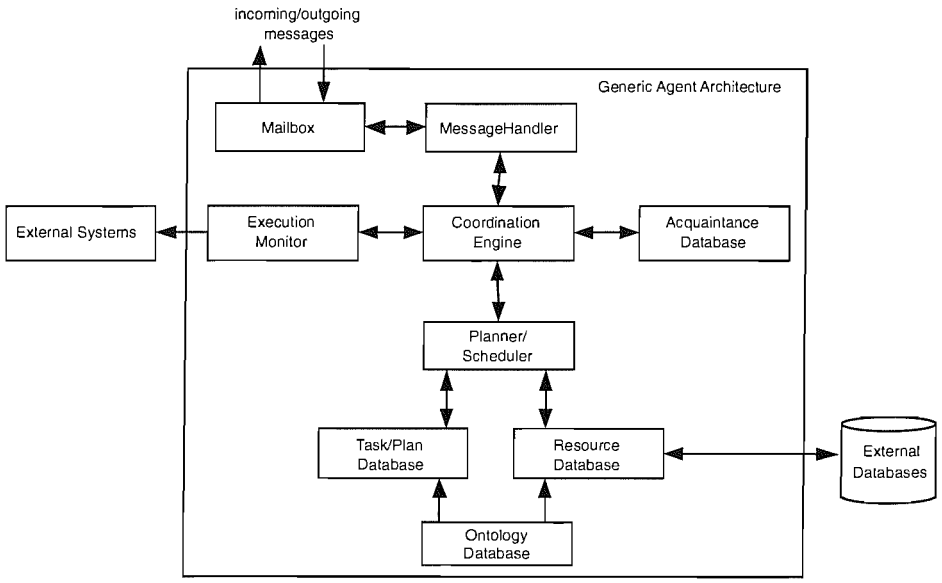

FIGURE 2.8: Generic ZEUS agent architecture.

derlying motivations for those architectures. The toolkits investigated where chosen for their maturity (in terms of years of development and breadth of application in a variety of domains), popularity (in terms of their prominence within agent research literature), and as exemplars of the variety of approaches.

ZEUS

The ZEUS agent development toolkit aims to provide both a development environment as well as a development methodology for multi-agent systems [155]. ZEUS is the result of experience gained while developing two real world multi-agent systems, one for business process engineering [157] and the other for multimedia information management [211]. It abstracts the common features of these two systems based on a philosophy which calls for a separation between domain specific problem solving and agent-level functionality, a friendly graphical interface for development, an open and extensible design and a strong support for standards. This last point is considered especially critical since without it they claim that industry wide uptake cannot be achieved.

According to the ZEUS perspective, agents are deliberative, so they reason explicitly about which goals to select and which actions to perform. They are goal directed, so any action performed is in support of a specific goal. They are versatile, so they can perform a number of goals and engage in more than one task. They are truthful, so when dealing with other agents

they always state the true facts. Finally, agents are temporally continuous, so they have a notion of time and can synchronize based on a clock.

Based on this approach, the ZEUS toolkit provides a set of components that represent specific agent functionalities such as planning and scheduling algorithms, agent communication language capabilities (using the FIPA ACL) and communication protocol implementations, ontology support and coordination.

The assembly of these components readily leads to the construction of what is termed a *generic ZEUS agent*, illustrated in Figure 2.8. Agents can send and receive messages, through *Mailbox* and *Message Handler* components. A *Resource Database* component has a list of the resources available to the agent, with the possibility to directly interface with external databases. Through the *Execution Monitor* component, agents can interface with external systems such as legacy systems and also keep track of actions. The *Coordination Engine* component handles the agent's goals, deciding which to follow or abandon. It also handles interaction with other agents, based on the available interaction protocols. Information about other agents, such as name and abilities, is kept in an *Acquaintance Database* component. Finally, the *Planner/Scheduler* component has the task of producing plans and the timings for when actions defined in the plans should be performed in reference to specific goals, as requested by the *Coordination Engine*.

This generic agent has all the rudimentary tools necessary to form the base of an agent functioning in a variety of domains. Although it is possible to provide different implementations for these building blocks, and therefore obtain different types of generic agents, it does not seem possible to deviate significantly from the organizational structure of the intercomponent relationships. Nevertheless, since the code for each of these components is provided as part of the overall ZEUS package, it is possible to configure them in any manner desired or add or replace existing components.

RETSINA

RETSINA (Reusable Environment for Task Structured Intelligent Network Agents) is a multiagent systems toolkit developed over a period of years, and at least since 1995, at the Intelligent Software Agents laboratory of Carnegie Mellon University's Robotic Institute. RETSINA has been used extensively in a range of applications, such as financial portfolio management [164] and eCommerce [213].

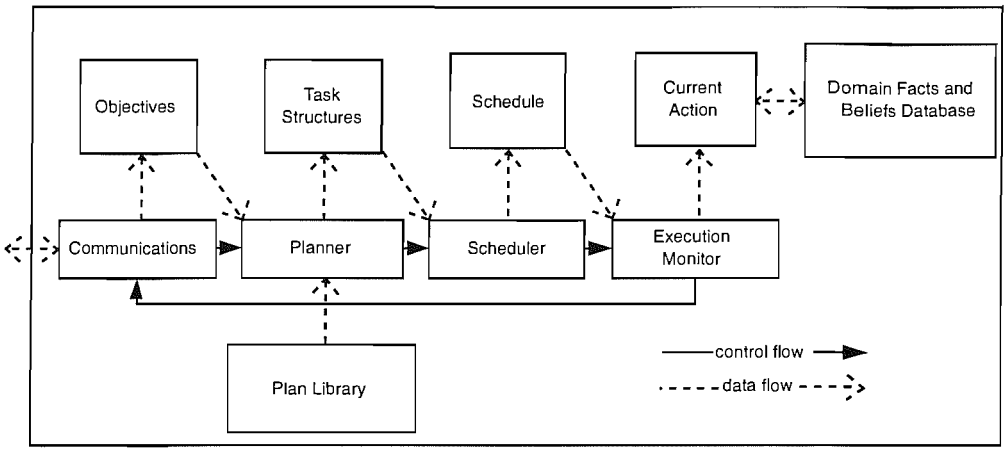

FIGURE 2.9: Retsina agent architecture.

The design of RETSINA is based on two central assumptions about agent applications development [206]. Firstly, multi-agent systems infrastructure should support complex social interactions between agents through the provision of services that are based on predefined conventions on how social interaction will take place. These predefined conventions refer, mainly, to the use of a common communication language, protocols and ontologies. From the perspective of the multi-agent system infrastructure, agents are seen as black boxes, but they are expected to be able to participate in social interactions based on these conventions. Secondly, agents in a multi-agent system engage in peer-to-peer relationships. Any societal structures, such as hierarchies, should emerge through these peer-to-peer interactions, and should not be imposed by a centralized approach. This is in recognition of the need to avoid a reliance on centralized control, and allow for truly distributed structures to emerge. These assumptions for multi-agent systems development lead to a very clear separation between individual agents and the supporting infrastructure.

An agent in RETSINA is understood, in abstract terms, as a standalone survivable piece of code with communicative and intelligent behavior. In real terms, it is understood as any piece of software that is able to interact with other agents, and with the RETSINA multi-agent system infrastructure, following the conventions defined in RETSINA.

All agents are derived from a *BasicAgent* class, which provides the main functions required for operation in a RETSINA multi-agent system, such as message handling, logging, visualization, and discovery of other agents. This agent-specific functionality is separated from operation within specific operating environments by placing agents in an *AgentShell*, which provides the necessary interfaces for interaction with the underlying operating system. The *AgentShell* also

provides basic management functionalities such as starting up or shutting down the agent and a timer module.

The reasoning and planning for agents is handled by the RETSINA Agent architecture, shown in Figure 2.9. It is based around the interactions between a *Communication* module that handles messages from other agents, a *Planner* that derives plans based on a provided set of goals and a plan library, a *Scheduler* that uses the output from the Planner to schedule when tasks will be performed, and an *Execution Monitor* that handles the actual performance of actions. These modules are supported by appropriate knowledge and beliefs, which are divided into *Objectives*, *Task Structures*, *Schedules*, *Current Actions* and a *Domain Facts and Beliefs Database.*

RETSINA divides agent functionality into four main classes that are built on top of the BasicAgent and represent specializations of the basic architecture to deal with different types of functionalities.

- *Interface Agents* interact with users by receiving inputs and displaying results.
- *Task Agents* carry out the main problem-solving activities by formulating plans and executing them by coordinating and exchanging information with other agents.
- *Information Agents* interact with information sources such as databases or web pages. The task agents provide the queries, and the information agents are specialized in retrieving the required information by interfacing with databases, the web, and so on.
- *Middle Agents* provide the infrastructural support for the discovery of services between agents.

IMPACT

The IMPACT (Interactive Maryland Platform for Agents Acting Together) system [203] is perhaps unique in the level of detail (both formally and informally) with which its respective components are presented and explained. From the very outset the developers of IMPACT state that it is essential to have a solid view of what an agent program is and how it can be distinguished from other programs. In addition, they believe that agent infrastructure should also provide a common set of services that the agents will need as well as the required structures that will enable interaction between the agents and the underlying infrastructure.

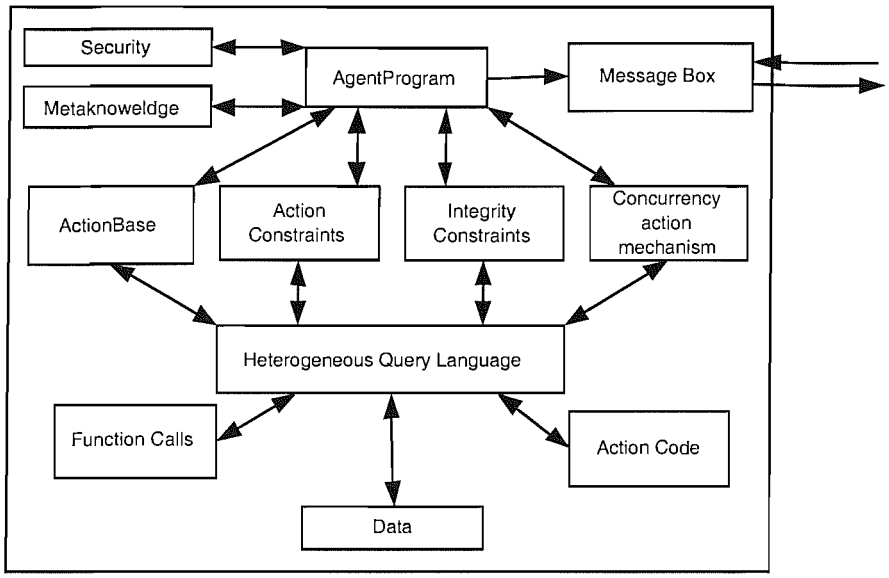

FIGURE 2.10: IMPACT agent architecture.

Agents in IMPACT are divided into two parts:

- the software code, which consists of data types and functions that can manipulate those data types; and
- the wrapper, which provides the actual *intelligent* agent functionality.

The software code could be any software program, and represents the actual interface to the environment through which the agent effects change in it. The wrapper represents the actual agent functionality that is able to manipulate the software code according to the behavior dictated by the wrapper's programming. This division is the IMPACT solution to the requirement for being able to agentify any program through a wrapper.

The wrapper is further divided into a set of basic components that come together to provide the IMPACT agent architecture, illustrated in Figure 2.10. All actions are regulated by the *Agent Program* that specifies which actions an agent should or should not perform in specific situations; the Agent Program defines what IMPACT terms the agent's *Operating Principles*. The Agent Program itself is defined according to an *Agent Program Language* that allows for a wide set of regulatory modalities (Do, Obliged, Forbidden, Waived and Permitted). An *Action Base* component maintains descriptions of all the actions an agent can perform, along with the preconditions for the execution of actions.

It is important to stress that IMPACT takes a wider view of what represents an action than many others. Everything an agent does, including tasks that are traditionally taken for granted or considered an integral part of the architecture, such as planning or timing, are considered actions that must be explicitly defined within the Action Base. Actions can be performed concurrently, and are regulated by a *Concurrent Action Mechanism* component that decides, based on the current agent state and desired actions, whether a composite action can be defined to achieve the desired actions. Concurrency is also regulated by a set of *Action Constraints* that explicitly define when certain actions cannot be performed concurrently. A set of *Integrity Constraints* specify which agent states are legal in a given context and ensure that the agent does not perform any actions that may violate these constraints. A *Heterogeneous Query Language* component provides the interface with the software code part of the agent. Finally, an agent is equipped with *Metaknowledge* that includes descriptions of the services the agent is able to provide, and beliefs about other agents, and a *Message Box* component that handles communication with other agents.

The most interesting feature of the IMPACT agent architecture, which clearly distinguishes it from other architectures, is the emphasis on ensuring that the agent operates within very well defined parameters. The agent architecture clearly stipulates the actions that are allowed, integrity constraints, action constraints, and so on. This provides a multilayered solution to the problem of being able to guarantee "correct" behavior. Furthermore, the development process of agents in IMPACT also includes several consistency checks that ensure there are no conflicting rules, such as both forbidding and permitting an agent to do something. We will not elaborate the details of these consistency checks here, but the interested reader can refer to the extensive articles on IMPACT elsewhere (see, for example, [86, 87, 88]).

JADE and LEAP

The JADE (Java Agent Development Environment) toolkit provides a FIPA compliant agent platform and a package to develop Java agents. It is an open source project distributed by TILab (Telecom Italia Labs) that has been under development since 1999 at TILab and through contributions by its numerous users. At the time of writing, version 3.1 is available, which implements the FIPA2000 specifications. The platform has undergone successful interoperability tests for compliance with the FIPA specifications. LEAP (Lightweight Extensible Agent Platform) is the result of a research project aiming to provide an agent platform that is suitable for limited

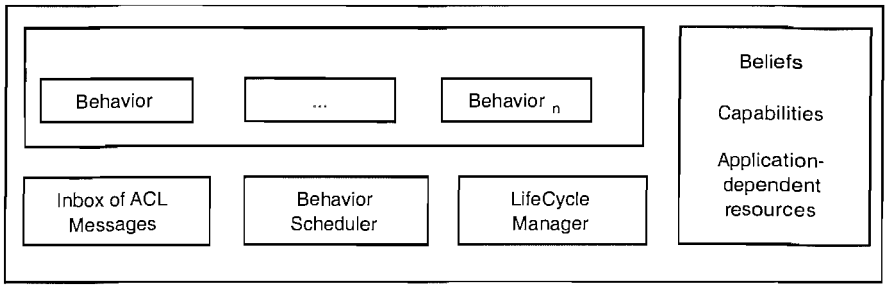

FIGURE 2.11: Jade agent components.

capability devices, such as PDAs and mobile phones [135].

The relationship between the two projects is that LEAP is a lightweight implementation of the core functionalities of the JADE FIPA platform, and can be used in conjunction with the JADE libraries for agent development. The latest release of JADE integrates LEAP so as to provide a unique toolkit that enables the development of FIPA compliant agent applications on devices ranging from limited capability mobile devices to desktop computers.

The JADE toolkit facilitates the development of agents that can participate in FIPA compliant multi-agent systems. It does not define any specific agent architectures but provides a basic set of functionalities that are regarded as essential for an autonomous agent architecture [22, 20]. These are derived by interpreting the minimum concrete programming requirements for satisfying the characteristics of autonomy and sociality. Autonomy is interpreted as an implementation of agents as active objects (that is, with their own thread of operation). The requirement for sociality leads to enabling agents to hold multiple conversations on a peer-to-peer basis through an asynchronous messaging protocol.

This basic single agent infrastructure is provided through an *Agent* class, which developers then extend to provide their own implementations of agents. Programs extending the Agent class operate within JADE containers that manage the agent lifecycle. Agents can be started, stopped, removed, suspended and copied. Each agent has access to a private message queue, where messages are stored until the agent chooses to retrieve them, and access to a set of APIs that allow the formulation of FIPA ACL messages. An outline of the main aspects of the agent class are illustrated in Figure 2.11.

Specific agent actions take place through a concurrent task model. Each task, or behavior as it is termed in JADE, is an extension of the Behavior class of the JADE toolkit. Each agent has a behavior task list, and the Agent class provides methods for adding or removing behaviors. Once

an agent is placed within a container and set into operation, behaviors are executed based on a round non-pre-emptive scheduling policy. Of course, complex tasks require a more sophisticated scheduling of behaviors as well as the conditional execution of behaviors. JADE provides models that are divided along the lines of Simple behaviors, to address tasks not composed of subtasks, and Composite behaviors, to address tasks made up through the composition of several other tasks. There are also cyclic and one shot implementations of Simple behaviors, and parallel, sequential and finite state machine implementations for Composite behaviors. Development is further aided by the provision of specific implementations of Behavior to handle basic tasks such as receiving or sending messages, and support for the set of interaction protocols defined by FIPA.

The LEAP core for JADE offers a lightweight version of the JADE container that can operate on PDAs. LEAP agents use a device specific *Communicator* module, which handles the specific connectivity protocols of the device and network at hand. Agents for limited devices use the same task based model as JADE agents, within the limitations of the device's processing capabilities.

Others

Our review of agent architectures within the context of agent development toolkits is inevitably limited since it would not be practical to review every single approach. However, we judge the systems reviewed to be particularly representative of the field.

Other significant efforts are: SoFAR [146], in which the focus has been on extensive support for integrating and managing ontologies within an agent-based system; CoABS [130], in which the aim was to provide infrastructure for accommodating the integration of agents developed using different agent toolkits; DECAF [105], in which the focus has been to provide robust agent architectures that could deal with run-time scheduling of tasks; Sensible Agents [14], in which agent autonomy can be varied from command-drive, slave master and fully autonomous agents. Significant mobile agent systems such as D'Agents [106], Aglets [133], Mole [17] and SOMA [18] have not been reviewed because their focus is primarily on the underlying mobility mechanisms or the security mechanisms. Therefore they have very little to say about the nature of agenthood or the relationships between agents.

2.3.9 Discussion

The review of agent models confirms the richness and variety that exists within agent research, but also highlights two important issues, described below.

- Firstly, there is a variety of agent architectures but there are no clear attempts (with the possible exception of SMART) to describe diverse architectures using a common set of concepts, so that developers can adopt models that can better address the wide range of requirements that heterogeneous computing environments place, as we discussed in Chapter 1.
- Secondly, there are few links between the architectures used by agent development environments and more abstract agent models. This indicates that there is a general lack of continuity from theory into practise.

Efforts dealing with the development of appropriate agent architectures can be divided along the lines of whether the focus is on deliberative (reasoning about the environment in order to reach a decision), reactive (simply matching environmental stimuli to responses), and hybrids of the two that mix reaction and deliberation. The arguments for the relative benefits and disadvantages for each are well rehearsed. Proponents of deliberation recognise that the execution of complex sophisticated tasks through a clear, long-term goal-directed behaviour requires some amount of internal reasoning, while the reaction group believes that a lot can be achieved through the interaction of basic components that will give rise to emergent intelligent behaviour. This is certainly an important debate but one that does not aid application developers directly. What is more significant from an application development point of view are ways of choosing between approaches and possibly mechanisms for mixing approaches so as to derive the best possible results. The more abstract models of the SMART framework [82] and Russell and Norvig [186], provide better guidelines for characterising the *range* of agent types. Notably, SMART does this through a clear and unambiguous, formally defined framework.

The component-based agent models proposed, such as Sabater et al. [189] and DESIRE [37], provide some redress for the need of linking abstract models to practical implementation concerns, through an approach that allows for the creation of agent architectures based on the modular composition of diverse components, each one fulfilling a clearly defined functionality. However, they are limited by the lack of an abstract agent model that provides some guidelines as

what are the implications of different configurations of architectures within the context of an agent-based approach.

The toolkits discussed attempt to tackle a wide range of issues, but the lack of strong underlying theoretical principles leads to a confusion as how to go about providing solutions. In a sense, the requirements for such toolkits are *too* wide, ranging from defining agent models to providing guidance as to the route from problem to application design and finally providing appropriate technical implementations and lower-level middleware support. Toolkits would benefit from a clearer delineation between different concerns so as to better focus on either just specific agent issues or lower-level middleware issues [8]. Revealingly, although all toolkits provide some form of agent architecture only IMPACT goes so far as to formally define what constitutes an agent program. The only drawback of the IMPACT approach, however, is that the formalisms do not allow for alternative architectures to be defined but only apply to the IMPACT agent architecture. In other words their theory of agents in closely tied to the agent architecture making no allowance for a diversity of architectures within an application.

Bryson and Stein [42] identified this general problem of a multiplicity of architectures but few means to choose between them and claim that in order to make progress in agent research it is necessary to find a way to describe different *idioms* of agent architectures in a *common way* so as to allow others to understand and utilize them. In addition, a clearer separation between concepts and implementation would provide developers with a greater choice for matching technologies to ideas so as to best suit their needs.

Luck and d'Inverno have also recognised this lack of reconciliation between practice and research [80], as well as the difficulty in tracking progress [137] and SMART has been developed with exactly such concerns in mind. SMART makes no assumptions about agent architectures or about the underlying infrastructure but attempts to provide an appropriate ontology of agent types and relationships that will allow the description of a range of situations. Although this approach is both conceptually elegant and sound since it does manage to describe a wide range of situations through simple mechanisms its immediately apparent use is in *describing* existing systems. It does not, however, aid in *building* systems.

In a way its strength, in that it lies at a very high conceptual level, is also its weakness since paths down to practical concerns are not clear. Nevertheless, we believe that SMART is an ideal candidate to act as a basis for our own work. A particularly useful aspect of SMART is that all concepts are formally presented through the Z specification language [201]. This leads to very

few doubts about the exact meaning of those concepts. Our aim is to provide these paths to implementation without losing any of the expressive powers SMART currently has.

2.4 Inter-Agent Issues: Models of Agent Interaction

Having discussed models for *individual* agents, we now turn our attention to the issue of supporting interactions between agents. At this level we examine general models for interaction between agents that can facilitate the identification and reasoning about relationships. We discuss SMART [82] once more, since the framework also deals with agent relationships, as well as Social Power Theory [50], which provides a framework for characterising different types of dependencies between agents, and TuCSoN [162], which provides an interaction model in support of coordination and regulation of interactions. Subsequently, we examine interaction models stemming from research in agent methodologies [127, 160, 233]. Finally, we discuss the issue of run-time agent discovery, i.e. how can agents discover what other agents are present within an environment, which is key to enabling dynamic interactions between agents.

2.4.1 SMART

The SMART framework builds on its model for individual agents to describe relationships between agents, and provides formal definitions for a multi-agent system. In SMART [82] a multi-agent system arises from the interaction between two or more agents where at least one is autonomous and where at least one relationship is created due to an entity satisfying a goal for another entity. An autonomous agent is required because only autonomous agents can generate their own goals, so without the presence of one a multi-agent system would never be set into motion. An autonomous agent cannot, however, be expected to satisfy all of its goals on its own so eventually it will have to seek assistance elsewhere. It is at this point that a relationship with another entity, with a view of satisfying a goal, is created and a multi-agent system is instantiated.

The agent seeking to satisfy the goal, which in SMART is termed the *viewing* agent, must locate a *target* agent to adopt the goal. SMART defines relationships between agents and non-autonomous entities as *engagement* relationships and relationships between autonomous agents as *cooperation* relationships. This illustrates the different nature of the relationships between autonomous

agents since autonomous entities will only enter a relationship if it is in line with their motivations, while non-autonomous entities are considered to be predisposed towards satisfying the goals of any engaging entity. Finally, SMART uses the notion of an *engagement chain* to represent the situation where a single goal generated by an autonomous agent leads to the creation of a series of relationships between server entities, resulting in a chain with the autonomous agent at the start of the chain and the goal that is being satisfied for that autonomous agent dependent on all members of the chain performing their task. In such engagement chains the engagement between an entity and the next link further down the chain is a *direct* engagement while the engagements between server agents that are removed more that one link apart are *indirect* engagements.

2.4.2 Social Power Theory

Social Power Theory attempts to provide a theory of agent interaction based on the analysis of the *dependencies* between agents [144]. Agents become dependent on other agents when they cannot achieve their goals individually, leading them to interact with others in order to obtain help. However, since agents may be autonomous entities with their own goals, it is necessary to *influence* other agents to adopt those goals. As a result, *networks of dependencies and power* between agents are created. Conte and Castelfranchi [59, 61] argue that by allowing autonomous agents to perform reasoning about such networks different models of interaction such as cooperation, social exchange, coalitions, and so forth, may emerge.

Individual agents are considered as having *personal powers*, determined by their capabilities, resources, skills, knowledge or motivations, which they use to satisfy their own goals [50]. When such powers can be used to satisfy goals of other agents or when these powers are not sufficient to satisfy the goals of the agent relationships of *power* and *dependence*, respectively, are created. Using this approach the different types of relationships are categorised as follows.

- **Mutual influence** is a situation where two agents depend on each other for the same goal.
- **Reciprocal dependence** occurs when two agents depend on each other, but for different goals.
- **Unilateral dependence** occurs when one agent depends on another for its goals, but the other agent does not depend on it for any of its goals.

Using this typology of relationships agents can go on to reason about how they could influence other agents to adopt goals, for example through the promise of a price or through the threat of sanctions. According to the number of agents available to an agent to satisfy its goals it can be described as more or less dependent on the society, while according to whether an agent is required by a number of other agents to satisfy their goals it can be described as more or less useful to society.

2.4.3 TuCSoN Coordination Model

The TuCSON interaction and coordination model [162] is based around the notion of agents interacting *through* independent coordination media, called *tuple centres*, spread across Internet nodes. Each tuple centre is associated to a node and is identified by a name. Tuple centres are enhanced with a *behaviour specification*, which defines behaviour in response to communication events taking place at the tuple centre. These responses are termed *reactions* and are defined through a sequence of *reaction goals*. Reactions lead to changes in the state of the tuple centre.

Agents exchange messages through tuple centres, with the implication being that the perceived result of a communication from one agent to the other is a *combination* of the communication primitive along with the changes caused to the tuple centre by the triggered reactions. Through these mechanisms coordination and regulation of agent interactions is decoupled from the agents themselves and made the responsibility of the individual tuple centres.

2.4.4 Agent Methodologies

As we discuss in Chapter 1, a prerequisite to the development of an effective methodology is the provision of agent abstractions that will form the main artifacts that support the process of design specification. In this section we examine the most prevalent methodologies which provide such abstractions.

GAIA and ROADMAP

GAIA is a methodology developed by Wooldridge et al. [233] and is motivated by the need for methodologies that are specific to agent systems as opposed to general object-oriented analysis and design. GAIA was designed to deal with coarse-grained computational systems, to

maximize some global quality measure, and to handle heterogeneous agents independent of programming languages and agent architectures. It assumes static organizational structures and agents that have static abilities and services. ROADMAP extends GAIA by adding elements to deal with requirements analysis in more detail, by using use cases and by improving support for handling open systems environments [127]. Moreover, it supports the specification of interactions based on AUML [16]. Here, we present a unified view of both methodologies.

The highest level concept is that of a *system* as an organisation of interacting agents. Agents have *roles*, which are generic characterisations of specific types of behaviour, such as 'president' or 'employee'. Roles are akin to *classes* in object-oriented design. Each role has an associated set of *responsibilities*, *permissions*, *activities* and *protocols*. Responsibilities are the functionalities that different roles should perform and are associated with safety and liveness attributes in order to better characterise their importance or priority. Permissions represent what an agent is allowed to do, typically what information sources it can access. Activities describe the sort of computations agents can perform and protocols the ways in which different roles can interact.

ROADMAP extends GAIA by also considering an *environment model*. This model provides the basis for describing any environmental changes during the system execution. It consists of a *tree hierarchy of zones* in the environment (for example, the Internet, a local computer or the physical environment of a house) based on object-oriented inheritance and aggregation and *zone schemas*, characterized by a textual description of the zones. An environment may contain *static objects* (any entity in the environment known to some agent, but with no interaction), *objects* (any entity an agent interacts with), *constraints*, *sources of uncertainty* (which have to be analyzed), and *assumptions* made about the zones.

SODA

The SODA (Societies in Open and Distributed Agent Spaces) methodology departs from the premise that inter-agent issues are as important as intra-agent issues, and should be treated as such within the context of a methodology [160].

Agents in SODA are described by *agent classes* which specify the *roles* (one or more) that an agent can occupy, and can be further characterised with information such as *cardinality* (i.e. how many agents of one class can exists) or *location* (with respect to a predefined topological model). A role defines tasks that an agent occupying the role is responsible for. Tasks are either

individual (requiring well-defined competence and limited resources) or social (requiring access to several different resources). Social tasks are assigned to *groups* of agents with individual roles defining the responsibility of individual agents within the social task. Finally, *interactions protocols* are also associated to roles and define how agents may interact.

Agent societies are characterised by the social tasks that must be undertaken, the set of permissions associated with behaviour in the society, the participating social roles and the interaction rules associated to groups. It is envisioned that societies are designed around coordination media (such as TuCSoN described above) that regulate the interactions between agents.

Finally, SODA also uses an environment model where resources are mapped onto what are called *infrastructure classes*. These are characterised by the services, the access modes, the permissions granted to roles and groups, and the interaction protocols associated to resources.

Others

Similarly to agent toolkits there is also a wealth of agent methodologies, and the ones we have considered so far are exemplary of the general direction of research. We briefly discuss some other relevant methodologies below and just provide pointers to yet others, such as MESSAGE [45], Prometheus [163], and PASSI [65].

MaSE MaSE (Multi-Agent Systems Engineering) [71] is based on UML notation [185], which it applies to the task of analysing, designing and implementing an agent-based system. A basic notion of MaSE is that of roles as an aggregation of system goals, where goals are functional requirements of the system. In order to derive roles MaSE begins by identifying, analysing and decomposing the system goals. Use cases are then used to derive sequence diagrams that will reveal communication paths and interactions between different aspects of the system. With goals, use cases and sequence diagrams in place roles are derived. Roles are then decomposed in order to attach specific tasks that will achieve the required goals.

Kinny et al. Kinny et al. provide a methodology clearly directed at BDI agents that also takes into account object-based techniques. The resulting proposal [132] defines an approach based on developing three views of the system. An object model that describes the objects and their associated data structures, a dynamic model describing events, actions and inter-

actions and a functional model that describes the flow of information in the system. Once more the ideas of roles is introduced and used in a similar manner as GAIA and MaSE.

Tropos The Tropos methodology is also based on object-oriented techniques, offering processes for the application of UML mainly for the development of BDI agents [103, 150]. Tropos makes use of the $i*$ concepts, such as actors (where actors can be agents or roles) and social dependencies between actors (including concepts such as goals, tasks and resource dependencies) [235]. The use of $i*$ provides clear definitions for basic concepts that underpin all phases of the methodology enabling the specification of *actor and dependency models*, *goal and plan models*, *capability diagrams* and *agent interaction diagrams*.

2.4.5 Discussion

The models for interaction reviewed here can be divided along two broad lines. On the one hand, models such as SMART and Social Power Theory, are *prescriptive* since they provide a framework for reasoning about and relating different types of interactions. On the other hand, practically all the models used within methodologies are *descriptive*, since they provide tools for describing relationships through roles and interaction protocols, but no means for reasoning about the implications of different types of relationships. The exception to this categorisation is TuCSoN, which is more concerned on how interactions can eventually be regulated.

Now, for effective systems design both prescriptive and descriptive tools are required. The former for facilitating reasoning about relationships and their implications to system design and the latter for specifying such relationships. However, while there is a wealth of descriptive tools, as evidenced by the wealth of methodologies, the prescriptive tools are not adequate. SMART only focuses on relationships that are a result of agents sharing a common goal. However, in practice there are other types of relationships such as conflicts that must be tackled. Social Power Theory, overcomes this problem since it supports reasoning about both relationships where common goals are shared as well as other types of dependencies. However, Social Power Theory lacks an underlying abstract agent model and, in general, links to practical agent development that would enable us to use it within the context of practical agent infrastructure. In fact, SMART has been used to describe social dependency networks [82], and we shall return to compare it to our own work in Chapter 5.

Finally, there is no support within the context of methodologies or the abstract agent models

discussed for identifying possible relationships at run-time. Such a capability is crucial for dynamic agent systems, where agents may enter or leave at any point. We aim to address this issue by building on the SMART abstract model, which will provide the necessary theoretical underpinning.

2.4.6 Agent Discovery

The problem of dealing with the run-time discovery of agent capabilities has been signalled relatively early through research into agent technologies [67], and comes under a number of headings such as capability brokering, matchmaking, or is simply considered as one aspect of the wider problem of agent coordination. Genesereth and Ketchpel divided the solutions into two broad categories [100]; *direct coordination*, where agents requiring services have to handle on their own the problem of finding a service provider, and *assisted coordination*, where agents rely on specialised programs that assist in the process. The direct approach, however, is only effective in situations where the number of agents is fixed, relatively small and communication with them can easily take place. In open, heterogeneous environments assistance in the discovery process is practically a necessity. Accordingly, the bulk of research has focused in creating appropriate mechanisms for providing assistance in the discovery process.

The space of possible mechanisms, divided according to flows of information between clients, service providers and the facilitating programs has been comprehensively modeled by Decker et al. [69]. They call any program that facilitates the matchmaking process a *middle agent*, and define a space of nine alternative middle agent mechanisms based on what information each of the three agents in the process (client, middle agent and service provider) have available. The information consists of what service is requested and what services are provided. Within this space the most commonly used types are: *brokers*, which are aware of what services are available and what service requests are made and match clients to providers accordingly; *matchmakers*, which provide a list of what services are available to clients that choose from the list who to contact (i.e. a yellow pages service); *blackboards* where service requests are posted and providers choose which client to contact. Wong and Sycara [228] extended this work by introducing six dimensions of middle agents based on information held by a middle agent: who is the information provider; how is the information dealt with once received; how can the information be queried; how detailed are queries; and does the middle agent act as an intermediate between all transactions between the provider and the requester agents.

A prerequisite to matching a service request to a service provider is that the description of the service request and the available services is made in an appropriate format that can be understood by all the related parties. Furthermore, the agent communication language used needs to provide appropriate performatives that can deal with service requests. The latter issue is covered by both KQML [95] and the FIPA ACL (www.fipa.org), which have perfomatives such as *advertise, subscribe, recommend*, and *broker*. The former issue has been addressed through the development of a number of alternative languages. The LARKS language (Language for Advertisement and Request for Knowledge Sharing) [207], which is used within the context of the RETSINA toolkit [206], describes service requests and advertisements using the same structure, including information such as the context of the capability description, necessary inputs and outputs and constraints to the service. Along with the LARKS description language its developers provide a number of matchmaking algorithms using both syntactical and semantical matchmaking. An alternative approach is taken by Cassandra et al. [47], which builds on their experience with the InfoSleuth system [152]. They criticise the LARKS approach as providing overly detailed descriptions and as a result may not scale well. They suggest a layered approach where capability description is divided into the conversation language used to communicate with the service, the interface to the service, the semantics of what the service does, and the domain the service operates over. Specific ontologies can be used for each of these aspects and the capability description framework allows all these descriptions to be composed into one advertisement. According to the developers this provides, at the same time, a more flexible approach to the problem that allows agents to take advantage of specific ontologies for describing different aspects of their service and leads to a more uniform capability description. The IMPACT [203] toolkit uses a more simplified approach, with queries of the form *sell:tickets(opera)?*, but then employs powerful semantic matchmaking algorithms that draw relationships between concepts, for example between *theatre* and *opera*, based on a thesaurus that can be updated by each agent participating in the system.

More recently, with efforts to standardise ontology languages through the Web Ontology Language [143], and initiatives such as Semantic Web Services [4], the research is coalescing around some well understood mechanisms for discovery [205]. As such, what is important within the context of our work is to demonstrate how agent discovery and agents that facilitate such discovery can be understood within a wider framework of models for agent-based infrastructure.

2.5 Organisational Issues: Regulating Agent Societies

The review of models of agents and agent interaction covers the main concerns of our research. However, we also briefly consider organisational issues since we will attempt to relate such issues to our work within the context of implementation in Chapter 6. At this level we are concerned with work that attempts to impose structure upon and regulate agent societies. We briefly discuss the main proposals coming from the areas of distributed systems management policies, norms and electronic institutions.

2.5.1 Distributed Systems Management Policies

As distributed systems have grown in the number and complexity of different interacting components it has become necessary to introduce automated means for the management of the behaviour of components. *Policy-based network management* addresses this need by separating the definition of management policies from their enforcement through automated policy managers [200].

Perhaps the most influential academic research in policy specification comes from the conceptual grounding provided by Sloman [200]. Sloman defines policies as *"one aspect of information which influences the behaviour of objects within the system"*. Policies are developed in the context of a *subject* influencing a *target* in a distributed environment. The basic construct for policies coming from Sloman's work is the notion of *authorisation policies* that define what is or is not permitted (positive or negative authorisation) and *obligation policies* that define what a manager must or must not do (positive or negative obligation). Obligations are subject to an *interpretation* from the manager and, as a result, can be disregarded while authorisations cannot be disregarded. This approach is extended to also cover issues such as role-based policy specification [141] where policies are defined based on the *role* of a manager in an organisation.

The work of Sloman has been adopted by agent researchers, most notably Bradshaw et al. [204] through KAoS, and combined with Semantic Web technologies so as to provide several of the required expressive constructs for defining authorisations and obligations and their delegation. This work also takes into account some of the issues relating to conflicting policies between different domains, and provides means for resolving them [216].

Kagal et al., developed the Rei policy language [128], following a more decentralised and adap-

tive model than KAoS. The dynamic modification of policies is supported using speech acts and the suggested deployment models for this work examine different scenarios, such as FIPA-compliant agent platforms [2], web pages and web services.

2.5.2 Norms

Norms, at their most basic, can be considered as a means of regulating behaviour between agents. However, unlike policies which are defined and imposed on agents, norms may emerge through a variety of means. Norms have been studied extensively within the fields of philosophy, sociology and law [184, 214, 215], with such research informing the development of frameworks of norms within agent-based systems. They are understood through a number of different perspectives, and we outline some of the main ones here.

- One line of research [13, 28, 221] considers norms as *patterns of behaviour* that emerge from the interactions between agents without previous planning. It attempts to account for the choices agents make and the constrains society imposes through the interaction of autonomous entities.

- Norms can also be considered as *constrains on actions* [38, 195]. This is perhaps the most similar view to policies, since norms specify which actions are forbidden or allowed within a particular context.

- Social commitments represent agreements to do something between two or more agents [48, 119]. Social commitments can also be considered as norms since they represent the *obligation* of agents to do something, and social pressure can be exerted to make agents fulfill them.

- Finally, norms are also considered as *mental states* that may influence agent behaviour [60, 59].

In addition to these alternative approaches to reasoning about norms a number of models have been developed to allow the specification of norms (e.g. [49, 62, 234]).

[2] http://www.fipa.org/

2.5.3 Electronic Institutions

Electronic institutions attempt to make explicit the structure that should regulate the interactions between agents by providing specifications for what interactions are possible within a given context and what are the implications of interactions in terms of commitments created. The conventions that govern electronic institutions are typically divided into ontological and communication conventions [153], social conventions that govern collective interactions [181], and rules that normalise individual behaviour [182].

At the level of ontological and communication conventions an electronic institution makes explicit what are the entities an institution deals with, such as the goods to be traded, the participants and the roles they occupy as well as issues such as locations, time intervals, and so forth. Furthermore, the *language* for interaction is made explicit through access to a common *dialogical framework* [154]. A dialogical framework is composed of a communication language, a representation language for the domain content and an ontology.

The social conventions are defined by making explicit the possible activities within an institution as a composition of multiple, well-separated, and possibly concurrent, dialogical activities, each one involving different groups of agents that follow well-defined communication protocols. Such activities are termed *scenes*, while changes between scenes are defined by *performative structures* which establish links and traversal paths.

Finally, rules are divided into *intra-scene* rules, which dictate foe each agent role within a scene, what can be said, by whom, to whom, and when, and *extra-scene* rules, which define what paths agents may follow between scenes depending on their roles.

2.5.4 Discussion

In this section we examined some of the different approaches to *regulating* heterogeneous and dynamic systems. While these issues are not the main focus of our research, the need for regulation is one of the motivations behind our aim of providing an appropriate model of agent interaction. In order for developers to choose an appropriate regulatory mechanisms they should first be able to identify and characterise the types of relationships that may emerge. We will return to examine the issue of regulation within the context of dynamic relationship identification and characterisation in Chapter 6.

2.6 Conclusions

The review chapter highlights several shortcomings in existing research that are broadly related to the lack of principled approach to constructing both individual agents and the reasoning about interactions between agents, through abstract specification that is linked to practical implementation.

At the level of individual agents there is a wealth of alternative agent architectures, with each attempting to better address the issue of supporting complex behaviour in the face of a heterogeneous environment. However, there is little work in providing overriding concepts, within the context of agent-based development, that can underpin such efforts so as to enable reuse across domains and effective comparison between alternative solutions. A notable exception is SMART that does provide such an abstract framework for describing a variety of agent types. However, SMART is limited in that its focus is primarily on the *description* of agents and does not provide links from those descriptions to agent *construction*.

At the level of agent interaction, although there is a variety of frameworks, largely stemming from research in agent methodologies, that enable us to *specify* issues such as agent roles and interaction protocols, there are few models that allow us to reason about different types of relationships and the implications such relationships may have on the overall system performance. In this respect Social Power Theory is particularly useful, however, it is limited by its lack of reference to a clear abstract agent model. The SMART framework offers support for a specific type of agent relationships (where agents share a common goal) but it does not address other types.

In conclusion, principled, reusable models in support of agent construction and reasoning about agent relationships are key to enabling the construction of multi-agent systems in dynamic, heterogeneous environments. The review of existing work indicates that while there has been some effort to provide such models it is limited in its scope in that it only deals with individual agent construction or agent interactions or its applicability because of few links between abstract models and practical implementation concerns.

Chapter 3

SMART

"Truth emerges more readily from error than from confusion."
Francis Bacon (1561 - 1626); English scientist and philosopher

3.1 Introduction

Our overarching aim is to provide *resuable* models that will both support the construction of individual agents and enable reasoning about the relationships between agents, so as to facilitate the development of agent-based applications in heterogeneous and dynamic computing environments. In Chapter 1 we indicated that a central task towards achieving this aim is the identification and, if possible, adoption of an existing conceptual framework that can provide some of the necessary abstraction to support agent development. As we have have already argued in that chapter, it is preferable to adopt an *existing* set of concepts and develop and refine them, rather than begin afresh. By building on an existing, well established, theoretical framework, we can avoid reinventing basic notions and adding to the existing multiplicity of different approaches, as well as benefit from the existing efforts in developing and refining that framework. In addition, the exposition of our proposed models will gain from having to *explain* the reasons behind the need for change and progression *within* a well defined conceptual framework.

However, in order to be able to effectively develop on the basis of an existing conceptual framework, it must be amenable to such further development and refinement along the lines of our research aims. As discussed in Chapter 2, SMART [82] goes some way towards fulfilling these basic needs by providing a theoretical underpinning that is clearly independent from specific

agent architectures and makes minimal assumptions about agent interactions. From the outset, it was developed with the aim of providing clear and unambiguous meanings for common terms and concepts so as to both enable alternative designs to be described on a common basis and to provide foundations for subsequent development [78]. It has been refined and used over a number of years, and its capabilities have been demonstrated to some extent through work describing a number of alternative agent architectures and interaction mechanisms, such as BDI agents [75, 79] and the contract net protocol [77]. The comprehensive set of concepts that SMART provides, ranging from the description of individual agents up to the relationships between agents, combined with its proven ability to describe a number of existing architectures and interaction protocols make it particularly suitable, and clearly in line with our aims of providing reusable models for agent systems.

In this chapter we present a detailed account of the SMART framework [82], since it underpins all other work in the thesis, by introducing some of the most salient concepts together with the formal notation used to define them. In addition, we make clear the relationship between SMART and our specific aims, and thus set the scene for the rest of the thesis. At the same time, we highlight some of the shortcomings of SMART, within the context in which we aim to use it, and introduce some initial refinements with reference to the basic notions of agenthood.

The chapter begins with a high-level overview of how the work presented in this thesis extends and refines SMART, following an identification of the main shortcomings of SMART with respect to supporting agent-based system construction in dynamic and heterogenous environments. We then discuss the use of the Z notation [201] within SMART and provide a brief overview of its main features. The presentation of Z allows us to proceed to the more detailed introduction and discussion of the foundational concepts of SMART. We begin with the concepts supporting the description of individual agents. We analyse the notion of agenthood to some extent and refine it so that it best suits the needs of *practical construction*. In particular, we distinguish between extremely simple agents and more complex types by combining SMART's notion of agenthood with Wooldridge and Jenning's defining agent characteristics [232]. The resulting agent types can be more closely related to the types of agents encountered in practical agent design and construction, and are used to support the models of agent construction developed in Chapter 4. Subsequently, we introduce and discuss SMART's approach to describing agent relationships and identify its shortcomings in this respect. We use these shortcomings to motivate the need for the work on agent relationships presented in Chapter 5 of the thesis. This chapter concludes with an overview of the main issues raised and how they will be addressed throughout the remainder of

the thesis.

3.2 SMART, *act*SMART, and SMART+

The SMART framework acts as a central point of focus from which we depart to provide the necessary refinements and extensions so as to deal with our overarching aims presented in Chapter 1. Here we provide an overview of the relationship between SMART and the extensions to SMART that we introduce in the thesis. In order to justify these extensions, we also provide a brief introduction into the main shortcomings of SMART, while a more detailed analysis of the shortcomings is developed throughout the chapter.

The SMART framework provides us with a set of abstract, formal models to support the specification of individual agent architectures and multi-agent systems. However, although these models provide a good departure point for us, they are limited in two important ways.

- Firstly, with regard to individual agents, there is no clear path from the abstract specification of agent architectures to their practical implementation. This constrains the applicability of SMART to the *design* and *construction* of agent systems. If we are to achieve our aim of providing theoretical models in support of *practical* development we must provide a clear path from the abstract concepts of SMART to their *implementation.*
- Secondly, with regard to multi-agent systems, the models are restricted to representing agent relationships only in those cases where the agents share a common goal. However, in the types of open multi-agent systems that we aim to support, it is also necessary to model possible conflicts between agents, and identify opportunities for cooperation between them (as discussed in Chapter 1).

In addressing these shortcomings, we extend and refine SMART in two directions by providing both more practical models and adding to the abstract concepts already there, as illustrated in Figure 3.1. In the figure we represent three different levels of abstraction. Firstly, the conceptual infrastructure defines the models that we can use to specify an agent system. Secondly, the *specification artifacts* represent specific instantiations of those models in order to design an agent system. Finally, the *design and practical implementation* represents the resulting multi-agent systems developed using the specification artifacts from the level above. At each of these three levels we describe what is provided by SMART and what specific extensions we add to it.

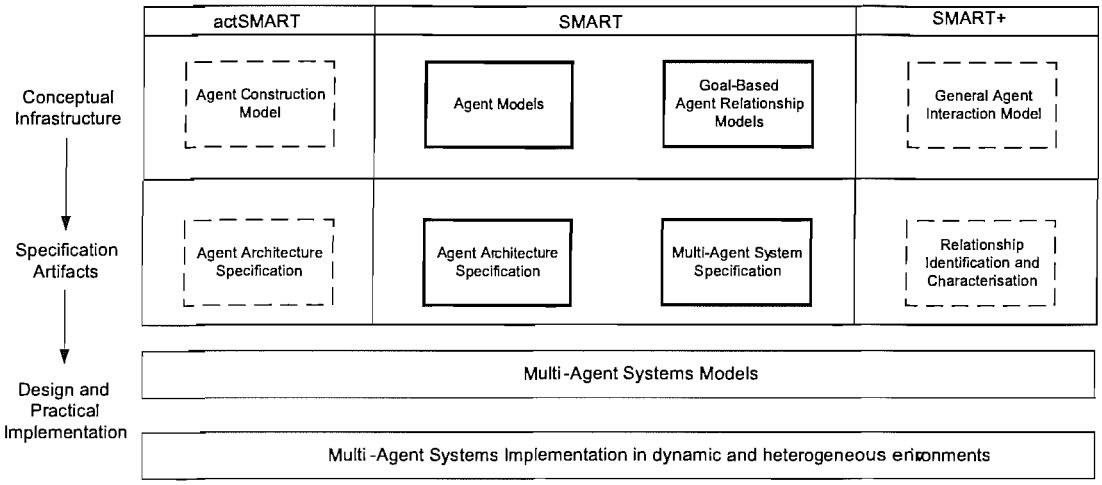

FIGURE 3.1: The relationships between *act*SMART, SMART, SMART+

The SMART framework provides the conceptual infrastructure for describing agents and goal-based agent relationships. These models *enable* the specification of agent architectures and multi-agent systems, respectively.

Now, we also require appropriate *practical* models for agent construction that will provide a clear path from the abstract agent models of SMART to their implementation. We therefore need to extend SMART in a more practical direction, while basing this extension on the abstract agent models already there. In the figure, this extension is under the heading of *act*SMART, (*A*gent *C*onstruction *T*oolkit for SMART). At the conceptual infrastructure level, *act*SMART provides a model for constructing agents which, at the specification artifact level, can enable the specification of agent architectures that can then find practical implementation at the lowest level. The shading of the *act*SMART boxes indicates that it lies at a more practical level that is closer to implementation, rather than SMART's abstract level.

In addition, we also extend the abstract conceptual infrastructure of SMART through a more general model of agent interaction that enables the identification and characterisation of a wider variety of agent relationships. We include these concepts under the heading of SMART+, since they lie at the same level of abstraction as SMART.

If we are concerned with the development of multi-agent systems operating in dynamic and heterogeneous environments then we must ensure that, when combined, *act*SMART, SMART and SMART+ provide appropriate models and specification artifacts to support design and implementation.

Before considering this further, however, we must introduce the notation used in the remainder

of this thesis.

3.3 The Z Specification Language

Formal methods are increasingly being used in a variety of subfields of computer science, as well as gaining ground in industry [217]. In particular, many formal methods have been constructed for use in the delivery of greater precision and clarity when defining systems and applications, and a large body of computing research has focused on their elaboration and development [31, 113, 212]. These include various kinds of temporal and modal logics [54], and specification languages such as Z [201],VDM [124], B [134] and CSP [114], the majority of which are supported by a range of software tools that facilitate their use and, in some cases, can even automatically generate software code. As well as specifying systems, and providing a means for mechanical checking of correctness, type correctness, etc, to reveal inconsistencies, ambiguities and other problems [56], formal specifications can also be used to specify *abstract* concepts to aid in their representation and reasoning about them.

In order to make the presentation of concepts of the SMART framework as unambiguous as possible, and to ensure consistency in the re-use of concepts throughout the framework, Luck and d'Inverno use the Z specification language [201]. Z enjoys wide recognition, both in industry and academia, as a powerful means for specification, and is supported by several text books (e.g., [30, 170, 229]), articles (e.g., [26, 27]), and industrial case studies (e.g., [2, 58]).

The specific benefits offered as justification for the use of Z in specifying SMART are briefly outlined below.

- Z is more *accessible* that many other formalisms, since it is based on existing elementary components such as set theory and first order predicate calculus. This ensures that it is generally accessible, requiring no special expertise, and reduces the learning curve for anyone aiming to use the framework.

- Z is sufficiently expressive, allowing for a consistent, unified and structured account of a computer system and its associated operations.

3.3.1 Z notation

A Z specification is made up of formal mathematical statements, which are typically combined with informal explanatory text to complement the formal statements. Providing both formal and informal descriptions is especially useful since the informal description provides direct access to the concepts, while the formal presentation ensures that any ambiguities are avoided.

Schemata

Z is based on set theory and first order predicate logic, and its basic unit is the Z *schema*, which allows specifications to be structured into manageable modular components. Schemas are divided into a *declarative* part that defines variables and their types, and a *predicate* part that defines relationships between, and restrictions on, variables. For example, the schema below defines a *Pair* to consist of two variables, *first* and *second*, with the predicate part declaring that *first* should be smaller than or equal to *second*.

Pair

$$first : \mathbb{N}$$
$$second : \mathbb{N}$$

$$first \leq second$$

Modularity and decomposition are facilitated through *schema inclusion*, by which one schema can be included in another. We can access variables in a schema through the notation *schema_name.variable_name* so that, for example, *Pair.first* refers to the variable *first* in the *Pair* schema.

Operations

In essence, schemas describe the admissible states and the operations of a system, which are defined in terms of *changes to the state*. Specifically, an operation relates variables of the state after the operation (denoted by dashed variables) to the value of the variables before the operation (denoted by undashed variables). Operations may also have inputs (denoted by variables with question marks), outputs (exclamation marks) and preconditions. In the *GettingCloser* schema below, there is an operation with an input variable, $new?$; if $new?$ lies between the

variables *first* and *second*, then the value of *first* is replaced with the value of *new?*. The value of *second* does not change, and the output *old!* is equal to the value of the variable *first* as it was before the operation occurs. The $\Delta Pair$ symbol is an abbreviation for $Pair \land Pair'$ and, as such, includes in this operation schema all the variables and predicates of the *Pair* schema before and after the operation.

GettingCloser

$$
\begin{array}{l}
new? : \mathbb{N} \\
\Delta Pair \\
old! : \mathbb{N} \\
\hline
first \leq new? \\
new? \leq second \\
first' = new? \\
second' = second \\
old! = first
\end{array}
$$

Relations and Functions

To introduce a type in Z when no information about the elements within that type is specified, a *given set* is used. This is an important abstraction mechanism that allows us to model things at the highest possible level. For example, we can write $[TREE]$ to represent the set of all trees without stating anything about the nature of the individual elements within the type. If we wish to state that a variable takes on a value, a set of values, or an ordered pair of values of this type, we write $x : TREE$, $x : \mathbb{P}\, TREE$ and $x : TREE \times TREE$, respectivelly. If we have $xs : TREE \times TREE$, then the expressions *first* xs and *second* xs denote the first and second elements of the ordered pair xs.

Perhaps the most important type is the *relation* type, expressing a mapping between *source* and *target* sets. The type of a relation with source X and target Y is $\mathbb{P}(X \times Y)$, and any element of this type (or *relation*) is simply a set of ordered pairs.

The definition of functions is also standard: relations are functions if no element from the source is related to more than one element in the target set. If every element in the source set is related to one in the target, then the function is *total* (denoted by $\rightarrow$); *partial* functions ($\nrightarrow$) do not relate every source set element. If no two elements in the source relate to the same element in the target set then the function is injective ($\rightarrowtail$). Further, if all elements in the target set are related then the function is surjective ($\twoheadrightarrow$).

Definitions and declarations

a, b	Identifiers
p, q	Predicates
s, t	Sequences
x, y	Expressions
A, B	Sets
R, S	Relations
$d;\ e$	Declarations
$a == x$	Abbreviated definition
$[a]$	Given set
$A ::= b\langle\langle B\rangle\rangle$ $\mid c\langle\langle C\rangle\rangle$	Free type declaration
$\mu\, d \mid P$	Definite description
let $a == x$	Local variable definition

Logic

$\neg\, p$	Logical negation
$p \wedge q$	Logical conjunction
$p \vee q$	Logical disjunction
$p \Rightarrow q$	Logical implication
$p \Leftrightarrow q$	Logical equivalence
$\forall X \bullet q$	Universal quantification
$\exists X \bullet q$	Existential quantification

Sets

$x \in y$	Set membership
$\{\}$	Empty set
$A \subseteq B$	Set inclusion
$\{x, y, \ldots\}$	Set of elements
$(x, y, \ldots)$	Ordered tuple
$A \times B \times \ldots$	Cartesian product
$\mathbb{P}\, A$	Power set
$\mathbb{P}_1\, A$	Non-empty power set
$A \cap B$	Set intersection
$A \cup B$	Set union
$A \setminus B$	Set difference
$\bigcup A$	Generalized union
$\#A$	Size of a finite set
$\{d;\ e \ldots \mid p \bullet x\}$	Set Comprehension

Relations

$A \leftrightarrow B$	Relation
dom R	Relation Domain
ran R	Relation Range

Functions

$A \nrightarrow B$	Partial function
$A \rightarrow B$	Total function

Schema notation

S d p	Schema
	Axiomatic def
S T d p	Inclusion
ΔS S S'	Operation
$z.a$	Component

Conventions

$a?$	Input to an operation
a	State component before operation
a'	State component after operation
S	State schema before operation
S'	State schema after operation
ΔS	Change of state ($S \wedge S'$)
ΞS	No change of state
$OP_1 \fatsemi OP_2$	Operation composition

FIGURE 3.2: Summary of Z notation (taken from [81])

.

The *domain* (dom) of a relation or function is the set of source elements, while the *range* (ran) is the set of target set elements. Examples of these operators can be seen below.

$$\text{dom}\, Fun1 = \{tree1, tree2, tree3\}$$
$$\text{ran}\, Fun1 = \{tree2, tree3\}$$

A summary of the Z notation is shown in Figure 3.2. We will not discuss further details of the language here but instead provide references to some of the many textbooks on the subject [30,

170].

3.4 SMART Agents

With the brief overview of the Z notation in place, we can now present the conceptual infrastructure of SMART, beginning with a discussion of the support provided for the specification of individual agents. The modular approach used throughout SMART means that these concepts form the *foundation* for all subsequent definitions, including agent relationships. This is particularly useful because it is directly inline with our aim of supporting *reusable* agent models. We begin with a detailed presentation of these foundational concepts that define the different types of entities and their relationship to the environment. Subsequently, we discuss SMART's notion of agenthood and introduce a refinement that provides more granularity in the different types of agents that we can define. This refinement is particularly useful for addressing the needs of *practical* construction, and its used in Chapter 4 to ground the abstract notions of SMART to specific agent constructions.

3.4.1 Foundational Concepts

Through SMART, Luck and d'Inverno set out to address the lack of an unambiguous agent theory that could be used to describe and relate existing work in the field, as well as act as the basis through which to develop new work. The use of Z serves to make the work as precise as possible, but the *main* advantage of SMART is that it has steered clear from dependencies on any specific agent architecture and from making any limiting assumptions about the environment or agent societies. This is particularly useful for our work, since we aim to accommodate heterogeneous agent societies, in which a variety of agent architectures, and dynamic environments need to be supported.

At the base of SMART is a view of agents as *entities* attempting to satisfy goals, where goals are desirable states of affairs. Entities are hierarchically organised in four different types, with each type refining the previous one. These entity types are described using three primitive types, *attributes*, *actions* and *motivations*, which are formally represented as given sets with no restrictions on how they could be manifest in a particular system instantiation. A short description of the primitive types follows, before we go on to describe the different entity types.

Primitives

Attributes are perceivable features of the environment and through them, entities and the environment in which they are situated can be described. For example, if we consider a mobile device as an entity, then some of the attributes that can be used to describe it are the name of the owner of the device, the location of the device, and so forth.

$$[Attribute]$$

An environment can then be defined as a non-empty set of attributes.

$$Environment == \mathbb{P}_1 \, Attribute$$

Actions are discrete events that can change the state of the environment. For example, a mobile device can perform actions such as communicating with other devices, storing information, and retrieving online documents.

$$[Action]$$

A goal is a desirable state of affairs in the environment, which is described by a non-empty set of attributes. For example, a goal to find a particular online document can be described as a state of affairs in which the location of the document is known.

$$Goal == \mathbb{P}_1 \, Attribute$$

Finally, a motivation is any *desire* or *preference* that drives an agent to set its own agenda, as opposed to having goals dictated to it by a user or other agents. It is defined as a given set.

$$[Motivation]$$

Entities

The four different *entities* can now be considered using these primitive types. The *Entity* schema below defines an entity to have a set of attributes, a set of actions (their capabilities), a set of goals and a set of motivations. The only restriction for something to be of type *Entity* is that it

must have a non-empty set of attributes, as stated in the predicate part of the schema.

Entity

$$
\begin{array}{|l}
attributes : \mathbb{P}\, Attribute \\
capabilities : \mathbb{P}\, Action \\
goals : \mathbb{P}\, Goal \\
motivations : \mathbb{P}\, Motivation \\
\hline
attributes \neq \{\,\} \\
\hline
\end{array}
$$

Objects are entities that have some capabilities, making it possible for them to perform actions that can change the environment. Thus, the *Object* schema includes the *Entity* schema, and further restricts it by requiring that the set of capabilities is non-empty.

Object

$$
\begin{array}{|l}
Entity \\
\hline
capabilites \neq \{\,\} \\
\hline
\end{array}
$$

Agents are objects that are attempting to achieve goals. This means that there is a desirable state of affairs in the environment that they are attempting to bring about. Correspondingly, the *Agent* schema includes the *Object* schema and constrains the set of goals to be non-empty. Agents can have or be *ascribed* goals that they retain over any instantiation or lifetime. In Sections 3.4.2 and 3.5, we discuss in more detail the issue of how goals can be adopted by, or ascribed to, agents.

Agent

$$
\begin{array}{|l}
Object \\
\hline
goals \neq \{\,\} \\
\hline
\end{array}
$$

The definition of agents given above relies on the existence of other agents to provide the agent's goals or ascribe goals to the agent. This means that some other entity is *always* required to provide or ascribe the goals. In order to ground the entity hierarchy, therefore, entities are required that can *generate* their own goals. These agents are defined as *autonomous* since they are not dependent on the goals of others, and possess goals that are *generated from within* rather than adopted from or ascribed by other agents. Such goals are generated by *motivations*, which *drive* an autonomous agent to generate its own goals and guide it in choosing the goals to adopt

when interacting with other agents. Formally, the *AutonomousAgent* schema below requires the set of motivations to be non-empty. We will not discuss the *generation* of goals by motivated agents, but an extensive analysis is available elsewhere [82].

AutonomousAgent

$Agent$

$motivations \neq \{\}$

In this way, a clear distinction is made between the notions of agents and autonomous agents. Agenthood is ascribed to any entity that acts in order to satisfy some goal, and motivations are required to support the self-generation of goals by agents. The ability of an agent to generate its own goals is what defines it as autonomous. We return to the distinctions between agents and autonomous agents in Section 3.5.

3.4.2 Neutral Objects and Server Agents

The basic framework described above provides the foundational definitions for entities from SMART's point of view. Now, in addition to these basic concepts, SMART also considers how agents, which are not autonomous, are created. In order to achieve this, the basic framework is further refined to accommodate more sophisticated analyses of agent interaction by introducing additional definitions of *neutral objects* as those objects that are not agents, and *server agents* as those agents that are not autonomous.

The relationship between neutral objects and server agents is complementary and dynamic. Neutral objects become server agents when they are given or ascribed goals. Thus, once these goals are achieved, or pursuing them is no longer feasible, the server agent reverts back to a neutral object. This is a significant characteristic of the framework, since it deals with the fundamental issue of instantiating agent entities, and we will focus on it later, when we investigate how it can help to characterise entities within a dynamic and heterogeneous environment. The schemas below formalise these concepts. A *NeutralObject* is an *Object* with empty sets of goals and motivations while a *ServerAgent* is an *Agent* with an empty set of motivations.

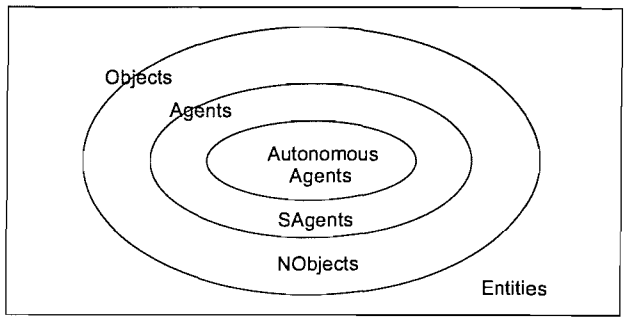

FIGURE 3.3: The entity hierarchy

NeutralObject

Object

$goals = \{\ \}$
$motivations = \{\ \}$

ServerAgent

Agent

$motivations = \{\ \}$

The relationships between all the different entity types are illustrated in Figure 3.3, in which they are shown as a Venn diagram. As indicated, the most general notion of entity subsumes all other notions, while neutral objects (NObjects) and server agents (SAgents) lie in the space between objects and agents, and between agents and autonomous agents, respectively.

3.4.3 The Utility of the SMART Agent Models

Having presented the basic SMART agent models for supporting the description of agents, we now discuss how these models can aid in agent development for dynamic and heterogeneous agent environments. We do this through an example that brings the abstract notions described in Section 3.4.1 and 3.4.2 closer to more practical concerns.

Suppose that you want a train ticket to visit London this weekend, according to some preferences about the trip and ticket price, and are equipped with a personal agent (PA) on your mobile phone that is able to perform the task of finding an appropriate ticket. Now, your PA is an autonomous agent with motivations such as minimising on-line connection time (saving network connection costs), minimising cost for tickets, and providing comfort. Given this task and the

set of motivations that drive its choice over which goals to achieve, it begins the process of accomplishing what is required. The first goal generated is to get a list of all the travel agencies that are able to provide train tickets to London for the weekend. This is achieved by locating a service in the environment with the capability of providing updated lists of such agencies. Now, such services can range from sophisticated brokers to simple registries. For the purposes of this example, we assume that the service is a simple registry, which provides just a basic *query-response* functionality for agents requiring a list of travel agents and a basic *register* functionality for travel agents wishing to advertise. Within the SMART framework, such a simple service can be modeled as a neutral object which, when engaged by the PA, instantiates a server agent with the goal of providing the required list of travel agents. Once the goal is satisfied, the service reverts to a neutral object if it is not being engaged by any other entity.

Having received the list, the PA attempts to contact the travel agencies. The travel agencies themselves can also be a mixture of sophisticated autonomous agents, with motivations, to simple agents that can only identify whether a ticket is available and provide it if the price is paid. Those behaving as *neutral objects* can be accessed directly to instantiate server agents, while for those which are autonomous agents the PA cannot directly access them, but must come to an agreement with them with regard to travel requirements and price. Having achieved the goal (limiting the number of calls so as to satisfy the motivation of minimising on-line time, and purchasing a cheaper ticket according to the other motivations), the PA reports the results and waits for further instructions.

This example illustrates some key points of the SMART framework.

- In heterogeneous computing environments, not all entities are sophisticated agents, and some provide very basic services, such as the registry service. When attempting to model such basic services within the context of a multi-agent application, we require abstractions that will not limit us with regard to the types of agents that we can model. Neutral objects and server agents offer just this capability. By considering a basic service as a server agent just when it is engaged to achieve a goal, we are *reusing* the concepts of agents, goals and relationships between agents, thus retaining the analysis of the system within the SMART framework. Such a capability is key in supporting our aim of *reusable models* for agent infrastructure.

- Similarly, the types of interactions between entities can also take different forms. They could involve one agent having complete control over another, or *cooperation* between

two autonomous agents. The different forms of interactions and the resulting relationships that can be formed in a multi-agent system are discussed by Luck and d'Inverno in [82] and we review them in section 3.6.2. Once more, being able to have these distinctions is crucial in a heterogeneous environment, since these distinctions allow us to more accurately capture the variety of situations that may arise.

- Finally, through motivations, SMART allows us to clearly model the difference between agents that are able to generate and choose their own goals and those that simply attempt to achieve those goals assigned or ascribed to them. This too, allows us to better model the range of situations that can arise, as we have seen in the example.

3.5 Refining SMART: Types for construction

Although SMART provides clear definitions for agents, it does not do so with the required level of granularity to distinguish between extremely simply entities and more complex ones that are not autonomous. However, this granularity is required to provide clear guidelines for developers attempting to model the variety of entities encountered in the types of applications we wish to support. In this section we address this issues by analysing the notion of agenthood within SMART, identifying its shortcomings, and contrasting it with the characteristics of agents identified by Wooldridge and Jennings [232] that is widely used within agent literature. Through the analysis, we indicate the shortcomings of both approaches when taken in isolation, and attempt to provide an explanation that combines the two. We argue that the combined explanation offers a better guide for agent developers, providing a clear understanding of agenthood which, as argued in Chapter 1, is one of the essential stepping stones towards reusable abstractions for agent construction.

3.5.1 Agenthood

SMART aids in describing agent systems by providing a conceptual framework of agent types based on the premise that the defining characteristic of agenthood is the *"doing for someone"*. Here, as long as an entity is fulfilling a goal for another agent, then that entity can be considered as an agent. The advantage of this approach is that it covers any situation because of the clarity of the rule. It is enough to identify whether an entity is satisfying some goal in order to declare it an agent. In fact, Luck and d'Inverno state that it is necessary for *"a viewing agent to analyse*

both the server agent and the agents engaging it in order to avoid conflict" ([81], p.45). When a goal is ascribed to an entity, SMART does not require that goal to be explicitly represented within the entity actually fulfilling the goal but, rather, "*the agentness of the object depends on who is currently viewing the object*"([81], p.25). The implication is that any entity, irrespective of its structure, can potentially be seen as an agent as long as some other agent views it as fulfilling some purpose. Although this helps in analysing the *relationships* between agents and in allowing agents to reason about other agents, it does not help in *constructing* agents, since it says little about the internal structure of agents, other than implying that they must be able to perform their stated capabilities through appropriate mechanisms.

The difficulty of addressing agent construction within the SMART framework is that *agenthood* does not just depend on structural features, but also on the relationships with other agents. In contrast, the widely accepted Wooldridge and Jennings characterisation [232], takes an *action-based* view of agenthood, nominating anything that presents the characteristics of autonomy, pro-activity, reactivity and social ability as an agent. This presents its own problems since the *interpretation* of these characteristics is left open. For example, autonomy can be interpreted both as the ability of an agent to choose and generate its own goals as well as its ability to independently decide which actions to perform towards achieving a *specific goal.*

Neither the action-based view nor the weak structural-based and strong relationship-based views of SMART are sufficient on their own. What we require is an account of agenthood that combines the two, to provide a clearer understanding of the artifacts to be developed when using an agent-oriented approach. Next, we introduce a possible means to combine these views that will provide the necessary understanding of the basic notion of agenthood to enable us to develop an agent construction model.

The first step towards reconciling the two approaches is to clarify SMART'S view of agenthood as the *"doing for someone"*. This view of agenthood is not itself a problem, but it is not *supplemented* by a clear account of the differences, in actual internal structural elements, between very simple objects (with absolutely no reasoning capabilities) satisfying goals, and objects performing more complicated tasks, but with no goal-generation mechanisms of their own. Being able to differentiate between the two at a structural level can provide clarity necessary for a construction model. In this following subsection we distinguish between *passive* agents, which have no reasoning capabilities, and *active agents.* In Chapter 4 we map these concepts to the agent construction model, illustrating how these abstractions are related to the practical construction of

agents. Here, we begin by looking at very simple objects and explain how we can differenentiate them from more complex objects which are not yet autonomous agents.

3.5.2 Passive Agents

Luck and d'Inverno use the extreme example of a teacup acting as an agent for someone who is using it to contain their tea to illustrate the point that even the simplest entity can be considered an agent as long as it is serving a purpose. When a person uses a teacup to hold their tea, the teacup is serving a purpose for its user. In analytical terms, the argument goes, the user has *ascribed* or *imposed* agenthood on the teacup. If someone else passes by and picks up the teacup (e.g., a waiter cleaning tables), the relationship between the teacup and its user is broken and the user's goal can no longer be satisfied. The waiter, on the other hand, does not view the teacup as serving any purpose until he or she is notified by the teacup's user that they have inadvertently interfered with the user's goal. The waiter can then also ascribe agenthood to the teacup since it is serving the purpose of fulfilling a customer's goal (the original user of the teacup). By ascribing agenthood, therefore, relationships can be better understood and formally analysed within a coherent agent-based view.

However, in this scenario, the teacup is completely *passive*. It is simply containing the tea because of its physical make-up, which enables it to contain fluids. The cup is in no way *aware* of the fact that it is satisfying a goal and the cup has no choice as to whether or how it should fulfill the goal. In fact, the greatest part of the job of fulfilling the goal of containing the tea is done by the cup's user. The user identifies that the cup is able to contain fluids, takes care in placing the tea in the teacup and makes sure that the teacup is upright so that the tea does not spill.

3.5.3 Active Agents

A more interesting class of agents includes those that *actively* take part in the achievement of goals. To illustrate this situation we develop a scenario in which both a passive and an active entity are used to achieve similar goals. Imagine that Luc wants to notify a colleague, Mike, about a meeting. Luc uses his wireless device to send a message to Mike about the time and place of the meeting.

In one scenario, illustrated in Figure 3.4, Mike has a relatively simple, limited capability, mobile

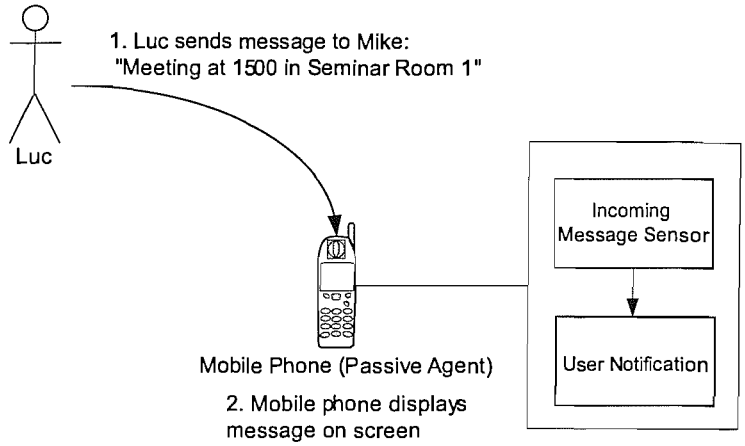

FIGURE 3.4: Passive agent

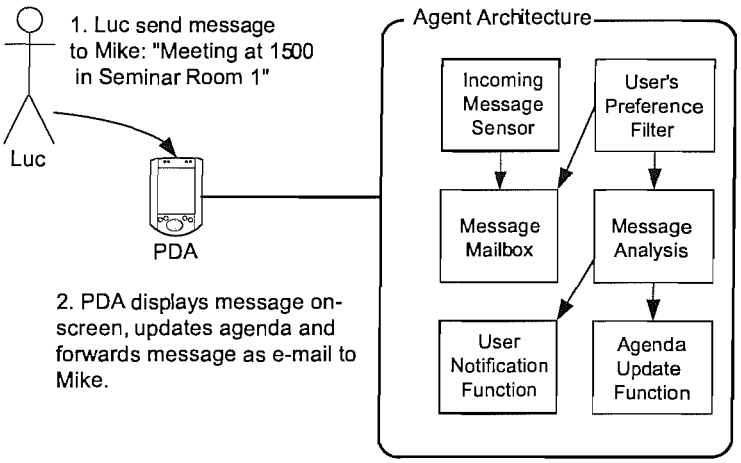

FIGURE 3.5: Active agent

phone. The mobile phone can only receive messages and display them on the screen. Once the message has reached the mobile phone, Luc's device ascribes to the mobile phone the goal of getting the message to Mike. The mobile phone, however, is not aware of this. It simply receives a message and displays it. If Mike happens to see it, then the goal will be accomplished. Thus the mobile phone is acting as a passive agent since it is not actively taking part in the achievement of the goal.

In a second scenario, Mike is equipped with a much more sophisticated wireless Personal Digital Assistant (PDA). Once the PDA receives a notification message from Luc's agent, it is not only ascribed a goal by Luc but it actively takes it upon itself to accomplish the task in a manner it deems appropriate. As shown in Figure 3.5, this may include analysing the message and acting upon it by updating Mike's agenda or taking care to attempt to inform Mike through alternative means such as sending an e-mail to him as well. In other words, the PDA takes some external

description of a goal (*notify Mike about the meeting*) and translates it into a series of actions based on some in-built knowledge and the current state of affairs in the environment, such as Mike's preferences and position. The PDA is an *active* member in the chain of events that cause Luc's goal to be satisfied.

3.5.4 Self-Direction and Autonomy

Both the mobile phone and the PDA are similar in that they have no explicit internal representation of who provided the goal nor that they are in fact satisfying a goal. The crucial difference between them lies at the level of interference from the part of the device towards the achievement of the goal. Although both entities can be considered as agents from the users' perspectives (in our case Luc and Mike) since they are satisfying a goal, we must recognize that their contribution to the goal is very different and thus deserves a distinction at the system design level. Furthermore, we should also recognise that the second example represents just one point on a *scale* of complex behaviour.

In order to distinguish the different types of behaviour, we introduce the term *self-direction*, which implies that an entity can, to a certain extent, direct its actions on its own. A *self-directed agent* is one which, given a goal, uses reasoning capabilities, built-in knowledge, and information about the environment in order to achieve that goal. Note that this is distinct from an *autonomous* agent since the latter is able to *generate* its own goals and decide whether or not to adopt a goal on its own. Of course, once an autonomous agent has decided which goal to achieve, we can discuss the extent to which it is able to take decisions about which actions to perform to achieve that goal. Therefore, an autonomous agent can be self-directed but a self-directed agent is not necessarily autonomous. In their characterisation, Wooldridge and Jennings do not make such distinctions between *self-direction* and autonomy, but choose to use the notion of autonomy without qualifying it further.

Interestingly, both self-direction and autonomy can be understood in terms of the degree to which an agent exhibits the other characteristics, namely reactivity, proactivity and social ability. These distinctions are important in that they allow us to better characterise the different types of agent entities that we construct, something that is particularly useful when attempting to construct agents and analyse the resulting systems. We describe the relationships between reactivity, proactivity and social ability, and self-direction and autonomy below.

Passive Agent	Active (Self-Directed) Agent	Autonomous Agent
No inherent reactivity, proactivity or social ability. Goals *ascribed* to them. Agent status depends on *viewing* agent.	Reactive in adjusting actions to changes in environment, with regards to goals being pursued. Proactive in generating subgoals or choosing actions in order to achieve goal. Socially able in contacting other agents in order to achieve its goals. Goals *assigned* to them.	Can exhibit all the characteristics of an active agent. Reactive in adjusting motivations to changes in the environment. Socially able to contact other agents based on goals generated through motivations. Goals *adopted* or *generated* by agent. Agent status can be verified both through external and internal examination of behaviour.

FIGURE 3.6: Agent characteristics

Self-Direction For self-direction, reactivity can be understood as the ability of an agent to react to changes in the environment while attempting to achieve its goal. For example, Mike's PDA should stop trying to notify Mike about the meeting through different means once it is aware that Mike has read an e-mail about the meeting. Proactivity is the extent to which an agent is able to choose actions and work towards achieving them in order to achieve a primary goal. For example, Mike's PDA is proactive in as much as it performed the actions of updating Mike's agenda, sending an e-mail to Mike, etc. Finally, social ability is the extent to which an agent is able to take advantage of other agents in order to achieve its goals.

Autonomy For autonomy, reactivity can be understood as an agent's ability to adjust its motivations, which may very well lead to alternative decisions about which goals to follow, based on changes in the environment. Proactivity can be understood as an agent's ability to generate goals to satisfy its motivations. Finally, social ability can be seen as an agent's ability to contact and cooperate with other agents in line with its motivations.

Given this interpretation, we can construct a table, as illustrated in Figure 3.6, in which the characteristics of each type of agent entity are listed. It can be used to identify self-directed entities by investigating to what extent they display any of the three components of self-direction, recognising, however, that there cannot be an overriding function, suitable for every case, that will provide the degree of self-direction of an entity. For example, if we consider the mobile phone, one could raise the argument that the mobile phone is indeed actively participating in the achievement of the goal since the message has to go through several stages in being changed from signals received on a radio frequency to text on the screen. The point, however, is not to exhaustively analyse each action an entity performs, but rather to find the *appropriate level*

of abstraction for the application at hand. The mobile phone engineer will certainly not gain anything by saying that the mobile phone is a passive entity. The designer of an agent-based system, however, of which one participating entity is the mobile phone, would be able to better understand the participation of the mobile phone in the overall system design by considering it as an essentially passive entity.

In conclusion, our aim is not to provide a quantitative measure of self-direction but rather to provide a guide to distinguish between passive, self-directed and autonomous agents. This distinction allows us to better reason about the system as a whole and identify aspects that require more attention or are perhaps more prone to fault. For example, when analysing a situation in which both passive and active entities agents take part, a developer could first focus on the active entities and make sure they perform the right tasks. We return to discuss these issues in the next chapter, in which the an agent construction model is introduced, and its relevance to the notion of self-direction is discussed.

3.6 Relationships: SMART to SMART+

Having discussed how SMART describes single agents, we now turn our attention to what SMART has to say about *relationships* between agents. The ability to model relationships is, of course, crucial for multi-agent systems. While the task is challenging even when dealing with static multi-agent systems, where the number of agents and the relationships between them do not change very often, it is even more important in heterogeneous and dynamic environments where the number and type of relationships should be expected to be in continuous flux.

The SMART framework focuses on modeling relationships between agents that arise when one agent seeks aid from another to achieve a goal. Starting from this premise, SMART then provides abstractions for describing the differences between relationships that include a server agent and relationships between autonomous agents. We examine these abstractions here, starting from the definition a *multi-agent* system and then go on to justify the need for describing relationships that are not the result of one agent aiding another, something that SMART does not adequately handle and which we address through SMART+ in Chapter 5.

3.6.1 SMART Relationships

Foundational Concepts

According to SMART, a *multi-agent* system arises from the interaction between two or more agents when at least one is autonomous and is interacting with at least one other agent that is satisfying a goal for the first agent. The *MultiAgentSystem* schema below formalises this definition. A multi-agent system can contain any number and type of entities, with the constraint that at least one is autonomous and there is at least one other entity such that there is some overlap between the goals of the two entities.

MultiAgentSystem

$$
\begin{aligned}
&entities : \mathbb{P}\, Entity \\
&objects : \mathbb{P}\, Object \\
&agents : \mathbb{P}\, Agent \\
&autonomousagents : \mathbb{P}\, AutonomousAgent \\
&neutralobjects : \mathbb{P}\, NeutralObject \\
&serveragents : \mathbb{P}\, ServerAgent \\
\hline
&autonomousagents \subseteq agents \subseteq objects \subseteq entities \\
&agents = autonomousagents \cup serveragents \\
&objects = agents \cup neutralobjects \\
&\#agents \geq 2 \\
&\#autonomousagents \geq 1 \\
&\exists\, aa1, aa2 : agents \bullet aa1.goals \cap aa2.goals \neq \{\,\}
\end{aligned}
$$

An autonomous agent is required because only autonomous agents can generate their own goals, and without the presence of one of them, a multi-agent system would never come into existence. An autonomous agent may not, however, be able to satisfy all of its goals alone and may have to seek assistance elsewhere. It is at this point that it interacts with another entity, with the aim of satisfying a goal, and a multi-agent system is instantiated. The agent seeking other agents to satisfy a goal, the *viewing* agent, must locate a *target* agent to adopt the goal. Through SMART's entity hierarchy, three distinct possibilities arise that are explained below and illustrated in Figure 3.7.

- If the target entity is a neutral object, then the viewing agent can transfer its goal to the neutral object, thus instantiating a server agent.
- If the target entity is a server agent, this implies that it is already satisfying a goal for some

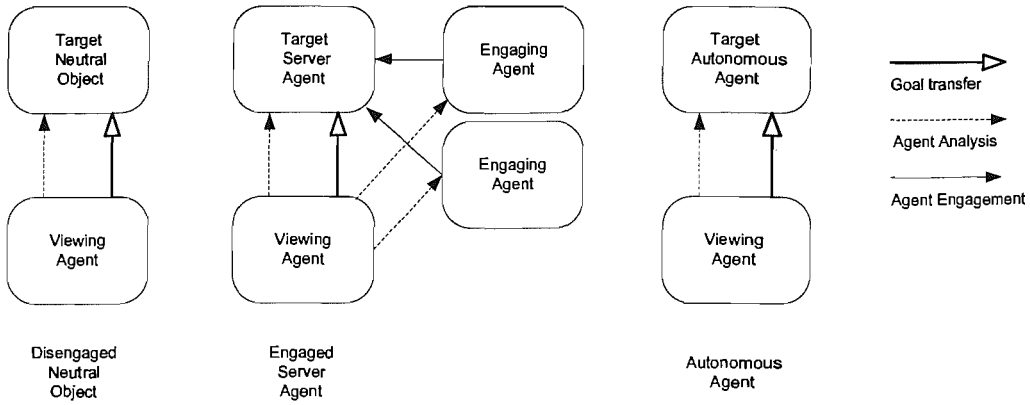

FIGURE 3.7: Goal adoption by neutral objects, server agents and autonomous agents

other entity. The viewing entity must, therefore, analyse both the target entity as well as the agents it is serving before attempting to engage it.

- If the target entity is an autonomous agent, the viewing entity must *persuade* the autonomous agent to adopt its goal. Furthermore, the goal will only be satisfied if it is consistent with the motivations of the agent, which will ultimately determine which goals an autonomous agent can generate.

Engagements and Cooperations

SMART defines interactions between agents and non-autonomous entities as *engagements*, and interactions between autonomous agents as *cooperations*. The difference in terminology illustrates the different nature of the interaction between autonomous agents, since autonomous agents will not interact unless the goal in question is consistent with their motivations, while non-autonomous entities are considered to be predisposed towards satisfying the goals of any engaging entity.

A *direct engagement* is defined to be an engagement between a client and a server agent with respect to a goal., as defined by the *DirectEngagement* schema.

DirectEngagement

$$client : Agent$$
$$server : ServerAgent$$
$$goal : Goal$$

$$client \neq server$$
$$goal \in (client.goals \cap server.goals)$$

In addition, SMART uses the notion of an *engagement chain* to represent the situation in which a single goal generated by an autonomous agent leads to the creation of a series of dependent interactions between server entities, resulting in a chain with the autonomous agent at the head of the chain, and the goal that is being satisfied for that autonomous agent dependent on all members of the chain performing their task. In such engagement chains, the direct engagement between an entity and the next entity further down the chain is a *direct engagement*, while the engagements between server agents that are more than one link apart are *indirect engagements*.[1]

Engagement chains allow a more refined categorisation of the different kinds of relationships between agents. For example, in order to represent the situation in which one agent has complete control over another, SMART defines an *ownership* relationship as one in which *an agent, c, owns another agent, s, if, for every sequence of server-agents in an engagement chain in which s appears, c precedes it, or c is the autonomous client that initiated the chain.* This definition is then specialised into *direct ownership* (when an agent owns another and it directly engages it, *unique ownership* (when an agent directly owns an agent and no other agent is engaging it) and *specific ownership* (when an agent owns another and the owned agent has only one goal).

A cooperation between autonomous agents is modeled as a goal, the autonomous agent that generated the goal, *generatingagent*, and the non-empty set of autonomous agents that adopted the goal, *cooperatingagents*.

Cooperation

$$
\begin{array}{l}
goal : Goal \\
generatingagent : AutonomousAgent \\
cooperatingagents : \mathbb{P}_1\ AutonomousAgents \\
\hline
goal \ni generatingagent.goals \\
\forall\, aa : cooperatingagents \bullet goal \in aa.goals \\
generatingagents \notin cooperatingagents
\end{array}
$$

Using the definitions provided by SMART for describing relationships between agents we can describe a wide range of interaction scenarios. The notion of engagements facilitates the description of scenarios that involve neutral objects while cooperations describe interactions between autonomous agents. Finally, engagement chains allow us to describe the interactions between entities that arise as a result of a goal generated by an autonomous agents.

[1] Note that we do not present *all* formal definitions here, but direct the interested reader to [82], where all formal definitions are presented.

Despite its current expressive capabilities the framework is still limited when attempting to deal with the range of situations that may occur in dynamic, heterogeneous environments. Below, we discuss the problems this limitation creates and how they can be overcome.

3.6.2 Refining SMART Relationships

SMART provides a good starting point for the analysis of interactions between agents that builds upon the basic concepts of entities and goals. As opposed to the issue of agents, where some of the concepts require clarification to suit our purpose at this level there does not seem to be a need for further clarification of *existing* concepts. However, there are some issues which are not covered by SMART, relating to a broader understanding of multi-agent systems within the context of dynamic, heterogeneous agent systems. It is on these issues that we will focus later on in the thesis, but we discuss the limitations of SMART in relation to these here.

The main shortcoming of SMART is that relationships are defined based on agents engaging other agents in order to achieve a goal. The adoption of a goal by an autonomous agent or the ascription of a goal to a neutral object is what leads to a relationship and, as a result, gives rise to a multi-agent system. However, a more *complete* understanding of relationships should take into account relationships that are *not* the result of agents cooperating to achieve goals. For example, when an agent queries another agent about its capabilities the agents are interacting but they are not sharing a common goal. Furthermore, every act of an entity within an environment will, in most cases, lead to the consumption of resources thus affecting other entities in the environment. This can be thought of as a relationship of agents *through* the environment. All these aspects of interactions between agents are not currently handled in SMART and our aim will be to address them by defining appropriate models for *identifying* when such relationships may occur and then *characterising* such relationships. We examine these issues in Chapter 5.

3.7 Conclusions

This chapter sets the scene for the rest of the work described in the thesis by identifying the relationships between the SMART framework, whose concepts underpin all other work presented, and the extensions to SMART we aim to introduce. SMART is particularly suitable because it enables us to model both a wide range of different types of agents, a feature inherent to the kind

of domains for which we wish to support application development, and a variety of ways that these agents can interact with each other based on their types.

At the same time, SMART does not address *all* our concerns, and we have identified in this chapter where it falls short of our aims. This analysis was divided along the lines of support for describing individual agents and support for describing agent relationships.

With respect to individual agents, SMART does not adequately address the issue of agent construction so as to support *practical* development. We aim to address this by an extension of SMART towards a more practical direction in Chapter 4. Furthermore, we argue that some further clarification of the basic agent concepts is required in order to make clear the distinctions between extremely simple entities and more complicated entities that are not autonomous agents. As a result, we introduced the notion of self-direction, which indicates that agents have some degree of freedom in choosing *how* to achieve a goal, as distinct from the notion of autonomy, which indicates that agents have some degree of freedom in choosing *which* goal to achieve. Using this notion we defined *passive* agents as those agents with no self-direction and *active* agents as those agents with some degree of self-direction. These clarifications can aid the development process by enabling the representation of the range of entities that may be encountered and providing indications as to where designers must focus their efforts.

With respect to agent relationships, SMART only addresses the issue of modeling agent relationships that arise as a result of the transfer of a goal or the adoption of a goal. However, in dynamic heterogeneous environments we must be able to model relationships where no goal is shared between the agents, such as the case where one agent *interferes* with the goal of another agent. We aim to address this by an extension of SMART that introduces a more generalised model of agent interaction, which complements the existing abstract models in Chapter 5.

Finally, this chapter demonstrates the utility of adopting existing work of possible, since SMART provides a rich set of concepts to build upon, allowing us to benefit from prior effort, and at the same time the justification for changes or extensions to SMART had to be based on a clear identification and expositions of shortcomings, providing us with a clear argument for the need of further development.

Chapter 4

*act*SMART : Agent Construction Model

"Deliberate before you begin; but, having carefully done so, execute with vigour."

Caius Sallustius Crispus (86BC-35BC); Roman historian

4.1 Introduction

In this chapter we introduce *act*SMART, an agent construction model that allows us to create specifications of agent architectures. The aim is to provide a clear link between the abstractions discussed in the previous chapter and their use in creating reusable models of agents that can be implemented in a practical setting.

Central to the development of agent systems is the architecture of the individual agents themselves. Not surprisingly, a significant amount of effort has gone into developing agent architectures based on a variety of paradigms such as deliberative architectures, reactive architectures and hybrids of the two. However, as discussed in Chapter 1, recent changes in computing environments complicate the task of designing agent architectures, since we need to enable agents to operate in a variety of different environments that may place different and varying demands and limitations on them. This also leads to the need to develop and support a *range* of different agent architectures within a single application, as well as to *modify* an architecture as demands change. Such modifications may need to take place both offline as well as dynamically at runtime, raising three important issues, outlined below.

- First, having to deal with a number of different architectures for a single application can

pose considerable difficulties for application developers. It inevitably increases the complexity of any design, since different architectures may require different types of analysis, and makes it harder to obtain a consistent and coherent view of the system using the same abstractions throughout. For this reason we argue that developers should have access to tools that allow the same types of analysis throughout the application, irrespective of the type of architecture.

- Second, without a consistent way of describing and constructing agent architectures, comparison between them is not facilitated, making it harder to decide which is more suited to a specific task. This has an impact both at the practical application development level as well as at the more general research level. In fact, SMART was in part developed to address this. We aim to extend this analytical capability beyond the level provided by SMART, to focus on the actual *construction* of agents.

- Finally, since we aim to deal with heterogeneous environments, the ability to reuse solutions becomes increasingly important if we are to simplify the task of application development for such environments. The reusability of architectural solutions across domains can be facilitated through a common means of describing and constructing them.

In this chapter, we address just these issues by presenting a model for agent construction that is *conceptually grounded* and *architecturally neutral*. It is conceptually grounded by the understanding of agent systems provided through SMART, and it is architecturally neutral because several different agent architectures can be expressed through the constructs provided. In producing such an agent construction model, we also aim to reconcile research in agent architectures (traditionally situated within the intelligent agent research community) with the demands of software engineers who require flexibility in implementation, coupled with a sound understanding of the underlying principles of agent-based systems. Thus, we enable agent systems application development to adopt a consistent way of constructing a variety of agent architectures.

We begin by outlining the design approach for the development of the agent construction model. The approach is formulated by first defining and justifying the need for four key features that the model should support. In particular, we argue for the need of supporting an *abstract agent model*, *architectural-neutrality*, *modularity*, and the ability to *reconfigure at run-time*. The abstract agent model and architectural-neutrality are, in part, inherently supported by the existing concepts of SMART and the refinements we introduced in Chapter 3. In order to also support

modularity and reconfigurability we, firstly, introduce a distinction between an agent's characteristics (attributes, capabilities, goals, and motivations), structure and behaviour, which enables us to access and modify each separately, and, secondly, use a component-based approach to agent construction. Finally, before presenting *act*SMART in detail we also discuss the relation between *act*SMART, SMART and application development.

With a discussion of our design approach in place, we then provide a detailed description of the agent construction model. In essence, an agent architecture is specified through an agent's attributes, capabilities, goals and motivations (characteristics), the components that make up the architecture (structure), and information and control flow between components (behaviour). These different aspects are managed by an agent *shell*, which allows the developer access to each aspect.

Throughout the discussion, we illustrate various issues by examples. However, we realise that it is difficult to illustrate the totality of the approach without substantial examples, so we provide two more extensive examples in the final section of this chapter. The first example presents a basic architecture for an agent participating in on-line auctions, providing a quick overview of *act*SMART in use. The second example begins with the definition of a negotiating agent architecture, which we then expand to also deal with argumentation. This example illustrates how *act*SMART enables us to reuse architecture designs.

4.2 Design Approach

4.2.1 Desiderata for an Agent Construction Model

In order to address the concerns raised above and provide some statement of requirements for the development of the agent construction model, we identify four desiderata. Although the set is not exhaustive, we consider it to be the minimum necessary set of of requirements.

Abstract Agent Model An agent construction model must be based on some understanding of how we can model agents in a manner that is independent of the agent architecture. This allows the comparison of alternative architectures at this more *abstract* level, ultimately providing application developers with more informed choices as to which architecture is suitable for the domain in question. In our case, the SMART framework provides such an abstract agent model (as discussed in Chapter 3).

Architecturally neutral The construction model should not lead to the construction of only a limited range of agent types, but should allow the widest possible range of architectures to be defined using the same basic concepts. In order to achieve this, there are two possible avenues to explore. One option is to define a generic agent architecture and describe other architectures in terms of this generic architecture. [1] Apart from the inherent difficulty in constructing any general, all inclusive model, the drawback of this approach is that there may be features of other architectures that cannot directly be *translated* to the generic one. The second option is to provide an architecturally-neutral model, so as to avoid this translation problem. Here, the challenge is to provide a model that is specific enough so that it actually offers something to the construction of agents, but general enough to support the development of a wide range of architectures. Through an appropriate *architecturally-neutral* model, we can consider a range of architectures based on a common set of agent-related abstractions and without losing expressive capability.

Modularity The model should allow for modular construction of agents. This is necessary both in order to meet general software engineering concerns and to delineate clearly the different aspects of an architecture. As discussed in the next section, our approach calls for a separation between describing agents in terms of their *characteristics*, their *structure* and their *behaviour*. Such a fine-grained approach can lead to a better understanding of the overall functioning of the agent as well as how it can be altered, since the different aspects of the architecture are clearly identified and the relationships between them made explicit.

Run-time reconfiguration The reality of current computing environments is that changes are often required as the system is running. With large systems that can contain dynamic, complex dependencies between various parts, it is crucial to be able to reconfigure agents at run-time. Reconfiguration may mean providing more functionality to an agent or changing the way it behaves in order to better meet application requirements.

4.2.2 Description, Structure and Behaviour

In the previous chapter, we indicated that while SMART is suitable for *describing* agents, it lacked the necessary features for *constructing* agents. For the purposes of SMART, this was not a problem since the aim was to provide a theoretical framework that would allow the description

[1] Based on Bryson's suggestions, as discussed in Chapter 2.

of a number of different agent systems. However, for our purposes it is crucial to be able to provide tools that facilitate the construction of agent architectures. Nevertheless, we do not want to *replace* the descriptive capabilities of SMART. Rather, we complement them with additional aspects, which are identified below.

SMART allows systems to be specified from an observer's point of view. Agents are described in terms of their attributes, goals and actions, not in terms of how they are built or how they behave. In other words, the focus is on the *what* and not the *why* or *how*. We call this a *descriptive specification*, since it essentially describes a situation without analysing its causes nor the underlying structures that sustain that situation. For example, if we return to the issue of neutral objects becoming server agents when *engaged*, we can see that SMART says nothing about what happens structurally within the entity that has changed status, nor how the mechanisms controlling its behaviour have brought about this change. These are the types of issues we need to address within a construction model. Therefore, *along* with the descriptive specification we need to have the ability to specify systems based on their structure, i.e. the individual building blocks that make up agents, as well as their behaviour. We call these other views the *structural specification* and the *behavioural specification*, respectively.

The structural specification enables the identification of the relevant building blocks or components of an agent architecture. Different sets of building blocks and different ways of connecting them can enable the instantiation of different agent types. By contrast, the behavioural specification of an agent addresses the process through which the agent arrives at such decisions as which actions to perform. Along with the descriptive specification, these specifications provide a more complete picture of the system from different perspectives. It is interesting to note that it is possible to begin from any one of these views and derive the remaining two, but the correspondence is not one-to-one. Several behavioural and structural specifications could satisfy a single descriptive specification and *vice-versa*.

For example, let us consider an agent that is designed with the purpose of participating in auctions in order to buy a specific item. A *descriptive specification* of such an agent may state that the agent belongs to a user, has certain rights with regard to buying items from auctions, is able to keep track of the progress of auctions, has the goal to buy an item of certain quality and at a certain price, and so forth. A *behavioural specification* may state that this agent begins its operation by collecting information about active auctions, then searches for those auctions that have items that fit its specification, and decides which is the more appropriate item before finally

Descriptive Specification	Behavioural Specification	Structural Specification
Attributes: Agent Owner = Ronald Ashri Allowed to Buy = True Capabilites: Search auction sites Buy Items Goals: Get latest Peal Jam CD at lowest possible price	Step 1 : Collect info on active auctions Step 2 : Search for latest Pearl Jam CD Step 3: Evaluate Auctions Step 4: Place Bids Step 5: Buy CD	Active Auction Information Collection Auction Bid Status Component Auction Evaluation Component Bid Placement Component Payment Component

FIGURE 4.1: Distinguishing between description, structure and behaviour

placing a bid. A *structural specification* may state that the agent has different components, each able to handle specific functionalities such as searching for auctions, paying, and so forth. The different aspects are illustrated in Figure 4.1. Alternatively, the structural specification may state that the entire functionality is contained within one tightly integrated control loop. Similarly, the behavioural specification could change to state that the agent searches through auctions and buys the first item that fits the requirements. In both instances, the descriptive specification remains the same, but the structure and behaviour of the agent that fulfill that descriptive specification change.

The agent construction model reflects these levels by allowing direct access to these different aspects of agents, based on a clear decoupling at the architectural level.

4.2.3 Component-Based Construction

In order to support the division of an architecture's different aspects as described above, and to satisfy the requirement for modularity and re-configurability, we take a component-based view of agent architectures.

Component-based software engineering is a relatively new trend in software engineering [53, 83]. Separate developments within the fields of object-oriented computing, re-usable software code, formal methods and modeling languages have all steered research towards a component-based approach [202]. Components are understood as units of composition that can be deployed independently from each other, through a third-party that coordinates their interactions [208]. Interaction with a component takes place through a well-defined interface, which allows the implementation of the component to vary independently of other aspects of the system.

The are several benefits of decomposition through a component-based approach, in line with our

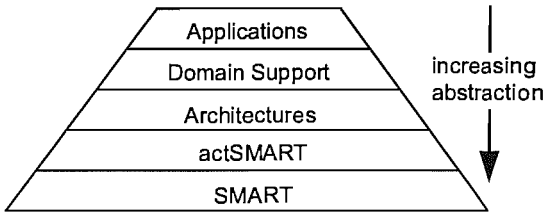

FIGURE 4.2: From SMART to applications

aims.

- Describing an agent architecture through the composition of components promotes a clearer identification of the different functionalities, and allows for their reuse in alternative contexts.
- Different types of components can be composed in a variety of ways to achieve the best results for the architecture at hand.
- By connecting the abstract agent model of SMART to component-based software engineering, we bring it much closer to practical development concerns within a paradigm that is not foreign to developers.

4.2.4 From SMART to Applications

Before proceeding with the description of *act*SMART, we clarify the relationships between *act*SMART, SMART, and the application level. These clarifications serve to indicate how the work presented here can be used within the context of the agent development process.

The relationships are illustrated in Figure 4.2. At the most abstract level lies SMART. Then, *act*SMART represents an extension of SMART to deal with the *construction* of agents. The *act*SMART model makes direct use of the notions of attributes and capabilities from SMART, but to a large extent the notions of components could be used with a different abstract agent model. More direct links between *act*SMART and SMART are made in Section 4.6, where we use *act*SMART to describe the different types of SMART agents possible, as discussed in Chapter 3.

Architectures for agents, which can range from application-independent architectures, such as BDI, to application-specific architectures, can thus be designed using the framework provided by *act*SMART, and based on the concepts provided by SMART. We should note that such application-independent architectures are not always required and may not always be advisable.

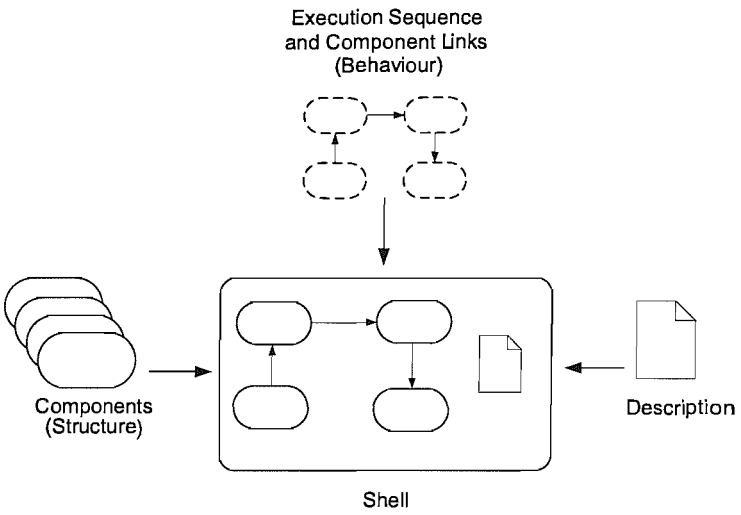

FIGURE 4.3: Agent construction model overview

For example, an agent dedicated to dealing with requests for quotes on fast-changing financial information, where performance optimisation is crucial, would benefit from an application-specific architecture tailored to that situation. Conversely, agents expected to deal with a variety of changing tasks and complex interactions with other agents, such as sophisticated negotiations, might benefit from a more generic and sophisticated deliberative architecture. One of the benefits of our approach is that while it distinguishes between the different cases, it can still consider them within the same conceptual and practical framework.

The next level is domain-specific support, which involves appropriate middleware to support agent discovery and interactions between agents in the specific distributed environments in which the applications operate, as well as other components that could supplement agent capabilities. Finally, specific applications can be built, making use of all the layers below.

4.3 Overview of the Agent Construction Model

In this section, we provide a brief overview of the agent construction model, while in Sections 4.4 and 4.5 we provide a more detailed description of the different aspects within it.

The aim of the agent construction model is to embody all of the design principles discussed above, so as to provide a direct route to implementation. Central to these concerns is the distinction between the structural, behavioural and descriptive specification and a modular, reconfigurable approach. The main concepts and the relationships between them are illustrated in

Figure 4.3, in which the central artifact is the shell that manages an agent architecture, with the architecture being made up of *components*. Components are placed within this shell and the *links* for data-flow between components are defined through the shell. In addition, the *execution sequence* of components is defined by the shell. The components form the structural specification of the agent, while the links and execution sequence define the behavioural specification. Finally, a description of the overall agent is also stored within the shell to complete the descriptive specification of the agent. These features provide for a *modular* architecture with clear distinctions between the different aspects of the architecture.

Now, since individual components are independent of the existence of other components, and all links between them are managed by the shell, we can more easily replace components or change data-flow between components in the shell, as well as alter the execution sequence. These features allow us to *reconfigure* the architecture in response to changing application requirements or changing environmental needs.

Throughout, the main concepts that underpin the development of agent architectures are the abstract agent model provided by SMART, and a functional separation of components into four generic types, described below. The different component types allows us to define architectures without needing to specify the *internal* behaviour of components in great detail. These features support the need for an *architecturally-neutral* model that can be applied in a wide range of situations.

4.4 Components

Components are the basic buildings blocks for an agent, they can be considered as the structural representations of one or more related agent functionalities, which are considered at two different levels. At an *abstract level*, the functionality is described in generic terms, which we will present shortly. At the *implementation level*, the abstract functionality is instantiated through the actual computational mechanisms that support it. The reason for distinguishing between these different levels is so that we can use *generic* component types to specify an agent architecture at a high level of abstraction without making direct reference to the detailed behaviour of each component. This allows us to move between the different levels while retaining a good understanding of the overall architecture, and identifying which specific components best suit each of the generic functionalities.

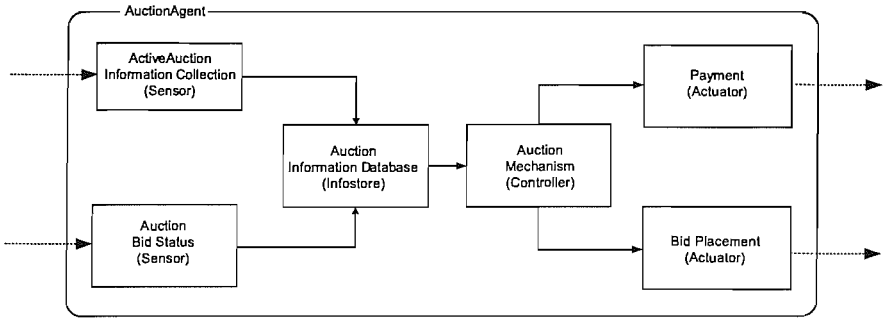

FIGURE 4.4: Example Agent Architecture

4.4.1 Generic Component Types

From here on, we set out the terms that can be used to describe components at an *abstract* level. We begin by dividing components into four generic types, each representing a class of functionality for the agent.

We use the example architecture illustrated in Figure 4.4 to explain each generic component type. The diagram presents a hypothetical architecture for an agent dealing with auctions. Information about ongoing auctions is collected by the *Active Auction Information Collection* component, while the *Auction Bid Status* component provides information about the current state of our bids. All this information is stored in the *Auction Information Database* component, and is processed by the *Auction Mechanism* component to decide where and what bid to place. Bids are placed through the *Bid Placement* component and, eventually, payments can be made through the *Payment* component. The *generic* functionality of the components can be divided into information collection (sensors), information storage (infostores), decision-making (controllers) and finally those directly able to effect change in the environment (actuators). These four generic types of components, described in more detail below, can be used to describe a very wide range of agent architectures, and we will present several examples later on.

- *Controllers* are the main decision-making components in an agent. They analyse information, reach decisions as to what action an agent should perform, and delegate those actions to other components. Controllers are *stateless*, since each decision is taken depending just on information provided through statements at any given execution, and not on previous decisions that a controller has taken. Information that may affect decisions over time should be stored in infostores so that it can be provided to controllers as required.

- *Sensors* are able to sense environmental attributes, such as signals from the user or mes-

sages received from other agents. They provide the means through which the agent gains information from the environment. Similarly to controllers, sensors are stateless.

- *Actuators* cause changes in environmental attributes by performing actions. Actuators are also stateless, since every action they perform is not influenced by previous actions.

- *Infostores* are components whose main task is that of storing information. Such information could be anything from the beliefs of an agent about the world, to plans, to simply a history of the actions an agent has performed, or a representation of its current relationships with others. In contrast to the other components, infostores are not stateless. The information they store represents their current state, and the manner in which information changes will be a result of the manner in which the infostore manipulates and updates information. For example, in the case of a BDI architecture, there may be various ways of representing and updating beliefs, such as dealing only with beliefs referring to the current state of the environment [179].

In conclusion, the main differences between component types are two. Firstly, controllers, sensors and actuators are stateless, while infostores have state. Secondly, only actuators and sensors deal with interaction with the *external* environment, with actuators affecting changes and sensors retrieving information. Finally, we note that while these four types are judged *necessary* for describing any significant architecture we have not encountered an example of an architecture that required more than these four types of components, indicating that they are also *sufficient* for describing the majority of cases.

4.4.2 Component Statements

The *internal* operation and structuring of components, irrespective of their type, is divided into a functionally-specific part and a generic part. In this subsection, we describe the generic part that is common to all components, and outline the types of information that components can exchange.

Each component accepts a predefined set of inputs and produces a predefined set of outputs. A component generates an output either as a direct response to an input from another component, a signal from the environment or an internal event. For example, a sensor component attached to a thermometer may produce an output every five minutes (based on an internal clock), or when

the temperature exceeds a certain level (an external signal), or when requested from another component (as a response to the other component).

In *act*SMART, inputs and outputs share a common structure; they are *statements*, which have a *type* and a *body*. The body carries the main information (e.g., an update from a sensor), while the type indicates how the information in the main body should be treated. We make use of three types of statements, described below.

- INFORM-type statements are used when one component simply passes information to another component. In order for one component to inform another of something, it must be able to produce the INFORM-type statement as an output and the other must be able to accept it as an input. Returning to the example auction agent architecture, the *Auction-BidStatus* sensor would create INFORM statements to be sent to the *AuctionInformation-Database* infostore.

- REQUEST-type statements are used when one component requires a reply from another component. In this case, the receiving component processes the REQUEST and produces an INFORM statement that is sent to the requesting component. The mechanisms through which statements are transmitted from one component to an other are introduced in Section 4.5. Once more referring to the auction agent architecture, the *Auction Mechanisms* controller could produce REQUEST statements to be sent to the *Auction Information Database* infostore, which can then reply with an INFORM statement.

- EXECUTE-type statements are used to instruct another component to execute a specific action. Typically, controller components send such statements to actuators so that changes can be effected in the environment. In the auction agent example, the *Auction Mechanism* controller could create EXECUTE statements to be sent to the *Payment* actuator, instructing it to pay for an item won in the auction.

It should be noted that this list of statement types is not exhaustive, and they are simply representative of the needs of most applications due to their generic nature. Some domains may benefit from more specific statement types. We should also add that message-passing between components at this level should not be compared with message exchange as defined in high-level agent languages such as KQML/FIPA [95]. Typing statements simply provides additional information to aid control of component behaviour.

The information within a statement's body is, in its most general form, described through *attributes*, as per the definitions given in Section 3.4.1. For the purpose of practicality we divide attributes along the lines of *architecture-specific* attributes and *domain-specific* attributes. Architecture-specific attributes are attributes that are only relevant within the internal scope of an agent architecture. For example, a BDI-based architecture could define attributes such as *plans*, *beliefs*, *intentions* and so forth. [2] Architecture-specific attributes can be considered as defining the *internal* environment of an agent. Domain-specific attributes define features that are relevant to the environment within which the agent is operating. So, in the case of the agent example above, these attributes may include features such as *auction-house name*, *item*, and so forth. Application-independent agent architectures, such as BDI-based architectures, typically make use of both types of attributes, including domain specific attributes *within* the architecture-specific attributes. Thus, a *plan* may prescribe an action to contact a service, as identified by its *service name*. The components of an AgentSpeak(L) [79] architecture, for example, could then manipulate *plans* and *beliefs*, and have some generic way of manipulating the *domain-specific* attributes. However, a developer may also choose to develop an agent that has no architecture-specific attributes, creating components that can directly manipulate domain-specific attributes. We return to examine this issue in Chapter 6. Below, we describe a typical operation cycle for a component to explain how the different types of statements are handled.

4.4.3 Component Operation

An outline of the component operation is shown in Figure 4.5. Components begin their operation in an inactive state within the shell. In this state they do not receive or send statements. Once activated by the shell, components perform any relevant initialisation procedures and then can enter one of two possible types of operation. The default type is to receive statements until the shell calls them to enter their execution phase. An alternative behaviour is for the receipt of a statement to *trigger* their execution phase. Below we consider the default operation first, before discussing the alternative behaviour.

When a statement is received, it is typically stored within the component until the component enters its component-execution phase. Once the shell directs the component to enter its execution phase, all statements received by a component are processed. According to the type of statements received, the component will do one of three actions, described below.

[2] This approach was followed by d'Inverno and Luck when formalising AgentSpeak(L) [79]

```
while active and not executing do
        listen for statements

        if statements received then
                store statements

        if call to execute then
                retrieve stored statements
        while stored statements not empty do
                if INFORM then
                        update relevant attributes
                if REQUEST then
                        retrieve relevant attributes
                        create INFORM statement
                        push statement to outbound stack
                if EXECUTE then
                        push statement to execution stack

        pop statements from execution stack and perform actions

        send statements to other components
```

FIGURE 4.5: Component Lifecycle

INFORM An INFORM statement simply causes the component to update any relevant attributes, based on the information contained within the statement.

EXECUTE An EXECUTE statement is placed on an *execution stack*. Once the processing of all received statements is completed, the EXECUTE statements are retrieved and the component performs the actions described within the statement.

REQUEST A REQUEST statement causes the component to attempt to retrieve the information requested and create an INFORM statement that contains that information. This INFORM statement is then placed in an *outbound* stack that stores all statements to be sent out. All outbound statements are sent once the processing of all received statements has finished and the actions prescribed by EXECUTE statements have been performed.

The entire process continues until a component is deactivated. Note that while a component is executing it cannot receive any statements. If statements are still arriving at the component, it is the task to the shell to manage those statements until the component is able to receive them.

The alternative behaviour for the component is to process every statement as it arrives, using the same process we described above for the different types of statements. This *event-based* behaviour is especially useful for infostores which are typically queried with REQUEST statements for information, so that they can provide the response immediately. Through the implementation of *act*SMART in Chapter 6, we will see some examples of combining components that execute

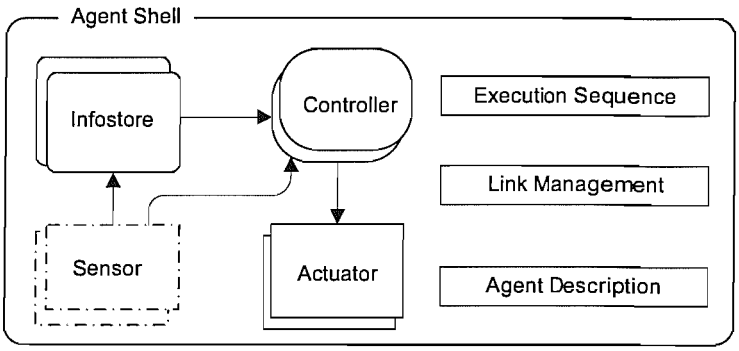

FIGURE 4.6: Agent shell

whenever they receive a statement and components that await the command to execute by the shell.

At any given time, the state of a component in terms of the information to be manipulated, is given by the set of statements that have not yet been processed, the set of statements in the execution stack, the set of statements in the outbound stack and any attributes that the component manipulates. Depending on the specific implementation of a component, it may be possible to interrogate components for their individual states.

With components, we are able to differentiate between the different tasks an agent architecture needs to perform, from a structural perspective. In contrast, the composition of components and the control of the flow of information between them provides the behavioural specification. In the next section, we see how this is managed.

4.5 Shell

As discussed in Section 4.2.3, the basic principles of a component-based approach is that components should be independent of each other, and the coordination between them should be handled by a third-party. As we have seen from the description of components in Section 4.4, communication between components takes place through the exchange of statements. Individual components are not aware of the origin of received statements nor the destination of statements they produce, ensuring that components are independent of each other. Third-party coordination is achieved by placing components within a *shell*, which acts as the *third party* that manages the sequence in which components execute and the flow of information between components. This management takes place by defining *links* between components and the *execution sequence* of

components. The basic aspects of a shell are illustrated in Figure 4.6. From this point on, we will use different representations for the different types of components in order to aid the illustration of agent architectures. *Sensors* are dashed rectangles, *infostores* are rounded corner rectangles, *actuators* are continuous line rectangles, and *controllers* are accented rounded corner rectangles. Components are placed within a shell, links are created between components to allow the flow of statements, and an execution sequence is defined. In addition, the shell can be used to maintain *descriptions* of agents in terms of attributes, capabilities and goals. We consider each of these aspects in more detail below.

4.5.1 Links

Information flows through *links* that the shell establishes between components. Each link contains *paths* from a *statement-producing* component to the *statement-receiving* components. Each component that produces statements has a link associated to it that defines the components that should receive those statements. Links also ensure that, in the case of a REQUEST statement, the reply is sent to the component that produced the request. Thus, links manage paths, which are one-to-one relationships between components. They are usually unidirectional, except in the case of a REQUEST statement, for which an INFORM may be returned in the opposite direction.

The shell then uses the information within links to coordinate the flow of statements between components. Ultimately, this coordination depends on the choices that a developer makes, since it requires knowledge of each component and how they can be composed.

By decoupling the handling of statements between components from the components themselves, we gain considerable flexibility. We can manage the composition of components and the flow of information without the components themselves needing to be aware of each other. It is the architecture developer's task to ensure that the appropriate links are in place. At the same time, we have flexibility in altering links, and it becomes easier to introduce new components. Furthermore, basic transformations can be performed on a statement from one component to the other to ensure compatibility if the output of one component does not exactly match the required input for another. For example, if a sensor component provides information from a thermometer based on the Celsius scale, while a controller that uses that information makes use of the Fahrenheit scale, the link can be programmed to perform the necessary transformation. These features satisfy our requirement for facilitating the reconfiguration of architectures.

4.5.2 Execution Sequence

Apart from the management of the flow of information, we also need to consider the execution of components for a complete view of agent behaviour. This is defined via an *execution sequence* that is managed by the shell. Execution of a component includes the processing of statements received, the dispatch of statements, and the performance of any other actions that are required. The execution sequence is an essential part of most agent architectures and, by placing the responsibility of managing the sequence within the shell, we can easily reconfigure it at any point during the operation of the agent. For many architectures this may be purely sequential, but there are cases in which concurrent execution of components is desired (e.g., the DECAF architecture is based on a fully concurrent execution of *all* components [105]). In general, the issue of supporting complex execution sequence constructs, such as conditional paths and loops, is considered to be an issue that goes beyond the scope of this research, and there is a wealth of existing research that can be accessed to address this need. For example, recent developments within the field of Semantic Web Services provides a process model language for describing the operation of a web service [4]. Nevertheless, through our proposed mechanisms, we facilitate the necessary separation of concerns to enable the integration of such work within the scope of agent architecture development. The architectures developed in this thesis do not make any use of concurrent execution of components.

4.5.3 Agent Description

The description of the agent as a whole can be maintained by the shell or explicitly within the agent architecture, with components dedicated to the task, depending on the capabilities and needs of the architecture. In the former case, the shell can store a number of attributes that describe the agent owner, its location, user preferences, etc. The level of detail covered by this description is mostly an application-specific issue, and this information can either be provided directly to the shell by the developer, or collected from the various components. The shell could query a component that is able to provide information about the current location, for example, and add that to the description of the whole agent. Likewise, it may keep a record of the current goal an agent is trying to satisfy, or the plan it is pursuing. The capability to collect and provide attributes describing the agent within the shell may be particularly useful in a situation in which a developer wants to export a view of the agent for debugging purposes, or when some information needs to be advertised, to facilitate discovery by other agents.

4.5.4 Agent Design

With the main aspects of the agent construction model in place, we now briefly describe the agent *design process*. Agent design begins with an empty shell. We could envisage implementations of shells being provided by environment owners, which would ensure compatibility with their environment, while allowing the agent developer relative freedom as to the structure and behaviour of the agent within the confines of the shell. Then, based on the purpose of the agent, the necessary components for sensing, acting, decision-making (controllers) and information storage can be identified. If such components already exist, the main task of the developer is to decide on the desired behaviour, in terms of execution sequence and flow of information, and whether any of the outputs of components need to be transformed in order to be aligned with the input needs of other components.

The components are then loaded into the shell, and links, as well as an execution sequence, can be defined. With the execution sequence in place, the operational cycle of the agent can begin. Agent operation can be suspended or stopped by stopping the execution sequence. This operational cycle can be modified by altering the execution sequence, and modifying links between components.

4.6 Linking *act*SMART to SMART

With *act*SMART we can describe several agent architectures through a component-based approach, since it provides a sufficient level of detail to support the implementation of arbitrary architectures. This allows us to describe architectures at a level that is close to implementation, while retaining the ability to abstract out some of the details, such as the specific operation of each component.

However, up to this point we have made no *direct* connections between *act*SMART and SMART. To a certain extent, the connections are implicit, since the notions of sensors and actuators clearly model the view of an agent as an entity interacting with the environment. In addition, the units of information within an agent architecture are attributes, as defined by SMART. Nevertheless, beyond these notions, *act*SMART could be used without reference to the abstract agent model provided by SMART. In this section, we explicate the correspondence between the two, to illustrate both the utility of an abstract agent model as well as the ability to refer to the construction

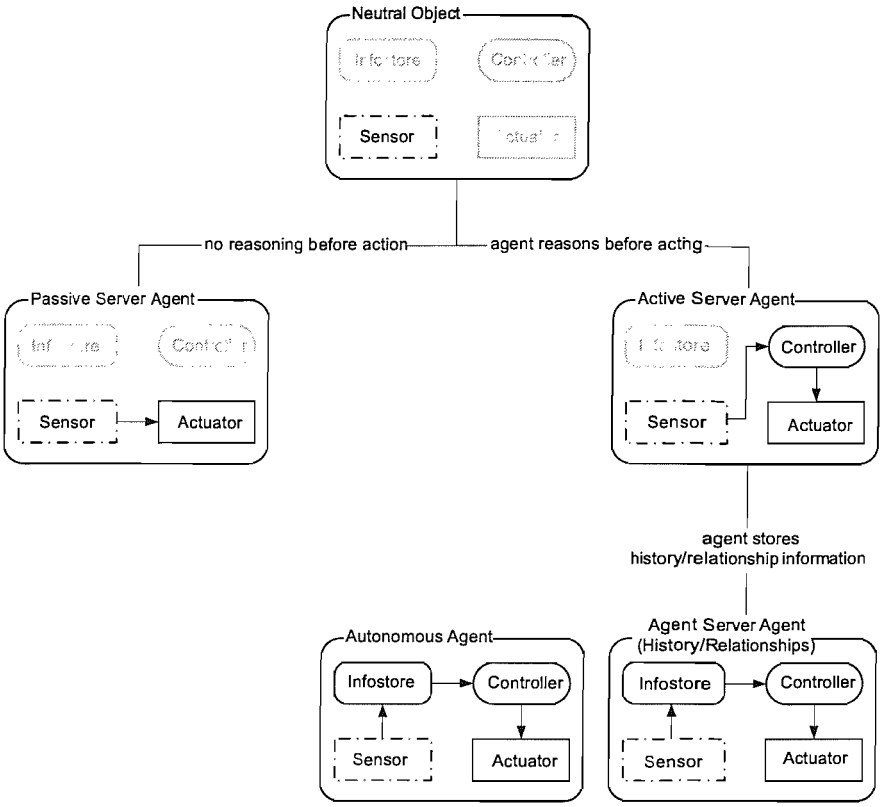

FIGURE 4.7: Entity hierarchy integrating structural and behavioural elements

of agents. We do this by using the agent construction model to provide an alternative view of the SMART entity hierarchy, which *combines* description with structure and behaviour, defining minimal architectures for the different types of entities in the entity hierarchy.

Recall that the entity hierarchy is made up of four basic types: entities, which are described by attributes; objects, which are entities that are able to perform actions; agents, which are objects with goals; and, finally, autonomous agents, which are agents able to generate their own goals. In addition, neutral objects are those objects that are not agents, while server agents are those agents that are not autonomous. Finally, we refine this hierarchy by introducing passive agents as those agents with no self-direction, and active agents as those agents with self-direction.

An agent application may contain entities that are *only* a set of attributes, with no capabilities or goals. However, from a combined structural-behavioural perspective it only makes sense to describe minimal architectures from the level of neutral objects onwards, which is the level where objects can actually perform actions and therefore have some well-defined structure and behaviour. Now, depending on the type of neutral object, its instantiation may lead to either a *passive* or *active* agent, and we provide minimal architectures for those as well. The combined

structural-behavioural perspective allow us to more clearly define the notion of self-direction since we can ground it to the existence or not of specific types of components. In addition, we also discuss the difference between agent architectures that makes use of an infostore. Finally, we provide a minimal architecture for an autonomous agent. Figure 4.7 illustrates the different architectures as well as the relationships between them. We describe each agent type, in turn, below.

Neutral Object All of the components of a neutral object are deactivated, except sensors. Sensors are required to receive information from the environment, so that a neutral object can respond to any requests. It is important to note that neutral objects can only perform an action if another agent sends a message for an action to be performed, or if the sensor is somehow activated by changes in the environment to cause an action to take place.

Passive Server Agent If a neutral object performs an action due to a message that comes through the sensor and *directly* causes an actuator to execute, the neutral object behaves as a *passive server agent*. In other words it has no self-direction. As a result, the *minimal* architecture for a passive server agent must include at least one sensor component and one actuator component with a link between them, where the output statement of the sensor is an EXECUTE statement for the actuator.

Active Server Agent When information from the environment is passed through a controller, which analyses and takes decisions based on it, the entity is behaving as a *active server agent*. This agent has some degree of self-direction, which is expressed through the controller. Therefore, a minimal architecture for an active server agent must include at least one sensor, one actuator and one controller.

Active Server Agent (History/Relationships) The minimal architecture for an active server agent described above did not make any use of infostores. However, an important type of agents, discussed both by Luck and d'Inverno [81] and by Russel and Norvig [186], relates to agents that store information about their past actions, the environment or relationships with other agents. Such agents are necessary to be able to perform long-term reasoning. In *act*SMART a minimal architecture for such an agent simply requires all four types of components.

Autonomous Agents Finally, autonomous agents must also have all four types of components in order to operate. Autonomous agents generate their own goals based on motivations,

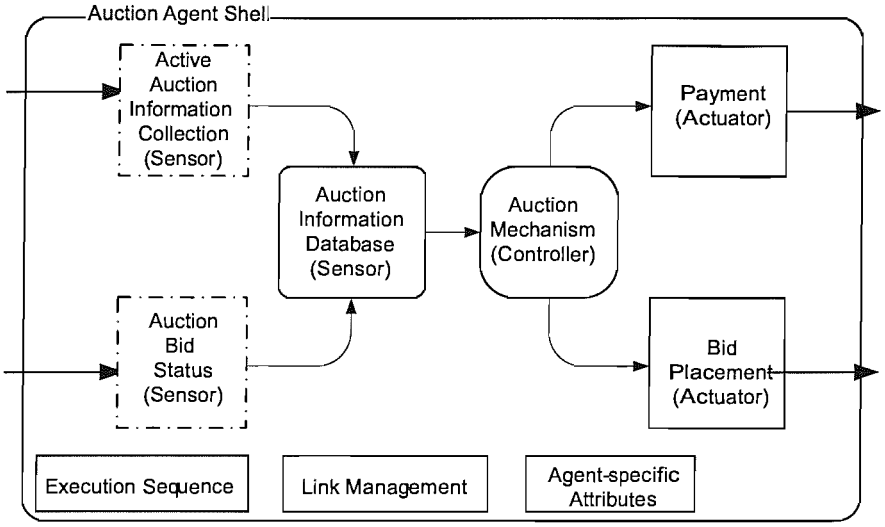

FIGURE 4.8: Example auction agent architecture

but in order to do that they need to have some understanding of how a goal benefits a motivation. This means that they need to be able to reason about the current environmental state and how it can be changed to achieve a goal that is in line with their motivations. As a result, they must be able to store information about the environment so that the controller can make use of it.

Through these distinctions, we can see that the main difference between passive and active agents is the use of controllers to take decisions. From this perspective, controllers can be said to encapsulate the self-direction abilities of the entity as a whole.

In the next sections we present two examples, of increasing complexity, that illustrate the use of *act*SMART.

4.7 Example Architecture: Auction Agent

The architecture illustrated in Figure 4.8 provides a straightforward example of how the agent construction model can be used, and at the same time illustrates how access to an underlying abstract model can be beneficial when dealing with architectures that do not have explicit representations for goals. The architecture represents an agent exclusively aiming to participate in auctions.

First, the components are linked as illustrated in Figure 4.8. The sensors wrap their information

within INFORM statements so as to send them to the infostore, which analyses them to decide how the information should be stored. For example, the attributes within the statement could indicate that the auction status should not be considered reliable after a certain time period has elapsed, or that the level of reliability of the information source is poor. The *Auction Mechanism* is then informed of the current situation as perceived by the sensors and by the *Auction Information Database*. Based on this information, the *Auction Mechanism* controller sends EXECUTE statements to the actuators, indicating which action to perform, and under which conditions that action should be performed. For example, if an actuator fails to place a bid before a certain time period, it should quit.

Note that there are no explicit references to goals in this architecture. However, at the level of SMART we can reason about the goals of achieving an appropriate price given the mechanisms defined within the *Action Mechanism* controller. We can also identify and reason about the agent's attributes (contained within individual components) and capabilities (as expressed through individual components) at this level and compare it with other agents. This is one of the benefits of having access to an abstract agent model. Further benefits come when placing this agent in the context of a multi-agent system whithin which we are able to reason about relationships with other agents based on the different agent types that SMART defines [139].

4.8 Example Architectures: Negotiating Agents

In this section we present a more complex example, investigating the suitability of our model for specifying flexible negotiating agent architectures. This is a class of agent architectures that is gaining increasing importance for a variety of application settings. We briefly discuss negotiating agents below.

4.8.1 Negotiating Agents

In multi-agent environments, agents often need to interact in order to achieve their objectives or improve their performance. One type of interaction that is gaining increasing interest is *negotiation*. We adopt the following definition of negotiation that reconciles views proposed by [121] and [222], which we believe is a reasonable generalisation of both the explicit and implicit definitions in the literature.

> *Negotiation is a form of interaction in which a group of agents, with conflicting interests and a desire to cooperate, try to come to a mutually acceptable agreement on the division of scarce resources.*

Agents typically have conflicting interests when they have competing claims on scarce resources, which means that their claims cannot be simultaneously satisfied. Resources here are taken to be very general; they can be commodities, services, time, and so forth, that are needed to achieve something.

To resolve such conflicting interests, a number of interaction and decision mechanisms have been developed [121], and there has been extensive work on implementing frameworks of negotiation, based on auction mechanisms (as evident, for example, in the Trading Agent Competition [209]) and frameworks that adopt heuristic-based bilateral offer exchange (e.g. [90, 91]). Although recently, argumentation-based approaches [122, 166, 172] have also been gaining interest, there are as yet very few *implemented* systems that cater for argumentative agents. One of the reasons for this is that many of these frameworks involve complex systems of reasoning, based on logical theories of argumentation, for which there are still many open research questions [171]. Another reason is that there are no software engineering methodologies that structure the process of designing and implementing such systems. This is why, in most cases, these systems are implemented in an *ad hoc* fashion.

Our aim is to address the software engineering issues related to the development of architectures for negotiating agents, ranging from simple *classical* agents to more complex *argumentative* negotiators. We use *act*SMART in conjunction with a general negotiation framework to design and describe the architectures of two generic classes of negotiating agents: *simple negotiators* and *argumentative negotiators*. Through this, we demonstrate how a generic architecture for argumentative negotiators can be achieved by extending the simple negotiator architecture and reusing its components, and show how this modularity is facilitated by the construction model.

4.8.2 Negotiation protocol

Before we start describing negotiating agents, however, we discuss the main components of a *negotiation framework*. In addition to the negotiating agents, a negotiation framework usually includes a communication language and an interaction protocol. For example, a negotiation framework based on a simple English Auction protocol would need a communication

language locution (or performative), say $propose(.)$, that can express bids. The protocol is the set of rules that specify, at each stage of the interaction, which locutions can be made, and by whom. In addition, the framework needs a language for representing information about the world, such as agents, agreements, arguments, and so on. This information is used within the communication language locutions to form utterances. For example, a bid might be presented as $propose(a, b, \{toyota, \$10K\})$, where a and b are the sending and receiving agents, and $\{toyota, \$10K\}$ is the specification of the proposal. Finally, a negotiation framework usually includes several information stores needed to keep track of various information during the interaction. This information may include proposals made by different agents, concessions they have committed to [222], and so on. Finally, the framework also needs a set of additional non-protocol rules, such as those that identify the winner in a particular negotiation, or those that specify that agents cannot retract their previous proposals, and so on.

Here, we focus our attention on the *construction of the agents* within the framework. We do not address, for example, how protocols can be specified in a modular fashion (which has been investigated in [15] for example), or how the locutions can be verified. We assume that developers have at their disposal definitions of the appropriate negotiating protocols, domain ontologies and communication languages, and instead deal with the problem of framing such mechanisms within an appropriate agent architecture. Note that we do not claim to have specified the *only* way of describing negotiating agents. Instead, we attempt to illustrate how *act*SMART can be used to capture a variety of negotiators.

4.8.3 Basic Negotiating Agent

Basic negotiating agents include those participating in auctions or those engaged in bilateral offer exchanges. The common aspect of these agents is that they engage in interactions in which the primary type of information exchanged between agents is proposals (i.e., potential agreements). We call the agents *basic* in order to distinguish them from agents that can engage in more sophisticated forms of negotiation that allow the exchange of meta-information (or arguments).

Now, in order to illustrate the use of *act*SMART for designing such an agent we present the design through a two-step process. In the first step, we define high-level descriptive, behavioural and structural specifications with the emphasis on the descriptive specification. Using these, we then proceed to identify, in the second step, the precise components, links and execution sequence

Descriptive Specification	Behavioural Specification	Structural Specification
Attributes Domain-Specific Attributes : Negotiation Protocols Handled Negotiation Issues Agent Owner Ontologies Used Architecture-Specific Attributes: Proposals Protocol Rules Mental Attitudes Opponent Model Environment Model **Capabilities** Interpret proposals Interpret negotiation protocols Maintain beliefs Create counter-proposals **Goals** Achieve desired negotiation outcome	Receive Proposal Analyse proposal against protocol Analyse proposal content Update beliefs Generate counter-proposal	Message Interpretation Negotiation protocol analysis Proposal content analysis Environment model infostore Opponent model infostore Mental attidute infostore Response generation

FIGURE 4.9: Initial specification of basic negotiation agent

that we require.

Basic Negotiating Agent: Initial Design

An overview of the initial design of the agent is discussed below, and summarised in Figure 4.9.

Descriptive Specification Recall that the descriptive specification of an agent is based on its attributes, capabilities, goals and motivations.

Domain-specific attributes The domain-specific attributes of the agent include the types of negotiation protocols in which the agent can participate, the types of issues over which it can negotiate (e.g. if it is a seller, the goods it can sell and whether it can negotiate over the price or other features of the goods), information about the owner of the agent, the types of ontologies it uses to describe issues, and so forth. All this is information that other agents can use to decide whether and how to interact with this agent.

Now, a developer can decide to make this information *explicit* within the agent architecture (by providing components that can directly manipulate the information). Alternatively, it may simply provide the information as additional descriptions through

the shell, while it is *implicit* within the architecture in the way components are implemented. The choice between them depends on whether the agent architecture actually needs to manipulate this information directly or not. For example, the application may require that the agent should be able to take some decisions about the type of information it provides about itself to other agents based on changes in the environment, and that such decisions should be made within the components that make up the agent architecture. In our case, we design an agent that does not manipulate this information explicitly, so it is simply provided in the agent shell. The implication is that we do not require components that are dedicated to just handling this information.

Architecture-specific attributes The architecture-specific attributes represent the types of information that the agent architecture components manipulate and use within statements transferred between components. In our case, this includes: *proposals* from other agents; *protocol rules* to enable the agent to decide on the valid responses; *beliefs* relating to the opponent; the agent's mental attitudes and the environment; *proposal content*; and *proposal evaluation*.

Capabilities The capabilities of the agent should include the ability to: interpret proposals from other agents; analyse those proposals based on a set of protocol rules and the beliefs it has about its mental attitudes, its opponent and the environment; maintain and update beliefs based on the interactions it has with opponents; and, finally, create responses to proposals.

Goals The goals of the agent – i.e. the desired negotiation outcomes – are in part influenced by the mental attitudes of the agent. However, the architecture does not require explicit representation of agent goals. Within the descriptive specification, we can consider that the overarching goal of the agent is the achievement of the environmental state that represents the desired negotiation outcome for the agent. This desired state is determined by the mechanisms used to evaluate proposals and generate responses, which ultimately decide when this environmental state has been reached. Here we see, once more, how access to a general, architecturally-neutral agent model allows us to reason about such things as goals even though they find no explicit representation in the architecture.

Structural Specification The structural specification of the agent should include components for: handling the interpretation of messages; analysis of negotiation protocols; analysis of

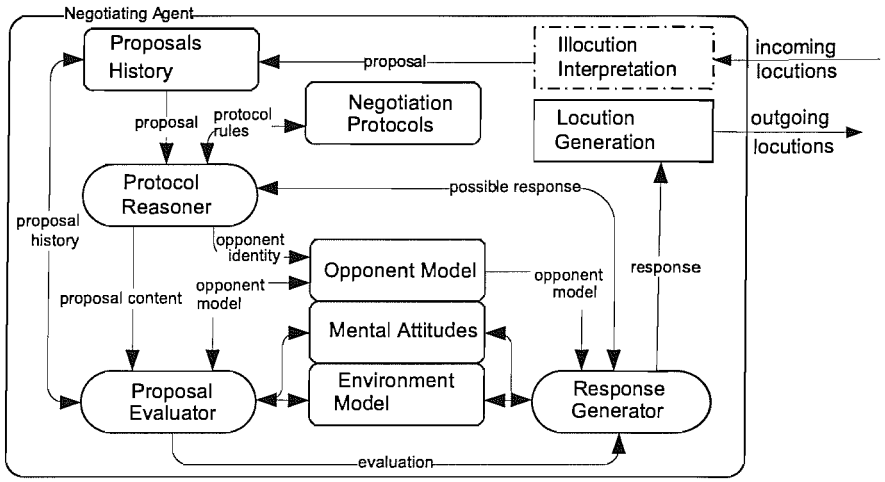

FIGURE 4.10: Basic negotiating agent architecture

proposal content; maintenance and updating of environmental models; opponent models and the mental attitudes of the agent; and the creation of counter-proposal messages.

Behavioural Specification The behavioural specification of the agent begins with the agent receiving some proposal from another agent. Then, the agent needs to analyse the proposal in order to determine whether it is valid given the interaction protocol that is being used for the negotiation. If it is valid, the agent can consider the actual content of the proposal and evaluate it given its own set of requirements for the negotiation outcome. With this evaluation it can then attempt to generate a response, within the constrains of the responses that are allowed given the interaction protocol, and finally sent out a message with the response.

Basic Negotiating Agent: Detailed Design

Using the initial specification provided above, we can now present a more detailed design for the agent, focusing in particular on the behavioural and structural specification.

The proposed architecture for basic negotiation agents is illustrated in Figure 4.10, in which we follow the conventions described earlier for illustrating the different types of components; the connecting arrows illustrate the flow of statements. The structural and behavioural specifications are detailed below.

Structural Specification The structural specification is divided into the four types of components.

Sensors The architecture has just one sensor to receive messages from other agents. The *Illocution Interpretation* sensor is responsible for interfacing with lower-level communication middleware, ensuring that incoming messages adhere to basic syntactic validity and extracting the actual proposal from the message. It can create INFORM statements with the proposal to provide to other components.

Actuators Only one actuator is needed for the architecture to handle the expedition of messages to other agents. Similarly to the *Illocution Interpretation* component, the *Locution Generation* component is responsible for interfacing with communication middleware and packaging proposals appropriately. The component is able to accept EXECUTE statements, with the content being the response it should send to other agents.

Infostores There are five infostores defined for the basic negotiating agent architecture. We examine each in turn below.

Proposals History The *Proposals History* component maintains a history of proposals. It can accept INFORM statements with proposals it should store and it can reply to REQUEST statements for providing proposals to components. The REQUEST statement defines which proposals the component should retrieve based on the opponent's identity and the number of previous proposals that are required.

Opponent Model The *Opponent Model* maintains models of the opponents the agent interacts with. It can accept INFORM statements to update models, and reply to REQUEST statements to provide information about opponents.

Mental Attitudes The *Mental Attitudes* component maintains information about the mental attitudes of the agent, which ultimately influence the agent's evaluation of, and response to, proposals. This infostore can accept INFORM statements to update its mental attitudes and can reply to REQUEST statements to provide information about them.

Environment Model The *Environment Model* component maintains information about the state of the environment as a whole. Similarly to the other infostore components, it updates this information via INFORM statements and provides information via REQUEST statements.

Negotiation Protocols The *Negotiation Protocols* component maintains information about the various negotiation protocols that the agent can participate in.

The component can reply to REQUEST statements in order to provide information about a protocol.

Controllers The architecture uses three controllers, described below.

Protocol Reasoner The *Protocol Reasoner* component reasons about the validity of incoming proposals against a negotiation protocol and identifies the valid responses given the current proposal. Furthermore, the component can dissect a proposal to extract information such as the originator of the proposal. The component can accept two types of statements: INFORM statements with a proposal for evaluation; and REQUEST statements to provide specific information about the current proposal. It can generate REQUEST statements to retrieve information about negotiation protocols and EXECUTE statements that can provide the direct another component to evaluate the proposal content.

Proposal Evaluator The *Proposal Evaluator* component evaluates proposals using information about the proposal history, the opponent, the agent's own mental attitudes and the environment in general. Based on this evaluation, it can provide a recommendation as to the suitability of the proposal given the requirements of the agent, as described within its mental attitudes. The component can generate REQUEST statements to get information about the history, the opponent, the agent's mental attitudes and the environment. It can also generate INFORM statements to update the models of opponents and the environment based on the evaluation of the proposal, and INFORM statements to provide its evaluation of the proposal. It can accept EXECUTE statements with a proposal it should evaluate.

Response Generator The *Response Generator* component generates a response to the opponent's proposal, based on the evaluation of that proposal and information about the opponent, the environment and the agent's mental attitudes. The component can generate REQUEST statements to get the required information about the opponent, the environment, the mental attitudes, and the possible responses given the current state of the negotiation protocol. It accepts an INFORM statement with the evaluation of the proposal. Finally, it generates an EXECUTE statement that requests the generation of an appropriate message with the response within it.

Behavioural Specification The steps for the execution sequence of the component and

the flow of statements between them is described here. Note that what is described here is a purely sequential execution of components.

1. The operation of the agent begins with the agent accepting a message at the *Illocution Interpretation* sensor. This component analyses the message and informs the *Proposals History* infostore.
2. The *Proposals History* component executes and provides the information about the current proposal to the *Protocol Reasoner.*
3. The *Protocol Reasoner* controller executes and requests information from the *Negotiation Protocols* infostore.
4. The *Negotiation Protocols* infostore executes, providing the reply to the *Protocol Reasoner.*
5. The *Protocol Reasoner* is called to execute once more. It now uses the information from the *Negotiation Protocol* to reason about the validity of the proposal and informs the *Opponent Model* about the opponent identity and the *Proposal Evaluator* about the content of the proposal.
6. The *Proposal Evaluator* controller then executes, requesting information about the opponent, the environment, the agent's mental attitudes and the history of proposals.
7. The components *Opponent Model*, *Mental Attitudes*, *Environment Model*, and *Proposals History* components execute, providing the required information to the *Proposal Evaluator.*
8. The *Proposal Evaluator* component executes once more, evaluating the information provided from the previous step, and generating an evaluation for the *Response Generator.* In addition, it may inform the *Opponent Model*, and *Environment Model* of required updates to their models.
9. The *Response Generator* component executes, requesting the required information from the *Opponent Model*, *Mental Attitudes*, and *Environment Model*, as well as from the *Protocol Reasoner.*
10. The *Opponent Model*, *Mental Attitudes*, and *Environment Model* components execute to provide the responses and the *Response Generator.*
11. The *Response Generator* executes again, and generates the response based on the information provided.

12. Finally, the *Locution Generation* actuator executes to send the required message.

4.8.4 Argumentative Negotiating Agent

Here, we instantiate the architecture of the basic negotiating agent in order to provide a generic description of agents capable of conducting argumentation-based negotiation (ABN). An argumentative negotiator shares many components with the basic negotiator. For example, it also needs to be able to evaluate proposals, generate proposals and so on. What makes argumentative agents different is that they can exchange meta-information (or arguments) in addition to the simple proposal, acceptance, and rejection utterances. These arguments can potentially allow an agent to (i) justify their negotiation stance; or (ii) influence the counterparty's negotiation stance [122]. This may lead to a better chance of reaching agreement and/or higher-quality agreements. In ABN, influencing the counterparty's negotiation stance takes place as a result of providing it with new information, which may influence its mental attitudes (e.g., its beliefs, desires, intentions, goals, and so on). This might entice (or force) the agent to accept a particular proposal, or concede on a difficult issue. Arguments can range from threats and promises (e.g. [197]) to logical discussion of the agent's beliefs (e.g. [166]) or underlying interests [172].

In order to facilitate ABN, the logical and communication language usually needs to be capable of expressing a wider range of concepts. For example, the proposal might instead be represented as $propose(a, b, P, A)$ where a and b are agents, P is a proposal, and A is a supporting argument denoting why the recipient should accept that proposal. ABN frameworks may also allow agents to explicitly request information from one another. This may be done, for example, by posing direct questions about an agent's preferences or beliefs, or by challenging certain assumptions the agent adopts. Since in this chapter we are more interested in the abstract structures within the agents, we shall not discuss these issues in more detail.

In order to be capable of engaging in ABN, an agent needs the following additional capabilities:

1. **Argument Evaluation** encompasses the ability of the agent to assess an argument presented by another, which may cause updates to its mental state. It is the fundamental component that allows negotiators' positions to change.

2. **Argument Generation** allows the agent to generate possible arguments, either to support

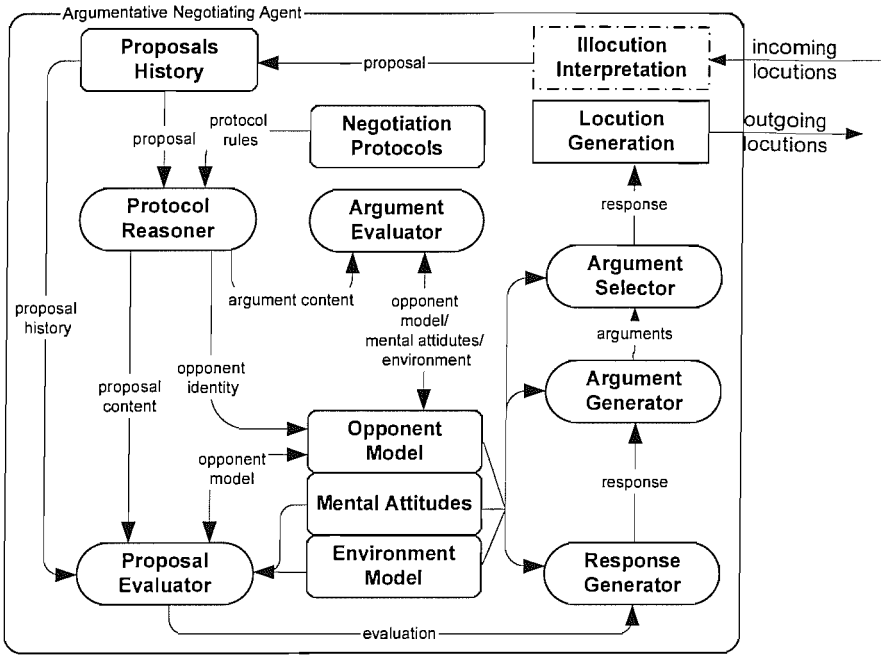

FIGURE 4.11: Argumentation-based negotiation agent architecture

a proposal, or as an individual piece of meta-information. The locution generated may also be a question to present to the opponent.

3. **Argument Selection** chooses between the number of possible arguments to present. For example, an agent might be able to make either a promise or a threat to its opponent. A separate component is needed to allow the agent to choose the most preferred argument. Selection might be based on some analysis of the expected influence of the argument, or on the commitments it ties the utterer to.

Figure 4.11 shows the specification of an argumentative agent using our construction model. All components from the basic negotiating agent have been used, complemented by the additional capabilities needed for ABN. Note that the diagram has been simplified for clarity, with the link from *Negotiation Protocol* to *Response Generator* and *Argument Generator* has been omitted although it is, of course, necessary. Below we show how the descriptive and behavioural specification are changed. The structural specification changes by adding the three new components that deal with ABN.

Descriptive Specification A crucial difference between the simple negotiation agent and the ABN agent is that arguments from opponents can change the agent's mental attitudes so that the agent's goals or motivations may change based on the new information obtained. As a

result, even this aspect of the descriptive specification is dynamic, and the ability to refer to this changing descriptive specification directly, at run-time, by extracting the relevant attributes is crucial. The descriptive specification must also include the new decision-making capabilities of the agent.

Behavioural Specification Here, the flexibility provided by the agent construction model is particularly evident. The agent essentially has the same links and information flows. It is simply *extended* with links to the new controllers, and is refined through changes to the execution sequence. The opponent model, mental attitudes and environment model are now updated by the evaluation of the argument received before the proposal is evaluated. The response is not sent directly to the opponent but arguments may be attached to the proposal by the *Argument Generator* and *Argument Selector* components. Finally, both the *Response Generator* and *Argument Generator* use the negotiation rules in order to determine what type of responses are possible.

4.8.5 Discussion

The examples of the auction agent architecture and negotiating agent architecture serve as a means to illustrate the application of the *act*SMART model in the development of both basic and more elaborate agent architectures. Here we highlight some of the benefits of the approach and link them back to the desiderata mentioned in Section 4.2.1.

- Access to an abstract agent model, in our case SMART, allows us to describe issues such as goals without having to represent them explicitly within the agent architecture. Goals can be defined in the descriptive specification of the architecture, and as such made clear. However, if there is no need for them to be made explicit within the architecture, then no implementation of goals is required in the structural and behavioural specifications. This provides more flexibility to the developer to provide a solution tailored to the problem at hand.

- The architectures for the auction agent and the negotiating agents are very different, since each is focused on solving the problem at hand. However, we can reason about both architectures using the same concepts, both at the abstract level through SMART and at the specification level through *act*SMART. This illustrates the benefit of an architecturally-neutral approach to agent construction, since it enables us to reason about a range of

different architectures through a consistent approach, thus minimising the learning effort for agent developers.

- Finally, modularity is achieved through the component-based approach that enables us to clearly separate the functionalities of the agent architecture. This, coupled with the distinction along the lines of description, structure and behaviour, allows us to re-configure architectures as illustrated by the move from the negotiating agent architecture to the argumentative negotiating agent architecture.

4.9 Conclusions

4.9.1 Related Work

Although a number of agent construction toolkits claim to make use of a component-based architecture (e.g., ZEUS [155], RETSINA [206], the majority do not do this through any consistent component model and do not provide the ability to reconfigure the architecture. However, as we have argued, it is necessary to have a consistent view of architecture construction and to support modularity and reconfigurability. In this section, we compare and contrast our work with existing work which we consider to have some related features, and which has attempted to achieve similar aims.

DESIRE

The DESIRE methodology [35, 36] is perhaps the closest work to our own since they take a very strong component-based approach and use it to define a variety of agent architectures, including some used by existing toolkits. However, DESIRE views both individual agents and multi-agent systems as a compositional architecture in which all functionality is designed as a series of interacting, task-based, hierarchically structured components. Other than at the lowest level components are seen as encapsulating processes, and composition of components is, therefore, a composition of processes. Communication between components is also supported through the notion of *information links*.

However, our approach differs in two important ways.

- Firstly, through *act*SMART, we aim to provide specific support for *individual* agent design rather than general support for an entire multi-agent system. As such, the approach is more lightweight and follows a far more specific route in the nature of components and the ways in which they can communicate. We introduce component types such as sensors and actuators to specifically model agent functionality and define specific *types* of statements that components can exchange.

  As a result, *act*SMART provides abstractions that developers can make *immediate* use of when attempting to design an agent architecture. With DESIRE, on the other hand, such abstractions would still need to be added to the existing concepts in support of component-based design since they do not provide any specific support for individual agent architectures. Beyond the increased development effort, the disadvantage is that there are no guarantees of a consistent view *across* agent architectures..

- Secondly, in the context of its aim to support reuse, *act*SMART provides some specific improvements over DESIRE. We base our construction model on a well-established *abstract* agent model, distinguish between different aspects of agent architectures through the descriptive, structural and behavioural specifications, and support the re-configuration of the architecture. All these features make *act*SMART particularly well suited to the heterogeneous, dynamic application domains we wish to support, and improve what can be accomplished with DESIRE.

Agent specification using multi-context systems

The work by Sabater et al. [189] and Parsons et al. [165] on agent specification using multi-context systems also has some similarities. Their basic model calls for agents to be constructed using *units*, which represent the main components of the architecture, and *bridge rules* that relate formulae in different units. In addition, they use *modules* to encapsulate related units that provide a specific functionality such as *planning*. Finally, they also support two types of messages; *ask* and *answer*. Units generate bridge rules that are *multicast* across either an *inter-module* bus or an *intra-module* bus. Components must examine the message to determine who they are intended for, before processing them.

Beyond the fact that the *emphasis* of the work is on handling different types of logics within a single architecture, rather than dealing with practical, implementation issues there are some significant differences at the level of *specifying* architectures. Firstly, in *act*SMART, we specify

types of components in order to aid design. Secondly, we provide a more flexible and cost-effective means of managing information-flow between components since we avoid *multicasting* messages to all components. At the same time, we still allow for reconfigurable information-flows, which was the justification for the use of a multicast technique by Sabater et al. Finally, as already stated above, *act*SMART benefits from its grounding in an abstract agent model that aids consistency across different architectures.

In general, *act*SMART and agent specification using multi-context systems could be considered as complementary since *act*SMART can inform the *practical* implementation of multi-context systems specification, while multi-context system specification can inform the development of *act*SMART to handle the use of different logics within an agent architecture.

JADE

The JADE (Java Agent Development Environment) toolkit is one of the few toolkits for agent development that does not restrict the developer to the use of any specific architecture [22]. It offers support for a number of *behaviour types* (which are akin to our notion of an execution sequence) that can be composed to define the control-flow of architectures, such as *sequence*, *parallel*, and so forth. Components can then be linked to such behaviour types, and the JADE infrastructure handles their execution. However, there is no underlying conceptual support for agent architectures nor any other concepts to facilitate the process from design to practical implementation.

Once more, JADE and *act*SMART can be considered as essentially complementary technologies. JADE offers the required underlying functionality to develop FIPA-compliant agent systems [22], and also offers some support for defining behaviours for agent components. These behaviours can be considered as equivalent to the *execution sequences* in *act*SMART, which also provides extensive support for specifying and implementing agent architectures.

4.9.2 Discussion and Contributions

As mentioned earlier, there are several systems that claim to provide (and in some cases do provide) a component-based approach to agent development. Of these, only a small number aim to support a *range* of agent architectures as discussed above. The agent construction model

described here complements and advances such work and, in particular, the work provides the following advances.

- Through *act*SMART we support the specification of a range of agent architectures, which can all be considered through a consistent conceptual model. This addresses a real need for support of agent development in heterogenous and dynamic environments, where the ability to tailor architectures to address specific application demands is important. At the same time, we allow for architectures to be easily reconfigured through a modular, component-based approach and provide a distinction between the descriptive, structural and behavioural specifications of an agent architecture which provides a developer with a clear distinction between different points of view and the ability to move between them while refining the design of an agent architecture. In addition, the development of the model leads to the following related contributions which can be used to inform the development of other agent construction models.
  - We categorise components into four generic types that allow us to specify abstract architectures before needing to focus on the internal behaviour of components.
  - We develop a graphical notation to represent such architectures.
  - We provide a well-defined notion of *shell* as the manager of the control-flow and information-flow between components, through the definition of an *execution sequence* and *links*, respectively.
- Through *act*SMART we also improve the suitability of a well-established theoretical model to application development since we provide a clear path for the *practical construction* of agents, based on the abstract agent models provided by SMART.

The two examples presented in this chapter provide a direct indication of how *act*SMART can be used to create agent architecture specifications. They highlight how the use of component types allows us to focus on the high-level specification without having to deal with the detailed operation of each component. This suggests that the development of agent architectures can be separated from the algorithms that, for example, deal with specific types of negotiation protocols, enabling overall design to proceed in a parallel manner while still allowing specific techniques to be incorporated within agent architectures at any stage. Furthermore, the negotiating agent example indicates how the specification of an architecture can be developed through

a process of refinement. In the specific case of the example, the design moved through two iterations. In the first, we provide a detailed descriptive specification but less clear structural and behavioural specifications, while in the second we further develop the structural and behavioural specifications using the descriptive specification as a statement of *requirements* that the other specifications should fulfill.

In conclusion, the agent construction model represents an important step towards our aim of providing *principled* and *reusable* models for agent-based development. In this chapter we have presented the main concepts of the model and examples of how it can be used to specify agent architectures. In Chapter 6 we present the application of this model for use within the context of architecture development for ubiquitous computing devices, and also discuss its implementation within a specific programming language.

Chapter 5

SMART+ : Relationship Identification and Characterisation

5.1 Introduction

In the previous chapter we examined the construction of individual agents and how we can design a wide range of agent architectures based on a common set of concepts that can be tailored to their operational environment. We now turn our attention to *multi-agent systems*, in which the *interactions* between agents are the central concern. Such *interactions* take place whenever one agent performs an action which, intentionally or otherwise, affects one or more other agents. When agents interact we can say that they are *related* by virtue of the fact that they are affecting each other. Interactions, and the resulting relationships formed between agents, are of critical importance to the overall system functioning, since they can have both beneficial and adverse effects. It is interactions that enable agents to coordinate (by which agents arrange their individual activities in a coherent manner), collaborate (by which agents work together to achieve a common objective) or compete (by which agents contend for access to common resources), and so on.

In this respect, a system designer has two overarching challenges to face. On the one hand, the system designer must ensure that the interactions that are necessary for achieving system-wide goals take place. For example, if agents require assistance to achieve their goals, they must be provided with mechanisms for discovering other agents able to assist them. On the other hand, the designer must also ensure that undesirable interactions do not take place. For

example, if a number of agents depend on a limited resource, the system must provide ways to control access to that resource. These challenges are compounded by the fact that in open or simply large agent systems, the possible interactions between agents cannot all be explicitly specified at design-time. This is especially true when dealing with *autonomous* agents operating in heterogeneous environments in which agents may join or leave the system at any time, and no assumptions are made about agent behaviour.

Given the above, it is clear that system designers require *relationship management* mechanisms that can constrain or empower agents to form only the kinds of relationships that are beneficial for the overall system. For example, agents could be forced to adhere to specific regulations which indicate whether they are allowed or not to perform an action in a specific context. However, such mechanisms cannot be applied to agent systems unless there is a clear understanding of what relationships may arise in a multi-agent system, or which regulations are required to lead to only effective relationships. Even if we could assume that for closed, static agent systems, such an understanding can be achieved, the same cannot be said for open, dynamic multi-agent systems since relationships, and by consequence the appropriate regulations to manage them, can change at any time. Therefore, there must be some method for *systematically identifying* the relationships that can arise, and only then addressing the problem of defining the necessary regulation or coordination mechanisms.

The need for some form of control over the behaviour of agents was identified long ago [63], and there has been a wealth of research on the subject since. The review of existing work on the issue of regulatory structures for controlling behaviour in multi-agent systems in Chapter 2 reveals that the focus of others has been more on relationship management than on relationship identification and characterisation. Currently, the former is largely achieved through the use of regulatory frameworks, stemming from work on policies (e.g. [84, 204, 128]), institutions (e.g. [89, 218]) and norms (e.g. [234, 49, 73]). In addition, there is significant work on coordinating middleware to enhance agent infrastructure [55, 161].

Our aim in this chapter is to focus on the latter, insufficiently addressed, issue of *relationship identification*. Of course, once a relationship has been identified we must be able to interpret that information is some useful fashion so as to determine how the identified relationships may impact on individual agent operation and the system as a whole. Thus, we also require a principled and comprehensive means of *characterising agent relationships*.

In order to identify the relationships that may be formed between agents, we introduce a model

of interaction of an agent with the environment. The model makes minimal assumptions about the agents themselves, considering only an agent's sensor and actuator capabilities. Based on these sensory and actuator capabilities, we identify which environmental attributes an agent can sense or affect respectively. This leads to two sets of attributes, one defining a region of the environment an agent can view and the other a region of the environment an agent can affect. By comparing this information between different agents we can identify which environmental attributes agents can sense or affect that are in common between them, and by consequence identify how two agents may be related. This technique can also be used in a more generic sense so as to provide generic relationships types based on the types of overlap between the environmental regions. We make use of this approach to identify and characterise all the possible relationships between two agents. This typology allows us to reason about relationships between agents at an abstract level, without needing to ground relationships to specific domain information. In turn, this enables us to define when a particular relationships management technique (such as a regulation) is applicable based on the identification of a generic relationship type. This is especially useful in environments where agent capabilities can constantly change. By basing the definition of regulations on generic relationship types we can ensure that regulations are enforced irrespective of the specific capabilities of individual agents.

Of course, agents may not make use of all their sensory or actuator capabilities, since what ultimately determines the specific actions an agent decides to perform are the agent's goals. Therefore, we also relate goals to the model of agent interaction with the environment and discuss how such information can allow us to reduce or expand the set of possible relationships that may be formed between agents.

Overall, the ability to identify and characterise agent relationships can be beneficial for the following reasons.

- It can guide the choice and design of regulatory frameworks to prevent malicious behaviour or interference between agents.
- Potentially missed opportunities for better cooperation between agents can be identified.
- It can provide a template of generic relationship types for coordinating agents at run-time, without having prior knowledge of their capabilities.

The next section introduces the model of agent interaction with the environment and provides formal definitions for the sets of attributes agents can possibly sense or affect. We provide some

examples of how the model can be used and discuss the assumptions underpinning the model. Subsequently, we develop the typology of agent relationships, which provides definitions for the most salient relationship types that can then be used to define other more specific types. Next, we discuss how knowledge about an agent's goals can allow us to narrow or expand the space of possible relationships, providing an example of the use of such information. Finally, by means of an example, we use the relationship typology and knowledge about an agent's goals to define a particularly interesting set of relationships, where agent's can *interfere* with each other.

5.2 Model of Agent Interaction

A number of different and interrelated issues determine the resulting relationships between agents. Some relationships are *built-in* by the system designer at design-time. For example, in the ZEUS system [155], agents must communicate with the root agent name service agent before doing anything else. [1] Alternatively, relationships may develop opportunistically as agents seek assistance in order to achieve goals, or unintentionally as agents perform actions that affect the environment that other agents aim to affect or sense. For example, in the RETSINA system [206], discovery of middle agents takes place dynamically, and only if the agent requires assistance in achieving a goal.

Our basic understanding of agents, as set out in Chapter 3 and Chapter 4, is that agents perform *actions*, which may change the *environment*, whilst pursuing *goals*. Actions can be divided into those that provide agents with information about the environment (sensors) and those that change attributes about the environment (actuators). Intuitively, we can think of each agent as able to create a *view* of some *region of the environment*, by its ability to retrieve the attributes that define that region. In addition, each agent is able to *directly influence* some region of the environment, by manipulating attributes that define that region. Drawing on this, we can say that when one agent is able to *influence* what another agent views or what another agent is able to influence, the two agents are *related*. The different ways in which this influence can emerge leads to different types of relationships.

Our aim is to produce analytical tools to enable the identification of relationships, building on just these basic notions. The emphasis is on being able to identify relationships that may not have been foreseen by the system designer both at system design-time as well as during run-

[1] The root agent name service keeps a register of all active agents in a ZEUS-based application.

time, something particularly relevant in open agent systems. In order to achieve this, we begin by introducing some basic but necessary formal concepts that build on the models of agents already introduced in Chapter 3.

Firstly, we develop models that can allow us to relate the sensory and actuator capabilities of an agent to the regions of the environment that an agent is able to sense or affect. In Section 5.3, we use these models to derive some generic relationship types that can be used to describe any relationship between two agents.

5.2.1 Agent Perception and Action

Agent actions are divided into those that retrieve specific attributes of the environment, representing the agent's *sensor capabilities*, and those that attempt to *change* attributes of the environment, representing the agent's *actuator capabilities*. The former define what an agent can perceive in the environment, and the latter what an agent can change. These actions are the only aspects of the agent's architecture that concern us at this point, as they form the *interface* between the agent and its external environment. So as not to restrict the models to any specific agent architectures, we do not concern ourselves with the internal state or decision-making capabilities (represented by an agent's *infostores* and *controllers*) of an agent. Nevertheless, as we will see later on, knowledge of the *exact* goals an agent is pursuing, although not strictly necessary, can enable a deeper analysis of interactions since it can lead to a better understanding of the reasons behind the manifest actions and limit the space of possible emerging relationships.

Agent Perception

The environmental attributes that an agent is able to perceive depend on the sensory capabilities the agent is equipped with. The set of capabilities can be divided into those that an agent is actually using at any particular moment and those it is not using. However, for the purposes of relationship identification, we need to represent the *entire set* of environmental attributes that an agent may attempt to view, irrespective of whether the agent is actually using them at any particular moment. This allows us to capture the widest possible set of relationships an agent may have with other agents and is in line with the aim to avoid any attempt to model the internal operation of an agent.

In addition, we do not model the fact that although two agents may *attempt* to view the same

aspects of the environment there is no guarantee that their sensory capabilities will produce the same results, since they may both sense and interpret the environment in a variety of different ways and it is not possible to assume that across different agents we will encounter consistent models of the environment. In other words, even if two agents have the same sensory capabilities, there is no guarantee that they actually have the same *mental models* of those aspect of the environment. This is a distinction that SMART makes, but which we purposely avoid making, since it necessarily requires a model of the internal operation of the agent architecture.

With these clarifications in place, we can now present a formal definition for the possible percepts of an agent, through the *PossibleAgentPerception* schema. It includes the *Agent* schema and is further refined by introducing the set, *perceivingactions*, which is the subset of the capabilities of the agent that are concerned with perceptions, and the function, *canperceive*, which determines the attributes of the environment that are *potentially* available to an agent through its perception capabilities.

PossibleAgentPerception

$$
\begin{array}{l}
Agent \\
perceivingactions : \mathbb{P}\, Action \\
canperceive : Environment \rightarrow \mathbb{P}\, Action \nrightarrow Environment \\
\hline
perceivingactions \subseteq capabilities \\
\forall\, env : Environment;\ as : \mathbb{P}\, Actions \bullet \\
\quad as \in \mathrm{dom}(canperceive\ env) \Rightarrow as = perceivingactions
\end{array}
$$

Agent Action

Similarly to agent perception, we can define the set of *possible actions*. Again, we are not interested in those actions that the agent will actually perform because of its current goals, but all the actions that agents could *potentially* perform. The *PossibleAgentActions* schema defines the set of actions that can influence the environment as the *effectingactions*, and the function returns the set of attributes these actions can influence as *caneffect*.

PossibleAgentAction

$$
\begin{array}{l}
Agent \\
effectingactions : \mathbb{P}\, Action \\
caneffect : Environment \rightarrow \mathbb{P}\, Action \nrightarrow Environment \\
\hline
effectingactions \subseteq capabilities \\
\forall\, env : Environment;\ es : \mathbb{P}\, Actions \bullet \\
\quad es \in \mathrm{dom}(caneffect\ env) \Rightarrow es = effectingactions
\end{array}
$$

Agent Influence State

Using the *PossibleAgentPerception* and *PossibleAgentAction* schemata, we can now define an *agent influence state* as all that an agent is able to potentially view and affect at any given moment. This notion is formalised in the *AgentInfluenceState* schema, which includes the *Agent*, *PossibleAgentPerception* and *PossibleAgentAction* schemata.

AgentInfluenceState

$$Agent$$
$$PossibleAgentPerception$$
$$PossibleAgentAction$$

Finally, the influence state of an entire multi-agent system is given by the environment and the influence states of each individual agent in that environment. The predicates of the *MAInfluenceState* schema state that for all the agents in the system, all their attributes and their situation in the environment are a subset of the environment. Furthermore, the possible percepts of all agents are a subset of the environment, and what they can affect is also a subset of the environment.

MAInfluenceState

$$environment : \mathbb{P}\, Attribute$$
$$agents : \mathbb{P}\, AgentInfluenceState$$

$$\forall\, a : agents \bullet a.attributes \subset environment$$
$$\forall\, a : agents \bullet a.possiblepercepts \subseteq environment$$
$$\forall\, a : agents \bullet a.caneffect \subseteq environment$$

5.2.2 Viewable Environment and Region of Influence

The schemata introduced so far have set the scene by defining how we can model what attributes an agent can possibly perceive or effect. Using these models, we now introduce two new concepts, which directly define the regions of the environment that we are interested in. A *region* is defined as a set of attributes in the external environment of the agent.

Viewable Environment An agent's *Viewable Environment* depends on its sensory capabilities, the environment it is situated in, and the other agents in the system, which also form part of the environment. Agents sense the environment in order to take decisions about which goals to perform or to verify the results of actions taken. The set of attributes they can

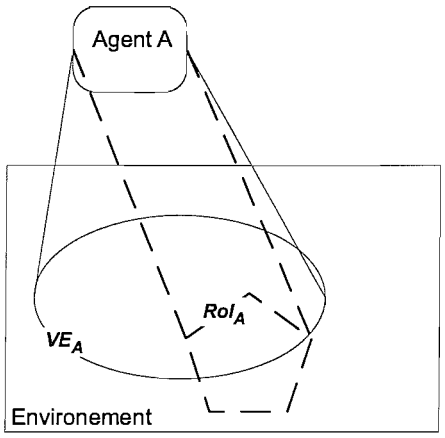

FIGURE 5.1: *Viewable Environment* and *Region of Influence*

potentially perceive within a particular environment, without recourse to aid from other agents, defines a *Viewable Environment*. The *ViewableEnvironment* schema formalises this notion. It includes the *MAInfluenceState* schema, and defines the *viewable* function as a partial function, which takes *AgentInfluenceState* as an argument and returns a set of attributes. The predicates state that the domain of *viewable* is the set of agents, while the range of *viewable* is a subset of the environment.

$$
\begin{array}{|l}
\textit{ViewableEnvironment} \\
\hline
\textit{MAInfluenceState} \\
\textit{viewable} : \textit{AgentInfluenceState} \pfun \mathbb{P}\, \textit{Attribute} \\
\hline
\mathrm{dom}\, \textit{viewable} = \textit{agents} \\
\forall\, a : \textit{agents} \bullet \textit{viewable}\ a \subseteq \textit{environment} \\
\hline
\end{array}
$$

Region of influence Agents can affect the environment by changing attributes in it. Those attributes they can change on their own, through their actuator capabilities, define a *Region of Influence*. [2] This notion is formalised in the *RegionOfInfluence* schema which, once more, includes the *MAInfluenceState* schema. The *regionofinfluence* function provides a set of attributes, the domain of which is the set of agents in the system, while the resulting attributes are a subset of the environment.

$$
\begin{array}{|l}
\textit{RegionOfInfluence} \\
\hline
\textit{MAInfluenceState} \\
\textit{regionofinfluence} : \textit{AgentInfluenceState} \pfun \mathbb{P}\, \textit{Attribute} \\
\hline
\mathrm{dom}\, \textit{regionofinfluence} = \textit{agents} \\
\forall\, a : \textit{agents} \bullet \textit{regionofinfluence}\ a \subseteq \textit{environment} \\
\hline
\end{array}
$$

[2] Similarly to the *Viewable Environment*, the *Region of Influence* can be partially defined through knowledge of the individual capabilities of each actuator component of the agent.

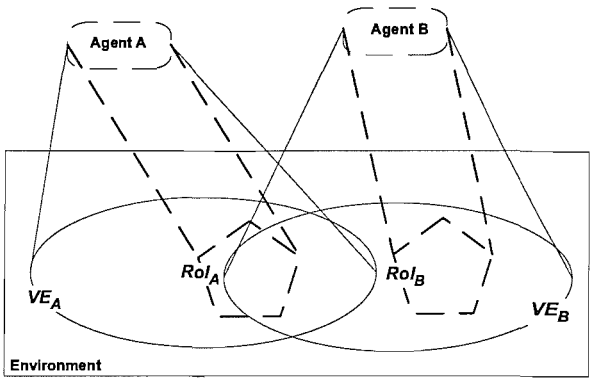

FIGURE 5.2: *Region of Influence* affects *Viewable Environment*

The *Viewable Environment* and the *Region of Influence* of an agent provide us with a model that relates an agent and its individual capabilities to the environment, within a particular multi-agent system. It provides us with information on those aspects of the environment that an agent could potentially affect and view. This model makes no assumptions about the agent itself nor its internal decision-making capabilities. Thus it can be considered to be architecturally-neutral and applicable to the widest possible range of agent types. Crucially, the utility of the model is evident when we represent one agent's *Viewable Environment* and *Region of Influence* in comparison to another agent's *Viewable Environment* and *Region of Influence*. Now, the different ways in which the *Viewable Environments* and *Regions of Influence* between two agents can overlap defines *a space of possible interactions*. In Figure 5.1, these concepts are illustrated by using an ellipse to represent the *Viewable Environment*, and a pentagon for the *Region of Influence*. We use this notation throughout when illustrating different situations. In order to better illustrate how relationship identification can be achieved using this model, we present some generic examples below.

5.2.3 Generic Relationships Identification Examples

As mentioned earlier, in order to analyse the kinds of relationships that emerge from the interactions between agents, we need to consider the overlaps between their respective *Viewable Environments* and *Regions of Influence*. It is *within* these overlaps that interactions are likely to emerge, since they represent the only points at which agents may influence each other.

In Figure 5.2, we illustrate these concepts. Here, we have a situation in which Agent A's *Region of Influence* overlaps with Agent B's *Viewable Environment*, and both agents' *Viewable Environments* overlap. Given this information, we can infer that Agent A and Agent B could be

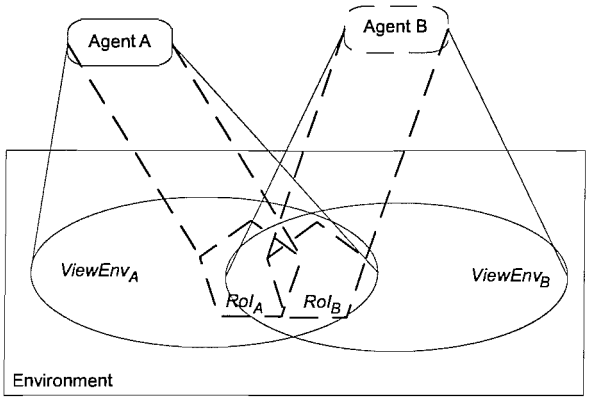

FIGURE 5.3: *Regions of Influence* overlap

related, with A able to directly affect the *Viewable Environment* of B, since it partly falls under A's *Region of Influence*. In other words, B can be *influenced* by the actions of A. Agent A, on the other hand, cannot be influenced by B. Crucially, A cannot *directly* affect the results of an action of B because it has no influence in the *Region of Influence* of B.

For example, consider a situation in which we wish to develop an agent-based infrastructure to support the collaboration and sharing of information between researchers participating at a conference. Each researcher is to be represented by a personal agent that will make public their personal profile (interests, publications, and availability) as well as research material (links to online material, presentations, software, etc) that they have stored locally.

A situation such as the one in Figure 5.2 could occur if the overlap between the *Viewable Environments* represented research papers that Agent A made available to other agents. With the goal of reporting to other agents on all documents of a specific type (for example, research papers on multi-agent systems), Agent B could periodically view the documents stored by A (i.e. sample the environment) while waiting for a relevant document to appear before informing other agents of its existence. So, whenever A performs an action that adds a relevant document to its public document store, it will eventually *influence* B's actions, since B must now inform interested parties about this addition.

The illustrated situation can also be interpreted as the ability of B to *observe* the results of actions performed by A. If the document store was not public, then some steps should be taken to prevent B from observing what documents were placed within it.

By contrast, in Figure 5.3, the situation is one in which the *Regions of Influence* overlap. This means that both agents can have a direct impact on the *actions* of each other. Thus, an action

from either agent could affect the environment in such a way that a goal of the other agent is constrained or aided. For example, this could happen if the two agents were both attempting to retrieve a document from a public document store that sets a limit on the number of documents retrieved.

5.2.4 Basic Assumptions

Before presenting a more formal characterisation of the different types of relationships that can be identified, we explicitly present below a number of assumptions that we make about the agents themselves that hold throughout our analysis.

1. The *Viewable Environment* and the *Region of Influence* are *not* necessarily well-defined continuous areas of the environment as the diagrams may suggest. However, representing them like this helps to provide a clear exposition.

2. There is no requirement for the *Viewable Environment* and the *Region of Influence* of an agent to overlap at all. If the *Region of Influence* of an agent does not fall under its *Viewable Environment*, then it will not be able to view the results of its actions, a situation that is not improbable. The more usual case is when only part the *Region of Influence* of an agent falls under the *Viewable Environment*. In other words, the agent is not fully aware of all the implications of its actions.

3. We do not assume that the *Viewable Environment* is the only kind of information that an agent can model. The *Viewable Environment* is simply the information that the agent can gain about the environment without recourse to other agents. Information about the environment provided by other agents is an issue we will examine later on.

4. We do not assume that when an agent acts in its *Region of Influence* it can be certain that those actions are realised. The only way to verify this is by sensing the affected environment, either through its own sensing capabilities or through other agents.

5.3 Relationship Typology

Having presented a model for agent interaction, and some examples of how it can be used to characterise possible relationships between agents, we now take a more systematic look at the

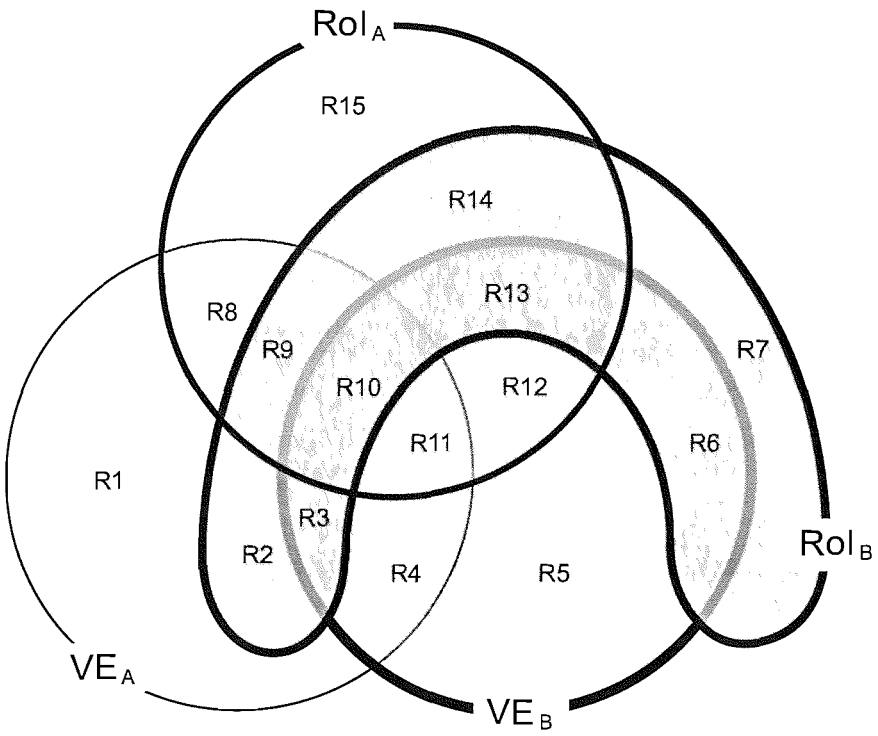

FIGURE 5.4: All possible relationships between two agents

entire space of possible overlaps. We develop a comprehensive typology of interactions that can provide the building blocks for defining a wide range of different relationships. However, before presenting the typology, we consider how the complete set of potential relationships may be examined comprehensively, and show that our analysis is complete.

The entire space of overlaps depends on the ways in which the four sets of regions (one for each agent's *Viewable Environment* and one for each agent's *Region of Influence*) can be combined. There are sixteen possibilities, which are illustrated by the Venn diagram in Figure 5.4. The Venn diagram uses Venn's construction for illustrating the possible combinations between four sets [219] and the resulting regions created (sixteen possibilities in all).

Now, of the fifteen regions enumerated in the diagram (the sixteenth being the region outside all the sets), we only consider those that involve intersections between two sets, ignoring regions $R1$, $R5$, $R7$ and $R15$. We can divide the remaining regions into three cases, which can be combined to produce other cases. These three cases are divided along the lines of whether the *Viewable Environments* of the two agents overlap, whether their *Regions of Influence* overlap, and whether a *Viewable Environment* overlaps with an *Region of Influence*.

Mutually Viewable Environment A *mutually viewable environment* occurs where the *View-*

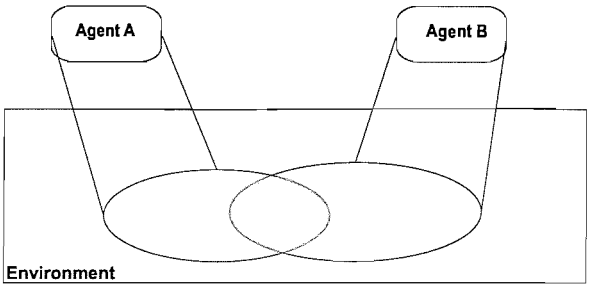

FIGURE 5.5: Mutually Viewable Environment

able Environments of both agents overlap. In the diagram this is made up of the areas defined by regions $R3$, $R4$, $R10$, and $R11$. These four regions are grouped according to whether both or one or neither of the *Regions of Influence*of the agents overlap with the *mutually viewable environment.*

Mutual Influence The *mutual influence* region is where the *Regions of Influence* of both agents overlap. As above, this region is also subdivided into four regions: $R9$, $R10$, $R13$ and $R14$. The differences between the four regions indicate whether the region of *mutual influence* overlaps with the *Viewable Environments*of one, both or neither of the agents.

Observable Actions A region of *observable actions* is one where a *Region of Influence* overlaps with a *Viewable Environment*, indicating that an agent can observe the actions taking place in that overlap. In the figure, this occurs in regions $R2$, $R3$, $R6$, $R8$, $R9$, $R10$,$R11$, $R12$, and $R13$. The type of observable action changes according to whether an agent is able to observe the actions it perform or the actions the other agent is able to perform.

Now, given these three basic types, and knowledge of all the possible relationships, we proceed to construct a typology by beginning with the simplest case where only the *Viewable Environments* of two agents overlap, and moving to consider the possible types of interaction when actions of other agents can be observed. Finally, we consider the possibilities when actions can be directly influenced by other agents due to overlapping *Regions of Influence* and combine that with the results on observability of actions.

5.3.1 Mutually Viewable Environment

We begin by examining the *Viewable Environments* of agents, irrespective of the *Regions of Influence*. The only possibility in this case is that they overlap, as illustrated in Figure 5.5.

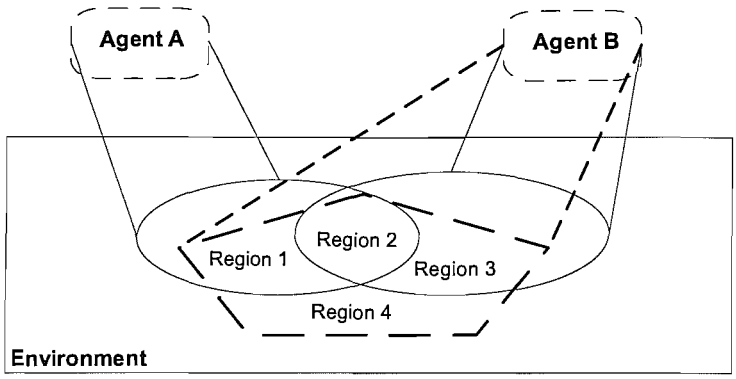

FIGURE 5.6: Observable and Invisible actions

This means that both agents are able to sense common regions of the environment. This situation is captured by the *MutuallyViewableEnvironment* schema, in which the function *MVE* accepts as arguments the relationships between the agent states of both agents, and returns a set of attributes, where those attributes are given by the intersection of A's and B's *Viewable Environments*.

$$
\begin{array}{|l}
\hline
\textit{MutuallyViewableEnvironment} \\
\textit{ViewableEnvironment} \\
MVE : (AgentState \times AgentState) \rightarrow \mathbb{P}\, Attribute \\
\hline
\forall\, a, b : AgentState;\ e : Environment \bullet \\
\quad MVE(a, b) = viewable\ a \cap viewable\ b \\
\hline
\end{array}
$$

Although such a situation cannot directly identify any relationship between the agents, it may be particularly important in certain environments. For example, knowing that two stock investors, which are seemingly unrelated, share a common *Viewable Environment*, may explain why they behave in a similar manner. A more practical example to illustrate the case of a mutually viewable environment is the ability of both agents to observe what files are added or removed from a filestore. We will use the example of actions with regard to a filestore to illustrate the various cases throughout.

5.3.2 Influenced Viewable Environment

The next step is to introduce the *Region of Influence*. However, we only do this for Agent B, as illustrated in Figure 5.6 in which, from A's point of view, there are two clear distinct possibilities, and two further refinements for each. Firstly, in Regions 1 and 2, the results of the actions of B

are visible to A since they fall within A's *Viewable Environment.* Of course, at the same time we can say that B is able to influence the *Viewable Environment* of A. Secondly, in Regions 3 and 4, the results of B's actions are not visible to A. We define these general cases before going on to specialise them further.

The *ObservableActions* schema defines the appropriate function for the first case. It states that the *observable actions* of B are those actions of B whose *Region of Influence* is within A's *Viewable Environment.*

ObservableActions

$$
\begin{array}{l}
ViewableEnvironment \\
RegionOfInfluence \\
observableactions : (AgentState \times AgentState) \rightarrow \mathbb{P}\, Attribute \\
\hline
\forall\, a, b : AgentState \bullet \\
observableactions(a, b) = viewable\ a \cap regionofinfluence\ b
\end{array}
$$

If we consider that Agent A is only able to monitor actions that are performed with regard to adding and removing files from the filestore then we can state that the *Viewable Environment* of A intersects with the *Region of Influence* of B in the region of the environment referring to the filestore.

Similarly, the *InvisibleActions* schema states that *invisible actions* of B with reference to A are those that are not within A's *Viewable Environment.*

InvisibleActions

$$
\begin{array}{l}
ViewableEnvironment \\
RegionOfInfluence \\
invisibleactions : (AgentState \times AgentState) \rightarrow (\mathbb{P}\, Attribute) \\
\hline
\forall\, a, b : AgentState \bullet \\
invisibleactions(a, b) = regionofinfluence\ b \setminus viewable\ a
\end{array}
$$

Returning to the example mentioned above, those actions of B that are not related to adding or removing files from the filestore that A is able to monitor are invisible to A.

Based on these definitions, we can now describe more restricted cases. We begin with the situation in which *both* agents can observe some actions of B, which would lie in Region 2 in Figure 5.6. The *BilaterallyObservableActions* schema defines this situation in which actions are bilaterally observable, and are given by the intersection of the *observable actions* for A on B

and for B on itself.

BilaterallyObservableActions

$$ObservableActions$$
$$bilaterallyobservableactions : (AgentState \times AgentState) \rightarrow \mathbb{P}\, Attribute$$

$$\forall a, b : AgentState \bullet$$
$$bilaterallyobservableactions(a, b) =$$
$$\quad observableactions(a, b) \cap observableactions(b, b)$$

Knowledge of the possibility of bilaterally observable actions can be relevant for those agents that require confirmation of their actions by another party, or for those agents that are concerned about the observability of their actions and would perhaps prefer to prevent it. With reference to the filestore example, we could use knowledge of bilateral observability to confirm that a file has been appropriately saved since A can provide further confirmation. Alternatively, B could decide not to store a file because A would be able to monitor that action.

Now, unilaterally observable actions are those actions of B that A can observe but B cannot (shown as Region 1). In this case, there is perhaps a stronger incentive for B to exploit the situation by cooperating with A so as to gain confirmation of the results of actions. The schema *UnilaterallyObservableActions* describes this.

UnilaterallyObservableActions

$$ObservableActions$$
$$InvisibleActions$$
$$unilaterallyobservableactions : (AgentState \times AgentState) \rightarrow \mathbb{P}\, Attribute$$

$$\forall a, b : AgentState \bullet$$
$$unilaterallyobservableactions(a, b) =$$
$$\quad observableactions(a, b) \cap invisibleactions(b, b)$$

In this case *only* A can confirm whether a file has been stored in the filestore, which in turn may make B's reliance on A greater if some form of confirmation of the result of the action is required.

Bilaterally invisible actions, represented by the *BilaterallyInvisibleActions* schema are those actions of B that both A and B cannot observe (shown as Region 4).

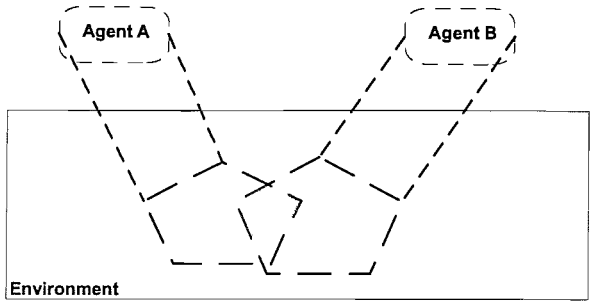

FIGURE 5.7: Mutually Influenced Actions

BilaterallyInvisibleActions

$$
\begin{array}{l}
\textit{InvisibleActions} \\
\textit{bilaterallyinvisibleactions} : (\textit{AgentState} \times \textit{AgentState}) \rightarrow \mathbb{P}\, \textit{Attribute} \\
\hline
\forall\, a, b : \textit{AgentState} \bullet \\
\textit{bilaterallyinvisibleactions}(a, b) = \\
\quad \textit{invisibleactions}(a, b) \cap \textit{invisibleactions}(b, b)
\end{array}
$$

Finally, for the sake of completeness, we can also define unilaterally invisible actions (in Region 3), as those actions of B that A cannot see but B can. The schema *UnilaterallyInvisibleActions* captures this.

UnilaterallyInvisibleActions

$$
\begin{array}{l}
\textit{InvisibleActions} \\
\textit{unilaterallyinvisibleactions} : (\textit{AgentState} \times \textit{AgentState}) \rightarrow \mathbb{P}\, \textit{Attribute} \\
\hline
\forall\, a, b : \textit{AgentState} \bullet \\
\textit{unilaterallyinvisibleactions}(a, b) = \\
\quad \textit{invisibleactions}(a, b) \cap \textit{observableactions}(b, b)
\end{array}
$$

5.3.3 Mutual Influence

Up to this point, we have only dealt with the issue of observability of actions. We now move on to examine the situations in which agents can influence each other's actions, by introducing *Regions of Influence* for both agents. In the first instance, as illustrated in Figure 5.7, we can say that two agents are able to directly influence each other if their *Regions of Influence* overlap (shown as the grey shaded area). The function for determining this is defined below, in the schema *MutualInfluence*.

MutualInfluence

$ViewableEnvironment$
$RegionOfInfluence$
$mutualinfluence : (AgentState \times AgentState) \rightarrow (\mathbb{P}\, Attribute)$

$\forall\, a, b : AgentState \bullet$
$mutualinfluence(a, b) = regionofinfluence\ a \cap regionofinfluence\ b$

Returning to the filestore example, mutual influence would occur when A is *also* able to add or remove files in the filestore that B may have added or removed.

Now, when a mutual influence relationship occurs (i.e. a non-empty set is returned), it is important to be able to model whether the two agents can observe the results of actions taking place in this region of the environment. We can use the previous definitions of observability of actions in Section 5.3.1 to model this.

First, we define the relationship by which Agent A can observe the region of mutual influence in the *ObservableMutualInfluence* schema, which includes the *MutualInfluence* schema, and states that this area is the intersection of the *Viewable Environment* of A and the area of mutual influence between A and B.

ObservableMutualInfluence

$MutualInfluence$
$observablemutualinfluence : (AgentState \times AgentState) \rightarrow (\mathbb{P}\, Attribute)$

$\forall\, a, b : AgentState \bullet$
$observablemutualinfluence(a, b) = viewable\ a \cap (mutualinfluence\ (a, b))$

In this case, A is able to *both* influence and monitor the actions of B with regard to the filestore.

Similarly, Agent A may not be able to observe this region of mutual influence. We define the case of invisible mutual influence in the schema *InvisibleMutualInfluence*.

InvisibleMutualInfluence

$MutualInfluence$
$invisiblemutualinfluence : (AgentState \times AgentState) \rightarrow (\mathbb{P}\, Attribute)$

$\forall\, a, b : AgentState \bullet$
$invisiblemutualinfluence(a, b) = (mutualinfluence\ (a, b)) \setminus viewable\ a$

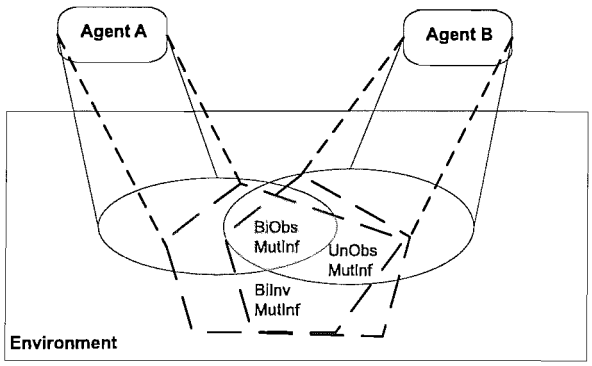

FIGURE 5.8: Mutual Influence and Observable Actions

With regard to the filestore example, a case of invisible mutual influence would mean that while A is also able to add or remove files it is not able to view the results of those actions.

Having provided definitions from one agent's perspective, we consider the situation in which both agents' *Viewable Environments* are taken into account. The first case is bilaterally observable mutual influence, in which both agents can observe the mutual influence region, as illustrated in Figure 5.8. The region in question is where both agents' *Regions of Influence* overlap as well as their *Viewable Environments*. The schema *BilaterallyObservableMutualInfluence* formalises this.

BilaterallyObservableMutualInfluence

$$
\begin{array}{l}
ObservableMutualInfluence \\
bilaterallyobservablemutualinfluence : \\
\quad (AgentState \times AgentState) \rightarrow \mathbb{P}\, Attribute \\
\hline
\forall\, a, b : AgentState \bullet \\
bilaterallyobservablemutualinfluence(a, b) = \\
\quad observablemutualinfluence(a, b) \cap observablemutualinfluence(b, a)
\end{array}
$$

In this case, both A and B can add or remove files and observe each other's addition or removal of files.

The *BilaterallyInvisibleMutualInfluence* schema provides the necessary functions for the mutual influence being bilaterally invisible.

BilaterallyInvisibleMutualInfluence

$$InvisibleMutualInfluence$$
$$bilaterallyinvisiblemutualInfluence : (AgentState \times AgentState) \rightarrow \mathbb{P}\, Attribute$$

$$\forall a, b : AgentState \bullet$$
$$bilaterallyinvisiblemutualinfluence(a, b) = invisiblemutualinfluence(a, b) \cap invisiblemutualinfluence(b, a)$$

In this case, neither of the two agents can observe the results of their actions with regard to the filestore, although they could both upset each other's actions. In such a situation, it may be necessary to introduce some form of control, possibly by a third party, that can ensure that the actions of the agents are appropriately coordinated.

Finally, we define the situation in which one agent unilaterally observes the region of mutual influence in the *UnilaterallyObservableMutualInfluence* schema.

UnilaterallyObservableMutualInfluence

$$ObservableMutualInfluence$$
$$unilaterallyobservablemutualinfluence : (AgentState \times AgentState) \rightarrow \mathbb{P}\, Attribute$$

$$\forall a, b : AgentState \bullet$$
$$unilaterallyobservablemutualinfluence(a, b) = (observablemutualinfluence\ (a, b) \setminus observablemutualinfluence\ (b, a))$$

In this case, we know that *only* A can observe the results of actions on the filestore, while both A and B can affect changes.

These types of possible relationships are particularly relevant for agents that wish to better coordinate their actions. For example, knowledge of a possible bilaterally invisible mutual influence can indicate that agents should be particularly careful when performing actions in that region, since not only are they unable to observe the results of their own actions, but they can also constrain the actions of another agent that is also unable to observe the results. A situation of unilaterally observable mutual influence could give one agent an advantage, since only one of them is able to observe the results of its and the other's actions in that region.

5.4 Goal Typology

Knowledge of the sensor and actuator capabilities of agents can provide us with enough information to identify a significant number of possible relationships and categorise them along the lines of the typology introduced above. Nevertheless, not *all* the possible relationships identified will actually be instantiated, and there may still be instantiated relationships that have not been identified. The reason for this is that the goals agents decide to pursue play an important role in determining which of all possible relationships agents choose to instantiate. With the additional knowledge of what goals an agent may actually pursue, we can narrow or expand the space of possible relationships by identifying interactions that an agent may pursue that are beyond its range in terms of its *Region of Influence* or its *Viewable Environment*, or by excluding those within its *Viewable Environment* and *Region of Influence* that it will not pursue. Therefore, a more focused analysis of relationships between agents could take place if we can incorporate knowledge of which regions of the environment an agent's goals refer to into the model of agent interaction with the environment. In order to achieve this, we provide a typology of agent goals with reference to an agent's *Viewable Environment* and *Region of Influence*. However, before we do that we need to differentiate between different types of goals according to whether the goal is to retrieve information from the environment or change it.

5.4.1 Query and Achievement Goals

In the broadest sense agents can have only two types of goals. On the one hand, they may want to *effect some change* in the environment, which implies changing attributes of the environment, while on the other hand, they may just want *some information about the environment*, which does not lead to any direct changes in the environment. Distinguishing between these two types of goals is important since the latter can only be achieved directly by an agent if that goal is in the *Region of Influence* of the agent, while the former can only be achieved if the goal is in the *Viewable Environment* of the agent.

We distinguish between these two types of goals by using the same terminology as the dMARS agent system, which is formalised in [75] using the SMART framework. Essentially, a *query* goal is one for which an agent tries to elicit some information, either from its internal beliefs or from the environment. As such, it can be satisfied if it falls within an agent's *Viewable Environment*. Conversely, an *achievement* goal may require that the agent performs certain actions in order to

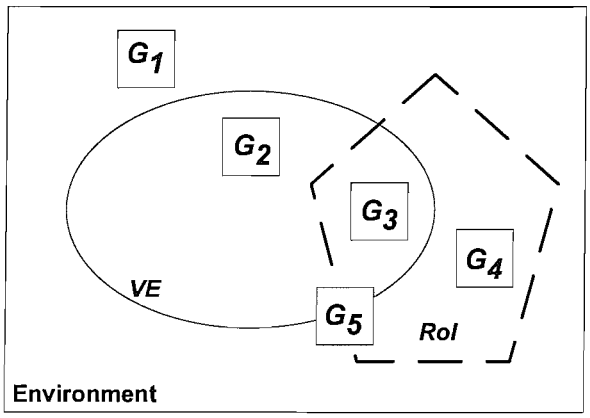

FIGURE 5.9: Types of goals

change the environment, if the environment is not already in the desired goal state. Thus, an *achievement* goal can be satisfied if it lies within an agent's *Region of Influence*.

5.4.2 Goal Regions

We categorise goals according to where within the *Viewable Environment* and *Region of Influence* they occur. The different types are shown in Figure 5.9, where goals, represented with a square labelled by a capital G, are overlayed across the *Viewable Environment* and *Region of Influence* of an agent. The different situations are described below.

No control — G_1 The agent has a goal that describes an environmental state falling outside of both the agent's *Viewable Environment* and its *Region of Influence*. As a result, this agent has no *control* over satisfying that goal, irrespective of whether it is a *query* or *achievement* goal. Some form of cooperation with another agent is essential in this case.

View control — G_2 In this case, the agent can satisfy a *query* goal but not an *achievement* goal, since the goal is within the agent's *Viewable Environment*.

Total control — G_3 A *total control* goal is one that lies within both the agent's *Viewable Environment* and its *Region of Influence*. As a result, regardless of whether it is a *query* or *achievement* goal, the agent can satisfy it.

Blind Control — G_4 In this case, the goal falls within the agent's *Region of Influence* but not within its *Viewable Environment*. As a result, the agent is able to satisfy it if it is an *achievement* goal but not if it is a *query* goal. However, the agent is not able to *verify* the results of its actions.

Partial Control — G_5 Finally, a goal may fall in a region that is partially under the agent's *Viewable Environment* or the agent's *Region of Influence*. In this case, the agent will have some combination of control based on the four types described above.

The *AgentGoals* schema formalises the four main cases above. It includes the *AgentState*, *ViewableEnvironment* and *RegionOfInfluence* schemata, and uses the *viewable* and *regionofinfluence* functions from them.

$$
\begin{array}{|l}
\textit{AgentGoals} \\
\hline
\textit{AgentState} \\
\textit{ViewableEnvironment} \\
\textit{RegionOfInfluence} \\
\textit{canview}, \textit{caninf} : \mathbb{P}\, \textit{Attribute} \\
\textit{none}, \textit{blind}, \textit{total}, \textit{view} : \mathbb{P}\, \textit{Goal} \\
\hline
\textit{canview} = \textit{viewable}\ (\theta \textit{AgentState}) \\
\textit{caninf} = \textit{regionofinfluence}\ (\theta \textit{AgentState}) \\
\textit{none} = \{g : \textit{goals} \mid \neg\ g \subseteq (\textit{canview} \cup \textit{caninf})\} \\
\textit{blind} = \{g : \textit{goals} \mid g \subseteq (\textit{caninf} \setminus \textit{canview})\} \\
\textit{view} = \{g : \textit{goals} \mid g \subseteq (\textit{canview} \setminus \textit{caninf})\} \\
\textit{total} = \{g : \textit{goals} \mid g \subseteq (\textit{canview} \cap \textit{caninf})\} \\
\hline
\end{array}
$$

With the goal typology in place, as well as the interaction typology, we have two significant analytical tools for identifying and characterising possible relationships between agents.

5.4.3 Example Analysis

In this section we present an example that makes use of both the relationship typology and the goal typology to coordinate agents at run-time with limited domain information. This example simply serves to illustrate the main concepts described here, and to provide an indication of how they can be used in a practical application development environment. More detailed examples are developed in Chapter 6, in which the implementation of the ideas in a more extensive application is described.

The aim of the example scenario is similar to the one described in Section 5.2.3, supporting a user to collaborate with other users by sharing information through personal agents. However, the domain is now a computer research lab and the focus is on supporting users with their day-to-day tasks.

An initial analysis of the domain reveals that researchers typically use at least two devices in the lab to achieve their day-to-day tasks, including one relatively powerful desktop or laptop computer, as well as a more limited mobile device, such as a PDA. In order to effectively support the users, the application should allow use of the agent-based system through all user devices, requiring agents to be installed on each device. Furthermore, agents serving the same user through different devices should cooperate closely. Each agent is to take advantage of the connectivity, storage and computing capabilities of their device so as to more effectively support the user.

Given this, there are two main problems to solve from the perspective of enabling the cooperation between a user's devices.

- Firstly, how can coordination and cooperation be effectively supported if there is no clear knowledge, at design time, of the exact capabilities of each of the devices or the exact tasks that they may attempt to carry out, since this depends on the equipment of each user and their individual choices as to how they want to use the system.
- Secondly, how can the infrastructure deal with changes in devices and possible changes in the application requirements as the system develops.

These two problems make it practically impossible to define coordination using any detailed application-specific knowledge, such as the connectivity capabilities of a device, since this information is discovered at run-time, and the type of coordination based on the discovered information must also be decided at run-time.

In this case, the ability to define coordination based on generic relationship types is valuable. The agents belonging to a single user are instructed to communicate, whenever possible, so as to share information on their capabilities and the current user goals. This allows the modelling both of the relationship between them and of the goals that must be satisfied. With such knowledge in place, we can attempt to guide coordination, at run-time, by defining generic rules such as the ones presented below.

- If there exists a goal of type *total control* for only one device, then that device should attempt to achieve the goal, since it is the only one that can both attempt the actions and verify the results. For example, if the user wishes to send a short message to a colleague as a reminder for an impending meeting, and the only device belonging to the colleague

in question is a Bluetooth-enabled PDA, then only the user's PDA is able to achieve the goal.

- If two devices are attempting to achieve the same goal, and that goal lies in a region where they are related through *Bilaterally Observable Mutual Influence* (i.e. where their *Regions of Influence* and *Viewable Environments* overlap) then they could both attempt to achieve the common goal. In this case, we could assign the goal to one of the devices at random. Alternatively, we could use some form of priority that would identify the workstation as having a higher priority for achieving such common goals, since we can make the assumption that its resources will be more readily available and not limited by battery concerns or unreliable connections due to just wireless access. Returning to the example of contacting a colleague, if both of the user's devices are Bluetooth-enabled, the desktop computer could undertake the task so as to avoid consumption of the limited battery life of the user's PDA.

- If a goal is of type *view control* for one device, and *blind control* for the other, which implies that there is some region of the environment in which they are in a relationship of *Unilaterally Observable Actions*, the agents should cooperate, with one performing the actions and the other verifying the results or with one identifying the current state of the environment and the other acting accordingly. For example, the user's PDA may be able to identify dynamically through the Bluetooth protocol which devices are able to provide information on their owners, and are able to provide that information through the 802.11b wireless protocol. The workstation can then use 802.11b to retrieve the profiles. The workstation has *blind control* since it can retrieve profiles but cannot discover the devices, while the PDA has *view control* since it can discover the devices by exploiting Bluetooth but cannot download the profiles due to a lack of storage on the PDA.

For example, while attending a presentation, a user requests that the personal agent on a Bluetooth-enabled device collects all information on the topic of the meeting that is available through other researchers in the lab and downloads any relevant publications. The user then switches off the device, because the battery is running out. Once back at the desk and at the workstation, the mobile device is switched on and communicates wirelessly, or through the usual synchronisation mechanisms, with the workstation. The information on goals and capabilities is exchanged and the two agents identify that while they can both access information on other users, for the PDA-based agent the *Viewable Environment* is limited to just those users whose information is

accessible via Bluetooth-enabled devices that are in range. On the other hand, the workstation is able to access all relevant users through their workstation agents, so it adopts the goal.

Through this basic example, we see how access to a relationship analysis tool that can identify generic relationship types can play a valuable role in facilitating coordination between agents at *run-time*. Given the emerging landscape of computing environments in which constant change and heterogeneity become permanent features, such tools will become increasingly important.

5.5 Describing *Interfering* Relationships

The types of relationships defined in Section 5.3 provide a generic view of relationships where the *intention* of agents is not considered. In other words, they are generic types considered outside of any specific application context. In this section, we illustrate how these definitions can be reused to define a set of relationships given such a specific context, by examining the cases where one agent can *interfere* with the activities of another agent. This analysis serves two purposes. Firstly, it defines a set of relationship that are particularly useful when attempting to define regulatory mechanisms to prevent or to control the interference. Secondly, it illustrates how the generic types defined above can be applied to specific contexts.

Weak influence A *weak influence* relationship occurs when an agent is able to affect aspects of the environment that another agent uses to decide what actions to perform (i.e. aspects of the environment the agent can perceive). Although a weak influence relationship can lead to a different outcome for the influenced agent's goal, it cannot directly affect actions of that agent.

Agent B is *weakly influenced* by Agent A if and only if (i) both agents are able to observe the actions of Agent A, and (ii) there is no mutual influence between the two agents.

The schema below formalises this by including the *BilaterallyObservableActions* and *MutualInfluence* schemata. In the predicate section, we state that for *WeakInfluence* to exist, the *bilaterallyobservableactions* functions must return a non-empty set of attributes, and the *mutualinfluence* function must return an empty set of attributes.

WeakInfluence

$$
\begin{array}{l}
BilaterallyObservableActions \\
MutualInfluence \\
weakinfluenced : AgentState \leftrightarrow AgentState \\
\hline
\forall a, b : AgentState \mid a \neq b \bullet \\
\quad (b, a) \in weakinfluenced \Leftrightarrow \\
\quad\quad bilaterallyobservableactions \neq \{\} \wedge \\
\quad\quad mutualinfluence = \{\}
\end{array}
$$

Strong influence A *strong influence* relationship occurs when an agent is able to affect both the *Viewable Environment* of another agent as well as its *Region of Influence*. In this case an agent can directly affect the goals of another agent because it can act on exactly those aspects of the environment that may represent desirable environmental states for the other agent.

Agent B is *strongly influenced* by Agent A if and only if both A and B can observe the actions of each other.

This situation is covered by the case of bilaterally observable mutual influence, so we simply need to include the *BilaterallyObservableMutualInfluence* schema in the *StrongInfluence* schema, and in the predicate section specify that the function *bilaterallyobservablemutualinfluence* should not be equal to the empty set.

StrongInfluence

$$
\begin{array}{l}
BilaterallyObservableMutualInfluence \\
stronginfluenced : AgentState \leftrightarrow AgentState \\
\hline
\forall a, b : AgentState \mid a \neq b \bullet \\
\quad (b, a) \in stronginfluenced \Leftrightarrow \\
\quad\quad bilaterallyobservablemutualinfluence \neq \{\}
\end{array}
$$

Sneaky influence A sneaky influence relationship occurs when an agent is able to affect the *Region of Influence* of another agent but not the *Viewable Environment*. This, of course, implies that the influenced agent cannot view the results of its actions, so cannot be aware that some other agent is affecting those results.

Agent B is *sneakily influenced* by Agent A if and only if there is a relationship of unilaterally observable mutual influence.

SneakyInfluence

$UnilaterallyObservableMutualInfluence$

$sneakyinfluenced : AgentState \leftrightarrow AgentState$

$\forall a, b : AgentState \mid a \neq b \bullet$
$(b, a) \in sneakyinfluenced \Leftrightarrow$
$unilaterallyobservablemutualinfluence \neq \{\}$

No influence Finally, when an agent cannot affect the *Viewable Environment* or the *Region of Influence* of another agent, no direct relationship can develop between them.

Agent B is *not influenced* by Agent A if and only if (i) there are no observable actions between A and B, and (ii) there is no mutual influence between A and B.

NoInfluence

$ObservableActions$

$MutualInfluence$

$notinfluenced : AgentState \leftrightarrow AgentState$

$\forall a, b : AgentState \mid a \neq b \bullet$
$(b, a) \in notinfluenced \Leftrightarrow$
$observableactions = \{\} \wedge$
$mutualinfluence = \{\}$

These four types of influence can now, in turn, act as a guide to characterise a range of specific kinds of relationships. For example, a competitive relationship for access to common resources can only take place if both agents can strongly influence each other, i.e. if their *Regions of Influence* and *Viewable Environments* overlap. A supervisor-student relationship is one in which the supervisor can strongly influence the student (e.g. by providing direct guidance on what research the student should do), and the student can weakly influence the supervisor (e.g. by generating new results that may convince the supervisor to change research direction).

5.5.1 Effect of Influence on Actions and Goals

In order to have a clearer understanding of exactly how one agent could affect the goals or actions of another in the context of these four types of relationships, we provide an analysis of the different cases. The analysis is based on the assumptions that the goals agents are trying to achieve are of type *total control*, i.e. they are within an agent's *Region of Influence* and *Viewable Environment*.

Having made such an assumption, it is sensible to define the relationships that evolve through the interactions between agents in terms of the contribution that such interactions have towards the achievement of their goals. [3]

Weak influence relationships

When only weak influence relationships occur, the influencing agent cannot directly impact goals. Nevertheless, it can still have a significant effect on the way the influenced agent achieves a goal, or whether the goal can be achieved at all. In essence, an agent could either be influenced so as to change its actions in order to achieve a goal or to change the goal completely. Below, we outline the different scenarios.

Goal does not change In the first type of case, the goal of the agent does not change as a result of the influencing agent. However, the actions performed to achieve the goal might change, as might the exact results of the actions, because of the goal.

No effect The influencing agent has no impact on the outcome of the goal because the attributes of the environment that are affected by the influencing agent are not taken into account for the execution of an action by the influenced agent.

Outcome of action changes Here, the influencing agent affects the environment in such a way that the outcome of the action performed by the influenced agent changes. However, the *goal* of the influenced agent does not change. For example, consider an agent with the goal of compiling a list of all researchers with an interest in the subject of *argumentation*. The goal is satisfied as long as such a list exists. The agent compiles the list by asking other agents to declare their interest or lack of it in the subject. The queried agents influence the *outcome* of the action by providing an answer. In any case, the goal is eventually achieved. However, the exact values described in the list have been influenced by others.

Action changes Agents may influence another agent to such an extent that the later needs to change its planned actions in order to achieve the goal. For example, if some agents refuse to declare whether they are interested in the topic of argumentation,

[3] Note that, if an agent's goal cannot be achieved within that agent's *Region of Influence*, then the agent must seek assistance from another agent that has access to the region of the environment within which the goal can be achieved. In this section, we do not consider those situations.

the agent of the example above may need to follow an alternative route, such as looking at their list of publications for evidence of an interest in the subject.

Goal changes The second type of scenario is when the influencing agent may change the environment in such a way that the influenced agent has to change its goal entirely. For example, let us assume that Agent A has two goals. The first goal, of primary importance, is to discover any paper on *negotiation*, and the second goal, of secondary importance, is to discover papers relating to *middleware*. If A is pursuing the secondary goal and discovers that new papers relating to the primary goal have been posted by B, A must then change goals to reflect the change in the environment. Thus, B has sufficiently influenced A, through actions that impacted on just A's viewable environment, so that A changed its goal.

Strong and Sneaky Influence

Strong and sneaky influence relationships can impact on a goal in a more immediate way since agents could change exactly those attributes that represent a goal for another agent. We identify three main cases below.

No change In the first case, the actions of Agent A do not affect the goal of Agent B. This means that although A is able to act in the *Region of Influence*of B, it does not perform actions that hamper the goal for B.

Goal constrained An agent can perform an action that changes the environment in such a way that a goal of another agent is constrained. For example, one agent may wish to access a document but cannot do so because another agent is already accessing it or has placed restrictions on its access.

Goal aided Alternatively, an agent can perform an action that helps towards creating the goal state of another agent. Such actions may have been intentional or may occur unintentionally. For example, if an agent has the goal of discovering a paper on *auctions* and another posts that paper, it inadvertentently aids the second one in achieving its goal.

5.6 Conclusions

In this chapter we introduced a typology of relationships in support of coordination and regulation, building on a basic model of interaction between agents and the environment. Each relationship type was associated with the goals of an agent, by defining goal regions to provide a useful tool for identifying possibilities for coordination between agents, especially in situations in which we cannot predefine coordination because of incomplete information about an agent's capabilities and goals. The same tools can also be used to identify how a multi-agent system should be regulated to avoid conflicts, as illustrated through the definition of interfering relationships.

5.6.1 Related Work

The issue of relationship analysis has not in general been sufficiently addressed by existing research. Initial attempts such the ISAAC automated team analyst [151] take a different approach, since the analysis tools are geared towards learning about agent behaviour, and are focused on analysing teams of agents. Although there is also a wider body of work on conflict management (a representative collection can be found in [210]), once more the relationship identification issue is not addressed. Our work also has some similarities with social dependence networks [50, 196], which were also modeled using SMART [76]. However, our approach differs, since we make minimal assumptions about other agents, basing our models solely on agent interaction with the environment and the observability of actions. In the next sections we compare more closely the ISAAC automated team analyst to our work and social dependence networks.

ISAAC

ISAAC is a system developed to analyse the interactions between agents in a team. It performs the analysis by examining data-traces produced during the execution of a system (typically a game between two teams of agents in the context of ROBOCUP), considering individual actions, patterns of interaction and statistics of engagement between teams. ISAAC attempts to create models of agents by learning from the traces of their actions, highlighting positive or negative actions with reference to an overarching goal. Developers can then analyse the resulting models

to identify relevant patterns and perform "what-if" analyses to determine the performance of the team.

The main difference between ISAAC and SMART+ is that while our analysis is based on a *pre-defined* abstract agent model and an interaction model that provides generic relationship types which are then related to real-world situations, ISAAC attempts to *learn* the agent models after agent interactions have taken place in order to then inform subsequent refinements to behaviour. Furthermore, ISAAC is geared towards analysis of *team* behaviour, and particularly improving one team's performance against another, while we focus on interactions between two agents and are concerned with improving the overall performance of an agent system.

Overall, although both techniques are aimed at facilitating relationship analysis, they clearly fulfill different roles. ISAAC is particularly well suited for analysing interactions once they have taken place, while our aim is to inform both design from the very first stage and the run-time coordination of agents.

Social Dependence Networks

Social dependence networks [196] underpin the computation model for social power theory, as proposed by Castelfranchi [50]. These are taxonomies of social relationships in which relationships are characterised by the *power* that one agent has over another. They facilitate reasoning about relationships so that agents can reason about inter-dependencies between them. Comparing social dependence networks to our own relationship analysis techniques is particularly interesting, since both are expressed using SMART [82]. This provides a further illustration of the benefit of having access to a common set of concepts, since we can be sure that we are comparing like with like.

The taxonomy of relationships is based on whether an Agent A depends on another Agent B to achieve a goal with respect to: B being able to perform an action A cannot perform; B having access to an agent A does not have access to; or both. The *dependence network* is then defined by the combinations of distinct *dependence situations*. These situations between agents are: mutually dependency (both depend on each other for the same goal), reciprocal dependency (agents depend on each other, but for different goals), unilateral dependency (just one agent is dependent on the other), and independence (an agent does not depend on anyone else). However, in order for such dependencies to be identified, we must have access to both the goals and plans

that an agent has to achieve those goals. It is the plans that indicate which actions or which other agents are required to achieve goals. This means that we must make assumptions about the *internal* operation of agents, namely that agents operate based on plans and that we have access to those plans.

Our relationship analysis models base their analysis on the *Viewable Environment* and *Region of Influence* of an agent, making it possible to make useful inferences without any reference to the internal structure of an agent. Furthermore, the goal typology is also based on where a goal lies within the *Viewable Environment* and *Region of Influence*, rather than what a plan explicitly defines as the necessary actions. As a result, the approach is more flexible and more widely applicable.

5.6.2 Discussion

The relationship analysis tools presented here can play an important role in managing dynamic and heterogeneous computing environments. At design-time they can aid developers in determining the most appropriate configurations of agents and how they can facilitate cooperation between them. At run-time they can enable agents themselves to reason about relationships between agents and adjust behaviour accordingly. This can be achieved both by specialised agents that are dedicated to the task of relationship analysis (we specify such an agent in Chapter 6), and by individual agents making use of the relationship analysis techniques.

Finally, these tools also demonstrate the utility of the earlier models as an enabling conceptual infrastructure for dealing with heterogeneous agent systems. The clear definitions of concepts such as attributes, goals and capabilities enable the formal definition of the intuitive notion of regions of the environment that agents can view and influence, based on models of agent perception and action.

Chapter 6

Applying *act*SMART, SMART and SMART+

"You cannot create experience. You must undergo it."
Albert Camus (1913-1960); French writer and philosopher

6.1 Introduction

With the work presented in Chapters 3, 4, and 5 on models of individual agents and the relationships between agents, and with the underpinnings provided by SMART [82], we have access to a considerable base of *conceptual infrastructure* to facilitate the implementation of a multi-agent application. Throughout the thesis we have provided several isolated examples of how such infrastructure can be used in a practical application setting, but we have not provided a *comprehensive* view of all the models operating together, something that is necessary to provide a more complete evaluation of the work. This chapter addresses this through the development of a demonstration application, where we make use of both *act*SMART to define architectures for the various agents in the application, and SMART+ to reason about the relationships between agents.

In particular, we develop a demonstration application inspired by the vision of *ubiquitous computing*, which refers to attempts to develop applications in which the use of computing technologies providing complex, integrated services is hidden as much as possible from the user. Weiser, who coined the term, described ubiquitous computing as technologies that "weave themselves

into the fabric of everyday life until they are indistinguishable from it" [223]. In practice, ubiquitous computing can be described as the exploitation of limited capability devices such as mobile phones and PDAs, along with computing capabilities embedded in devices such as printers, to provide services to the user that are tailored in relation to information such as the *location* of the user or the *context of use*.

With the most recent advances in computing and networking technologies, this vision is becoming increasingly viable [191] and, as a result, there are several efforts to develop models to support the vision of ubiquitous computing within the context of existing IT infrastructure (e.g. [109, 145, 173]).

The domain of ubiquitous computing provides a natural setting for the application of agent-based computing, and a realistic challenge to test the viability of the paradigm, since the environment is inherently heterogeneous and dynamic due to the inevitable continuous movement of users and devices within the environment. Indeed, this is demonstrated by the fact that there have also been several efforts attempting to address the challenges raised by ubiquitous computing using agent-based methods (e.g. [24, 52, 94, 98, 125, 126, 190, 193]).

Of course, our aim is not to provide answers to all the questions that ubiquitous computing raises. However, through a limited demonstration within the context of ubiquitous computing, we illustrate how our models can practically contribute towards both agent construction and the management of relationships between agents. The application scenario revolves around the provision of services to delegates attending a conference and support for collaboration between delegates through the exchange of personal information and resources such as papers and presentation material. We assume that delegates are on site at a conference venue, with various rooms for presentations, and public spaces for interaction with other delegates. The services provided to users are either access to physical devices, such as printers and projectors, or access to services that can provide information about local restaurants and accommodation. Users are represented by dedicated *user agents* operating on their devices, and the services provided are accessed through appropriate agents for each type of service. Furthermore, the system is supported with *infrastructure* agents whose aim is to facilitate cooperation and coordination between user and service agents. In particular we examine, through the application of our models, the following specific aspects of development and support for agent-based systems.

1. Firstly, we investigate and contrast the operation of *infrastructure* agents, whose sole purpose is to facilitate the run-time discovery of information about other agents and the oper-

ating environment in general. This information can then be used to promote coordination and cooperation between agents, through the enforcement of regulations or the support for direct interactions between agents in order to achieve their goals. These capabilities are crucial for the effective operation of a multi-agent system and, not surprisingly, almost all major toolkits provide agents to support this.

We term such agents *infrastructure agents* and treat them as distinct from *application agents*, with the latter achieving particular application-specific tasks. In particular, we investigate two types of such infrastructure agents:

- *middle agents*, which perform capability brokering, with their specification based on existing research on such agents, as discussed in Chapter 2;
- and, *relationship analysis (RA) agents*, which perform relationship analysis and management using the techniques developed in Chapter 5.

Note that while middle agents have been widely studied and are widely used, *RA* agents are only made possible because of the work in this thesis and as such represent a new type of infrastructure agent. Through their comparison we can determine the benefits that each brings to an agent-based application.

2. Secondly, we discuss in some detail the development of application-specific agents, touching on issues such as the practical implementation of abstract concepts like attributes, and how architectures can be reconfigured at run-time.

3. Finally, we describe our implementation of *act*SMART as a set of application programming interfaces, which enables the practical construction of agent architectures.

The application is simulated, in the sense that we do not make use of actual mobile devices and devices with embedded computing capabilities. The communication between agents and the use of different underlying communication protocols such as Bluetooth or 802.11b are also simulated. However, the agents operate as independent entities within the simulation environment and the tools used to develop agents are the same tools that would be used for the development of applications for mobile devices. We have also tested some of the agent implementations, without the communication capabilities, on low-end and high-end PDAs, so as to verify that the implementation was operational on actual devices.

The chapter begins with a discussion on *middle agents* as the first type of infrastructure agents we investigate. We introduce Decker et al.'s [69] model for middle agents, and adapt it for use

within SMART, before developing a specific architecture for a broker. Subsequently, we discuss relationship analysis and develop a specific architecture for an *RA* agent. The two architectures are then contrasted, enabling us to identify some of the main differences between the two types of infrastructure agents as well as where each would be most suitable. With the infrastructure agent architectures in place, we proceed to develop the application scenario, discussing how the different types of agents are related and how we can characterise them given the entity hierarchy of SMART, and our refinements of it with active and passive agents. We then develop the architecture for the user agent [1] to provide a clear example of the use of *act*SMART in architecture development and the capabilities it affords us to reconfigure architectures. With the user agent architecture in place, we then discuss the use of *RA* agents to facilitate cooperation between user agents. Finally, we outline the actual implementation of *act*SMART in Java and conclude.

6.2 Middle Agents

The need for obtaining run-time information about a multi-agent system has long been recognised and was initially characterised as the "connection problem" [67]; namely, how agents can find out about other agents and capabilities they may offer. There is already a wide range of solutions to this problem, centring around the use of *middle agents* for *capability brokering*, defined as "the task of finding an agent which has a capability that can be used to address a given problem" [225]. More specifically, Decker et al. [69], define middle agents as those agents that act neither as providers nor as requesters of information, *nor* perform any other actions, and instead act as the 'connectors' between providers and requesters by managing information required to enable one agent to access another or cause the other to perform an action. As discussed in Chapter 2, toolkits for agent development provide a variety of alternative middle agents designs, and there are also several description languages for agent capabilities and protocols for the advertisement of those capabilities.

We clearly need to support the development of middle agents within the context of *act*SMART, SMART and SMART+ since, as we discussed above, they fulfill a crucial task as part of a multi-agent system. However, since there is already a wealth of existing work, we do not aim to develop yet another model for middle agents but instead choose to adopt a well-established and

[1] Note that some aspects of the architecture specifications are only outlined in this chapter, with more extensive descriptions in Appendix A.

widely-used existing model, developed by Decker et al. [69], and illustrate how it can inform the specification of an agent architecture using *act*SMART.

Decker et al. describe *discovery* as the process of matching a *set of preferences*, as defined by a *requester*, against a set of *capabilities* that a *provider* offers. The task of the *middle agent* is to perform this matching between requesters and providers.

In their work, they provide a comprehensive categorisation of such *middle agents* according to the information that providers, requesters and middle agents have available about services requested and available capabilities. The three most significant types are: a *broadcaster* where capabilities and preferences are made available for all to see; a *matchmaker* or yellow pages service where capabilities are known by all but preferences are only known by the requester; and a *broker* where only the middle agent has knowledge of both capabilities and preferences.

We adapt this model to the SMART framework by replacing the generalised notion of preferences with a description of an agent using attributes, capabilities, goals and motivations, and we call this a *profile*. Agents can either provide a middle agent with a *wanted profile*, which describes the type of agent they are attempting to discover, or a *provider profile*, which describes the services provided. Below, we provide a detailed account of the structure of a *profile* and outline examples of the types of requests it can be used to describe.

6.2.1 Agent Profile

We examine each aspect of an agent in turn, so as to determine both what is required to define a wanted or provider profile as well as what is afforded by the expressive capabilities of SMART. The aim is to capture what an agent is able to query with reference to other agents or, say, about itself, starting from the basic understanding of agents as entities described by attributes, capabilities, goals and motivations.

Attributes Attributes describe the observable features of the environment in general. In the context of the profile of a specific agent, the attributes contained should relate just to that agent. These attributes can be of two distinct types, since they can either describe the agent directly or the agent's situation in the environment. The distinction is useful since it enables the modeling of more subtle profiles, such as that an agent belongs to a specific person and is currently in a particular location in a particular building. The first

requirement refers to an attribute specific to the agent, while the last two refer to attributes that are a result of the agent's situation. Attributes within a wanted profile indicate that an agent requires another agent matching those attributes, while within a provider profile they indicate the attributes of the provider.

Capabilities Capabilites describe the *actions* an agent is able to perform and, as such, are the most typically requested type of information about other agents, since it is the capabilities of an agent that determine, above anything else, whether the agent can help to achieve a goal. Once more a wanted profile specifies required capabilities while a provider profile specifies offered capabilities.

Goals Goals describe a desirable state of affairs that an agent wishes to achieve. A wanted profile containing goals indicates that an agent requires assistance to achieve those goals, while a provider profile indicates that the agent is able to achieve the goals specified. Matchmaking based on an agent's goals is generally not addressed in agent toolkits, since the focus is on matching capabilities. However, the ability to match by goals can be very useful in cases in which an agent knows the environmental state it wishes to bring about but does not know what capabilities of other agents are necessary to achieve that.

Motivations Finally, an agent can also include a description of its motivations within a profile. Such information can be used to select agents that are more likely to behave in a certain way. For example, we may specify in a wanted profile that a provider agent should be motivated to cooperate as much as possible, so as to benefit from the greater possibility of forming closer relationships.

Although each of the information types described above can be used in isolation to form a useful query, the real benefits come from *combining* them to create more complex queries. For example, an agent may wish to discover an agent that is currently in the same location and has the capability to use the printing devices that are close to that location.

Finally, agents can also indicate the *type* of entity they wish to discover in terms of passive, active and autonomous agents. It is entirely possible that both an autonomous and an active or passive agent are able to offer the same services. It is therefore useful to specify that, for example, an agent does not wish to interact with an autonomous agent that could refuse service provision or could provide less control as to how the goal will be achieved. Engagement of a passive agent would guarantee that the engaging agent would have absolute control over the

Descriptive Specification	Behavioural Specification	Structural Specification
Attributes Middle Agent Type Service Type Handled Location Handled Wanted Profiles Provider Profiles **Capabilities** Register Wanted and Provider Profiles De-register Wanted and Provider Profiles Match Profiles Notify Agents of Match **Goals** Match Wanted to Provider Profiles Notify Service Requestors of Match Notify of Profile Registration Expiration	Register Wanted or Provider Profiles Match Profiles Notify Service Requestors of Match Periodically check for expiration of profile registration Notify agents of expiration	ServiceRegistration (Sensor) IncomingQueries (Sensor) RegistrationManager (Controller) ProvidedServices (Infostore) WantedServices (Infostore) QueryManager (Controller) RegistrationMessages (Actuator) QueryReplies (Actuator)

FIGURE 6.1: Broker specification

actions performed by the service provider, while engagement of an active agent would provide guarantees that the engaged agent would attempt to achieve just that goal and would not change goals because of influences such as motivations.

6.2.2 Broker Architecture

Following the general discussion on agent profiles, we present here a specific architecture, which is aimed at supporting agent discovery. The architecture is for a broker, so only it has knowledge of both wanted and provider profiles. The description of the architecture is divided, as usual, into the descriptive, structural and behavioural specification, with an overview of all three aspects in Figure 6.1. Here we provide just the descriptive specification and a view of the architecture in Figure 6.2, while the structural and behavioural specifications are presented in A.2.

Descriptive Specification

Recall that the descriptive specification of an agent provides first a description of the attributes used by an agent and second a description of the agent, the agent's capabilities, goals and, if they exist, motivations.

Attributes The attributes describing a middle agent indicate the type of middle agent, such as

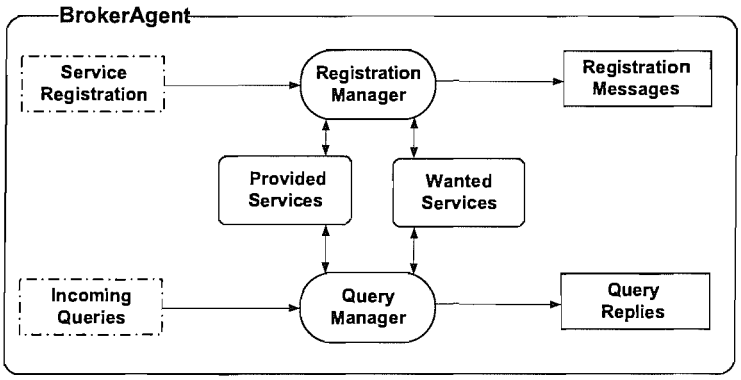

FIGURE 6.2: Broker architecture

broadcaster, *matchmaker* or *broker* (in our case this attribute indicates that it is a broker). If required by a specific domain, we could also describe the category of services that can be registered with the middle agent. For example, in our ubiquitous computing scenario, we have different middle agents for different locations or different types of devices. Finally, profiles are used as discussed earlier, and are supplemented with an additional attribute indicating the time period for which a profile should remain registered with the middle agent.

Capabilities Our broker has the capability to both register and de-register provider or wanted profiles, and to accept queries for provider services. In addition, the broker notifies agents whenever their provider or wanted profile registration period has expired, to give them the chance to renew it. Finally, it notifies agents of matches between profiles.

Goals The three main goals of the agent are to register the profiles corresponding to wanted and provided services (for a defined time-period after which registrations are removed), to match those profiles, and to notify first, service requestors of a match, and second both service requesters and providers that their registration will be removed unless renewed.

6.2.3 Discussion

Even though the broker described here allows us to perform only very basic capability matching, it provides a clear example of how such agents can be implemented and specified using SMART and *act*SMART. While such agents exist within the majority of agent toolkits, such as ZEUS [155] or RETSINA [206], their architectures are typically not made explicit and are inconsistent with the approach used to develop architectures for other agents in the system, which makes their design opaque to application developers and harder to reuse in other settings.

6.3 Relationship Analysis Agent

In Chapter 5, we indicated that the relationship analysis tools developed there could be used both at design-time by the designer of a system and at run-time by agents. Here, we develop an architecture for an agent dedicated just to the task of relationship analysis that complements the functionality of the middle agent.

The task of the *RA* agent is to identify situations in which there may be a conflict between agents, or where there may be possibilities for cooperation. Before we present the detailed architecture, we consider exactly what information is required from agents in order to perform the analysis, how the *RA* agent performs this analysis, and what management can be applied based on the analysis.

6.3.1 Identifying Agent Relationships

Required Information for Analysis

In order for the *RA* agent to identify the possible relationships an agent may have with others, it requires information about an agent's capabilities and attributes. The agent's capabilities define the *Viewable Environment* and *Region of Influence* of an agent, while the agent's attributes are required because they may provide relevant information about the current situation of the agent within the environment, such as its location. This information may, in turn, also impact on the agent's *Viewable Environment* or *Region of Influence*. For example, the short-range Bluetooth communication protocol only allows an agent to access devices that are in close proximity with it. Thus, in order to determine whether an agent can interact with Bluetooth devices, we need to know both that it has a Bluetooth capability and its location to determine which other devices may be in range.

Clearly each agent must provide this information and there are two ways in which it can do so. On the one hand, it can simply provide an agent profile, as discussed for middle agents above, and the *RA* agent can then infer (based on information about the domain) which attributes the agent can influence or view. On the other hand, agents can directly provide the sets of attributes that they can influence or view. In our case, we adopt the former approach since we assume that individual agents may not be able to define their own *Region of Influence* and *Viewable Environment*. Individual agents could either explicitly construct the profile themselves by having

appropriate controllers as part of their architecture, or the task can be delegated to the agent shell, as discussed in Chapter 4. The *RA* agent makes use of domain specific information that maps capabilities to the attributes they can influence or view. Finally, we note that the profile may also contain information about an agent's goals, which, as discussed in Chapter 5, can be used to better determine the relationships that may actually take place.

Now, since changing goals and changing attributes may impact on the resulting relationships, the *RA* agent *updates* profiles from agents. For some types of agents, the information about goals is likely to change frequently, while other information remains relatively static. The only instance in which information about capabilities and attributes would need to be updated is if an agent has undergone reconfiguration of its architecture, or if some of the relevant attributes have changed significantly. For example, in our application, an agent's location is simply determined by the room within the conference venue in which the agent is currently situated. However, other applications may demand a finer-grained approach to the problem, which would require agents to frequently update their location attribute information or the *RA* agent to be more proactive in gaining such information through other means, such as dedicated services for determining the location of agents.

Relationship Analysis

With the required information in place, we can proceed to identify the relationships between agents. The aim is to produce a table, as shown in Figure 6.3, that lists all the agents related to a specific agent according to the type of relationship. In Figure 6.3, Agent A is related to two agents, B and C. With B the relationship is one of *ObservableActions*, which means that an agent can observe the actions of another. In fact, in this case it is *BilatterallyObservableActions*, which means that both agents can observe the relevant *Region of Influence*. With C the relationship is one of *MutualInfluence* and, more specifically, *UnilaterallyObservableMutualInfluence*, which means that just B is able to observe the region of *MutualInfluence*. The process through which we arrive at such results is described below.

1. The first step is to identify the *Region of Influence* and *Viewable Environment* for the agent in question. This is achieved through an analysis of the agent's capabilities, attributes and the use of domain-specific information.

2. Then, we match this *Region of Influence* and *Viewable Environment* for each other agent

Agent identification: Agent A	
Relationship Type	**Related Agents**
MutuallyViewableEnvironment	
ObservableActions	Agent B
InvisibleActions	
BilaterallyObservableActions	Agent B
UnilaterallyObservableActions	
BilaterallyInvisibleActions	
UnilaterallyInvisibleActions	
MutualInfluence	Agent C
ObservableMutualInfluence	
InvisibleMutualInfluence	
BilaterallyObservableMutualInfluence	
BilaterallyInvisibleMutualInfluence	
UnilaterallyObservableMutualInfluence	Agent C

FIGURE 6.3: Relationship table for a single agent

for which the *RA* agent has the required information. A match occurs when one of the conditions for a relationship to exist is met. The conditions for a relationship to exist are all described in Chapter 5.

3. Finally, the table is stored for later reference and updated whenever relevant attributes or capabilities change.

6.3.2 Managing Relationships through Regulations

Having constructed tables of relationships between agents as discussed above, the *RA* agent must then identify which relationships indicate that some form of *relationship management* is required and apply appropriate management. In our case management is applied through *regulations*, which will allow us to describe what behaviour agents should exhibit once the need for management is identified. Below we provide an outline of the structure of a regulation, which is based on largely accepted notions of regulations as reviewed in Chapter 2.

Environmental Activation Criteria Environment activation criteria stipulate *when* a regulation is applicable by defining a set of attributes. In our case, the activation criteria are based on just the existence of particular types of relationships between agents.

Agent Activation In order to also deal with the activation of a regulation that depends on an

agent profile, rather than just the relationships the agent has with others, we introduce an agent activation component to the regulation. This is described through a profile, allowing the specification of agent attributes, capabilities, goals and motivations.

Forbidden and Mandatory Goals Finally, the regulation defines which goals are not permitted and which must be achieved by the agents that fall under the scope of the regulation. In essence, this provides goals that the agents should not pursue and goals that they should achieve given the regulation.

A regulation is applicable when the relevant relationships, defined in the environment criteria section, are identified and the agents participating in the relationship match the profile in the agent activation criteria. If no environmental activation criteria are defined, but only an agent profile then any agent matching that profile falls under the scope of the regulation. Similarly, if we do not define a profile but just the environmental activation criteria, then all agents identified to be in the specified relationships fall under the scope of the regulation.

The *RA* agent manages relationships between agents by determining which *regulations* apply to a specific identified relationship and informing the agents of those regulations. For the purposes of the demonstration we use two types of regulations: *mandatory* regulations that are typically used to prevent conflict, and *optional* regulations that are typically suggested where there is the possibility of cooperation between agents. Note that this notion of regulation is by no means complete but is adequate to demonstrate how *RA* agents can be used to manage relationships.

6.3.3 Relationship Analysis Agent Architecture

In this section we develop the specification for the *RA* agent architecture, using *act*SMART. As usual an overview of the descriptive, structural and behavioural specification is available in Figure 6.4, while a view of the architecture is provided in Figure 6.5. Once more we only provide the descriptive specification here, with the structural and behavioural specifications in A.3.

Descriptive Specification

Attributes The *RA* agent is described by the location within which it operates and the types of agents it analyses relationships between. For the purposes of this chapter, and our demonstration application, we focus on the relationships between user agents at the conference

Descriptive Specification	Behavioural Specification	Structural Specification
Attributes Location Handled Agent Types Analysed Agent Profiles Regulations **Capabilities** Accept Agent Profiles Update Agent Profiles Analyse Profiles Identify Relationships Notify Agents about Regulations **Goals** Analyse information to identify relationships Notify agents of relevant regulations **Motivations** SupportCooperation ReduceConflict FosterRelationships	Accept an agent profile Analyse profile to identify *RegionOfInfluence* and *ViewableEnvironment* Identify relationships between agents Notify agents of regulations If required update agent profiles and adjust identified relationship information	AgentRegistration (Sensor) UpdateInformation (Sensor) RegionIdentification (Controller) GoalCategorisation (Controller) RelationshipAnalyser (Controller) Conflict Analyser (Controller) CooperationAnalyser (Controller) MotivationEvaluation (Controller) DomainInformation (Infostore) RegionStore (Infostore) RelationshipStore (Infostore) GoalStore (Infostore) RegulationInformation (Infostore) RegulationsNotification (Actuator)

FIGURE 6.4: Relationship analysis agent specification

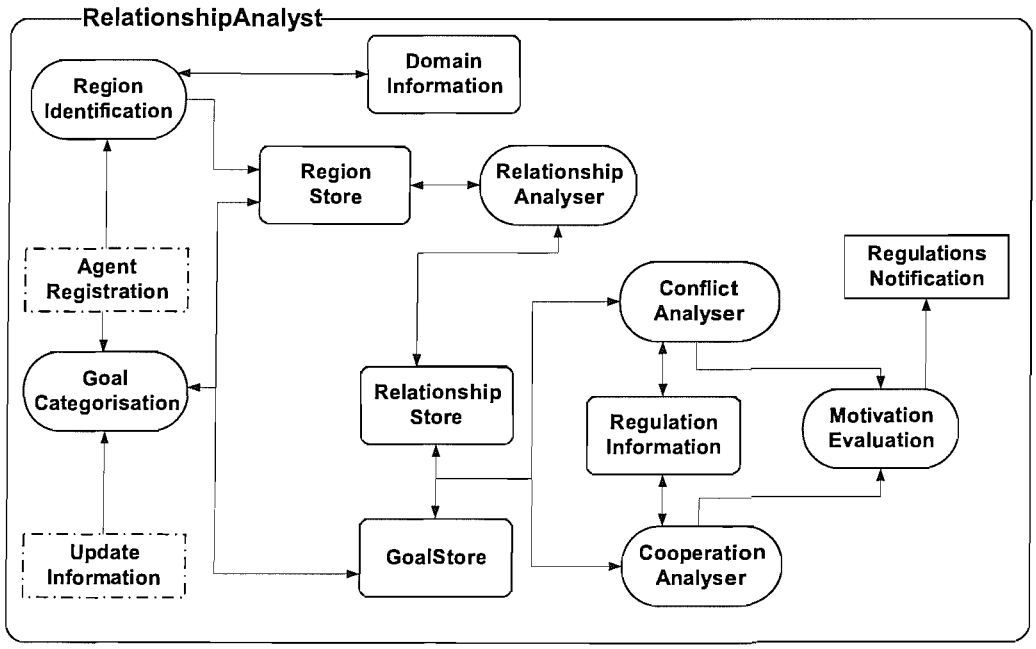

FIGURE 6.5: Relationship analysis agent architecture

venue, with each *RA* agent operating within a specific room. The architecture manipulates profiles, which contain agent attributes, capabilities and goals, and regulations, as described above.

Capabilities The *RA* agent is able to accept agent profiles, either when an agent is registering, or when providing an updated profile. This information is analysed, and agents are notified of relevant regulations.

Goal The goals of the *RA* agent are to analyse the information provided and, if relevant, to notify agents about any regulations they should adhere to. The exact goals at each moment are determined by the motivations of the *RA* agent, as discussed next.

Motivations Based on its motivations the *RA* agent chooses which agents to notify about what regulations. In an abstract sense, motivations are understood as *desires* that an agent attempts to satisfy by generating relevant goals. In our specific case, motivations are satisfied if an agent performs a goal that contributes some *utility* towards a motivation. We define utilities as simple numerical values attached to regulations to indicate how much a specific regulation (and, by consequence, the act of informing an agent about that regulation) contributes towards a motivation.[2] For example, if the *RA* agent is motivated to support cooperation that motivation is satisfied if the relationship analysis agent generates goals that inform agents about regulations in which the goals will lead them to cooperate. Regulations may contribute towards several motivations at the same time, so there is a process of selection to choose those with the highest overall utility at any given time.

For the purposes of the demonstration application, the *RA* agent has three motivations, described below.

- The **SupportCooperation** motivation leads the *RA* agent to choose regulations that require agents to attempt to cooperate in order to collectively achieve their goals. For example, in the previous example of the table of relationships between agents A and agents B and C, the *RA* agent identified that A and C are in a relationship of *UnilaterallyObservableMutualInfluence*. This means that C is able to observe the results of actions in that region but A cannot, although they can both perform actions there. Therefore, C could aid A by verifying the results of actions taken in that region, and informing A of them.

[2] In our case these values are defined at design time and remain static throughout the operation of the *RA*, unless the designer updates them. Note that more sophisticated frameworks supporting motivated agents, based on the SMART framework have been developed and could be used to inform application development [148].

- The **ReduceConflict** motivation leads the *RA* agent to choose regulations that prevent possible conflicts between agents.
- The **FosterRelationships** motivation leads the agent to choose regulations that foster relationships between researchers by exchanging their personal profiles.

6.4 Middle Agents and Relationship Analysis Agents

We have described architectures both for a broker, which makes use of well established techniques adapted to SMART, and for an *RA* agent, which makes use of the tools described in Chapter 5. There is clearly some overlap of functionality between the two types of agents since they both aid better coordination and cooperation. However, it is important to clearly identify how the agents differ in order to better inform the choice between them.

- Middle agents are essentially reactive, since they only reply when given a specific request by agents with some knowledge of what they are seeking. Relationship analysis agents, on the other hand, are proactive since they notify agents of opportunities for cooperation and attempt to prevent conflicts.
- Middle agents only make use of profiles to match service providers to service requesters. As a result, the functionality they can offer is limited to determining whether two profiles match. *RA* agents make use of the relationship and goal typologies, and can use more generic rules relating to types of relationships between agents, or types of relationships and goals.

Based on these differences, it is apparent that middle agents should be used when agents can identify their service requirements and can proactively communicate with middle agents in order to request a suitable match. *RA* agents should be used to proactively identify opportunities that individual agents may not have known were possible.

Furthermore, *RA* agents can promote an overall type of behaviour within a multi-agent system by focusing on issues such as increased cooperation. The use of motivations to make the agent autonomous with regard to what regulation to apply means that the *type* of motivations the *RA* agent has may significantly influence the resulting behaviour of the system. For example, an *RA* agent seeking to avoid conflict above all other costs but with little regards to promoting

cooperation will have a different impact to one that pays equal attention to both motivations. Therefore, different types of *RA* agents, in terms of their motivations and the way regulations are judged to satisfy those motivations, can lead to significantly different types of system behaviour.

6.5 Application Overview

So far we have introduced, in some detail, the architectures of agents providing some of the required supporting *infrastructural services*. The broker agent enables agents to discover suitable service providers, while the *RA* agent supports better coordination and cooperation between agents. We now turn our attention to the specific application scenario of supporting delegates attending a conference, discussing the nature of the other agents, and in particular the user agents, and the relationships between different types of agents.

As we mentioned previously, the services available to delegates at the conference venue are divided into two broad categories, as described below.

Information services provide information relevant to the conference, such as times of presentation, local restaurants and accommodation, and transport facilities. Information services are accessible from anywhere within the conference venue.

Physical services represent those services that are more tangible, such as fax machines, projectors, printers, and so forth. These are devices for use at the conference venue to accomplish specific tasks. Physical services can *only* be available at specific locations within the venue.

In the application domain, delegates are equipped with a range of different types of device, from limited capability mobile phones to more powerful laptop computers. Agents are installed on the devices that attempt to make the best possible use of the capabilities of the devices in order to provide access to as many services as the user requires. In addition to facilitating access to the services provided at the conference, agents can establish contact with other delegates' agents through the exchange of information such as research profiles, publications, or presentation material.

Finally, we assume that there are several lower-level network communication protocols ranging from the short-range Bluetooth protocol to 802.11g wireless connectivity and wired ethernet

connectivity, with devices being able to use some or all of these protocols.

Now, a typical usage scenario within the conference venue could develop as follows. A user arrives with a PDA with Bluetooth capabilities and an agent installed on it. The PDA is identified by a Bluetooth-enabled desktop computer, which is dedicated to the task of registering agents upon their arrival. Once registered, agents can identify themselves to other agents and use the available services. These other agents and services can be discovered dynamically, by taking advantage of any supporting middleware infrastructure, through the use of brokers, or based on direction from *RA* agents.

Next, the user provides its agent with the goal of retrieving information about suitable accommodation for that night, based on their preferences. In response, the agent attempts to locate a broker and request a relevant service. Once a suitable information service is identified, it is engaged by the agent and the required information is retrieved.

Subsequently, the user enters the main meeting room at the conference venue, at which point the user's presence is registered with an *RA* agent. The description of the agent is matched against those of other agents at that location so as to identify any possible conflicts or possibilities for cooperation. The analysis by the *RA* identifies that the user's presentation for a workshop the user is attending is within the *Viewable Environment* of others and a regulation is activated to limit access only to those delegates who are attending the same workshop, so as not to overload the user's PDA with requests for the presentation.

Finally, the agent may request the profiles for any participants matching the research interests of the user, and attempt to locate them or notify the user when they are identified in the same room as the user.

6.6 Application Entities

In this section we provide an overview of all the entities within the demonstration application, discuss their type based on the abstract agent model, and the possible relationships between them. A more detailed explanation of the supporting infrastructure for all agents is given in A.1.

The central conceptual artifacts of the SMART and SMART+ framework are entities and the attributes used to describe those entities. In turn, such things as capabilities and goals from SMART, and *Region of Influence* and *Viewable Environment* from SMART+, are defined by their

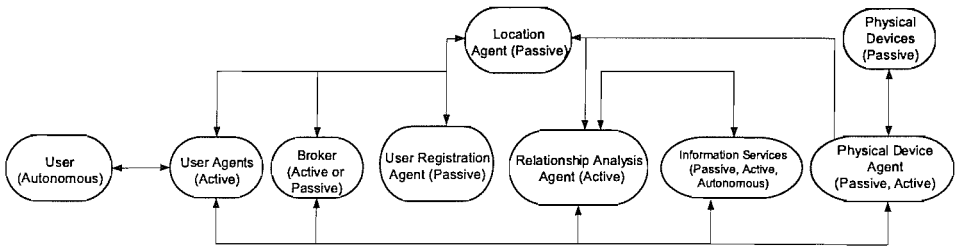

FIGURE 6.6: Entities within the conference application

relationship to entities and attributes. For example, capabilities can change or retrieve the values of attributes, and goals are defined as a set of attributes representing a desirable state of affairs, while *Regions of Influence* and *Viewable Environments* represent the sets of attributes that agents can change or sense. It is therefore natural, when attempting to develop an application using SMART and SMART+, to begin by defining a model of the application through the entities that are contained within the application and the attributes that can describe those entities. We then refer to this model when describing the capabilities of individual agents, defining how they can affect or retrieve information about other entities in the environment.

All the types of entities we consider are shown in Figure 6.6, in which the connecting lines between entities indicate possible interactions between the different types of agents. Interaction between agents takes place through the exchange of messages along a communication channel. Although the communication channel could be explicitly modelled as another entity within the application we have chosen not to do so, since communication protocols are considered to lie at a lower infrastructural level. Instead, the ability to communicate, and access to communication channels, is implicit in the actuator and sensor capabilities of agents. For example, if a device is only able to communicate using Bluetooth, then in order for agents to use it they must have an actuator able to interact using the Bluetooth communication protocol.

As mentioned earlier, each of the entities in Figure 6.6 is described by a set of attributes. Note that we also model *non-computational entities*, over which we have no control such as the user and physical devices, which are *represented* by user agents and physical device agents. However, from the perspective of system description, the distinction between the two is required so that we can identify which attributes are associated with the user or physical device, and which are associated with the agent representing them.

User The user is described by a *user name*, *affiliation* to an *institution*, and *research interests*. If the user is presenting a paper at the conference, they may also have information about the

time and *location* of the *presentation*. Users are considered as autonomous agent entities within this application, since they kickstart the operation of the system by generating goals and providing them to the user agents.

User Agent The user agent is described by its *user identification*, which provides the *user name*, *affiliation* and a *conference registration number*. In addition, the user agent may have a *location*, *resources* such as *papers*, *presentation slides* stored on the device, or *URL links* to online resources. Finally, user agents have *regulations* that define the goals they are allowed or prohibited from performing within a specific context.[3] User agents are considered always to be active server agents, since they are always engaged by the user.

Brokers Brokers have an *identification* and a *specialisation*. Their specialisation simply describes the kind of service they can offer agents. For example, they may provide brokering just for information services dealing with accommodation. Brokers are neutral objects which, once engaged by an agent, can become either passive server agents or active server agents according to their individual architecture. In our case, once activated, the broker becomes an active server agent, since it applies its own matching algorithm through a controller. A different type of middle agent may be best represented as either an autonomous agent or a passive agent. For example, a broadcaster would be best represented as a passive agent since it simply makes available information without matching services to requests within its architecture.

User Registration Agent The User Registration agent performs the very basic role of simply providing a user agent with a *conference registration number* upon the user's registration with the conference. In our implementation, once engaged the user registration agent behaves as a passive server agent.

Relationship Analysis Agents Like brokers, *RA* agents have an *identification* and a *specialisation*. In addition, they also have a *location*, and perform relationship analysis only for agents which are in the same location. *RA* agents are autonomous agents, and their motivations dictate the types of goals they generate in response to the identification of different types of relationships and the relevant regulations.

Information Services Information services are described by the services they offer, such as *Accommodation Service* or *Food Service*. Information services may vary, according to

[3] Agents are simply expected to adhere to any regulations of which they are notified. The enforcement of regulations is beyond the scope of this demonstration application.

their individual capabilities, from passive agents to autonomous agents.

Physical Devices The attributes used to describe physical devices depend on the physical device itself. For example, if the device in question is a printer, then we may use attributes such as *laser* or *inkjet*, *pages per minute*, and so forth. Physical devices are regarded as passive server agents, since they only react to commands provided to them through the physical device agent.

Physical Device Agent Physical device agents are described by the profile of the device they represent. They are passive server agents if they simply act as a proxies providing access for other agents to the physical device's capabilities. However, if they reason about how other agents are manipulating the device, and intervene when appropriate, they may then act as either active server agents or autonomous agents, depending on the level of intervention and the existence of goal generation capabilities.

Location Agent The location agent simply broadcasts a location identifier so that user agents, *RA* agents and physical device agents can set their own location attributes.

In this section we gave a broad overview of the application and the different types of agents within it. In the next section we discuss the user agent architecture so as to give a clearer idea of the relevant implementation details and how abstract concepts such as attributes find practical implementation.

6.7 The conference user agent

Following the broad overview of the SMART conference environment and the agents performing the supporting infrastructural tasks we now discuss how *act*SMART is used to provide an implementation for the basic user agent architecture. Note that here we only briefly discuss the user agent architecture, with a more extensive description, including a discussion of representing agent attributes using OWL [143], is provided in A.4.

The user agent architecture is illustrated in Figure 6.7. The components that make up the structural specification of the architecture have been grouped according to the overall functionality they cater of. *Sensor* components receive information through either Bluetooth, WiFi, or the owner interacting through the screen of the device. Information and reasoning about the owner

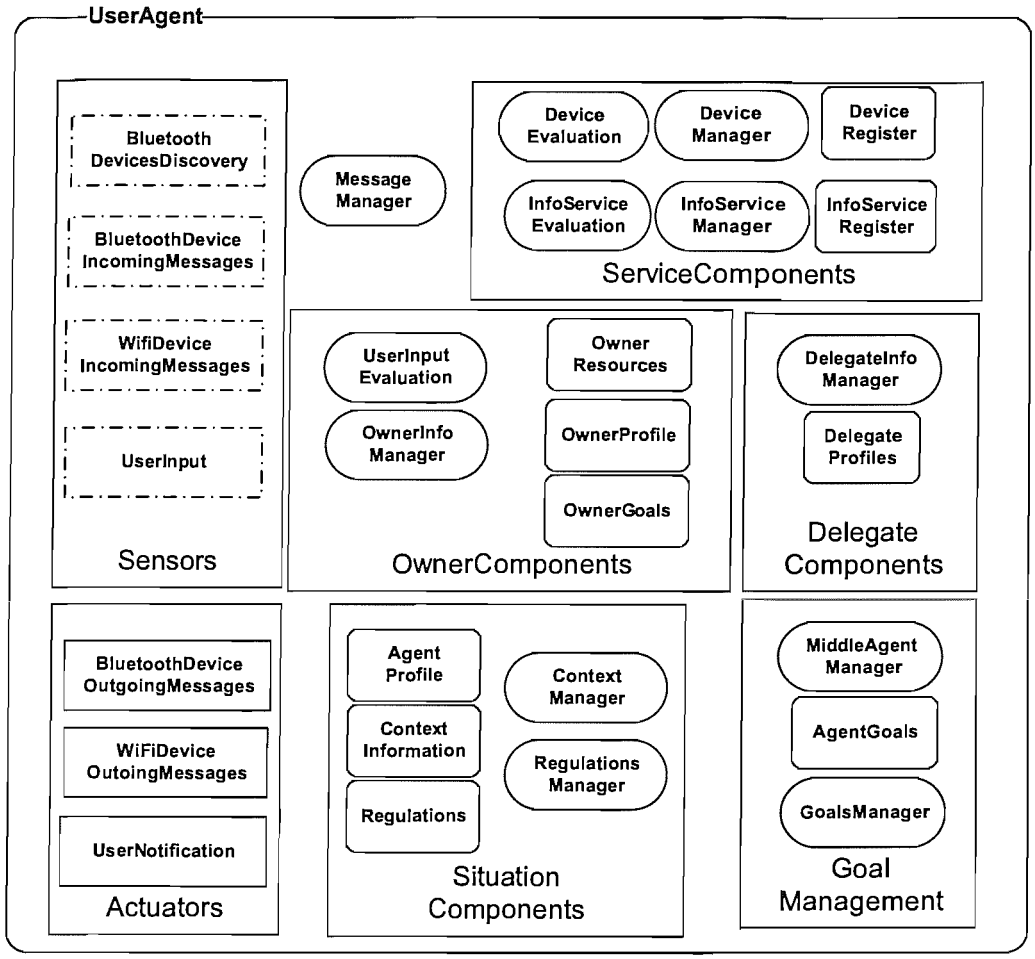

FIGURE 6.7: User Agent Architecture

and the owner's goals is handled by the *Owner Components*, while the *Delegate Components* handle reasoning and information about other delegates and the conference. The *Situation Components* handle information relating to the current context of the agent user, the agent profile, and the regulations that apply to the agent. *ServiceComponents* handle interaction with services, both information services and devices services. *Actuators* manage outgoing information, and essentially mirror the sensors. Finally, the *MessageManager* component handles the routing of incoming messages to appropriate components within the architecture.

6.7.1 Managing the user agent architecture

In order to illustrate some of the benefits of developing and managing the architecture using *act*SMART, we provide here some concrete examples of how we can manage the agent architecture to achieve better results.

Adapting to changes in the environment

Mobile devices are inevitably limited in their capabilities. However, ubiquitous environments present a constantly changing set of devices and services to interact with, as well as different modes of interaction. For example, a device may be able to communicate with other devices through a variety of low-level protocols such as 802.11b, Bluetooth as well as higher level agent language communication protocols. By using dedicated sensor components to deal with different types of interaction we can change the methods used at run-time based on device capabilities. For example, on initialisation of an agent, the agent shell can determine if its host device supports Bluetooth communication and accordingly activate and link the Bluetooth-enabled sensor component. Similarly, an agent shell can determine that a certain protocol, although supported by the host device, is not supported by anything else in the environment and, in consequence, the relevant component is unlinked and deactivated by the shell, thus minimising the load the agent places on the host device.

Suspending operation

A particularly useful feature of *act*SMART is the easy access it provides to the state of individual components and the agent as a whole. This, when combined with the ability to store that information to the persistent record store of devices [107], allows us to *suspend* the operation of the agent either through a user command or when the device is interrupted (e.g. by a phone call). This feature can also be used to periodically save data in order to be able to recover operation if the device unexpectedly switches off.

Modifying the architecture

Finally, through the mechanisms provided by Java mobile device technologies, and particularly, *over-the-air* provisioning of applications [107], we can take advantage of the flexibility afforded by *act*SMART to replace existing architectures with modified versions that can support greater functionality. For example, the architecture described above only has support for dealing with delegate profiles, but not their resources. The component-based nature of the architecture can easily allow us to provide this functionality by performing the following changes.

- Replace *UserInput* and *UserNotification* components to allow the user to define such goals

and retrieve the resulting information.

- Replace the *MessageManager* and *GoalsManager* components to handle the new messages and goals.
- Add a *DelegateResourceManager* controller and a *DelegateResources* infostore to handle the incoming resources.

The remainder of the architecture is identical since the additional functionality does not impact on the behaviour of any other components.

6.8 Using Relationship Analysis Agents

In the previous section we have discussed how *act*SMART can be used to define a relatively complex architecture, and how we can then take advantage of the flexibility afforded by *act*SMART to modify this architecture. We now examine how the use of *RA* agents, as described in Section 6.3 can aid in improving cooperation and coordination within the system. We do this by providing some examples of the types of relationships they can identify in the conference scenario. We recall that throughout the examples we assume that *RA* agents are location-specific agents (i.e. they operate within a specific room) and are able to communicate using both 802.11b and Bluetooth. We also assume that agents are willing to provide information to the *RA* agents, and do so upon entering a new location, but we realise that such assumptions would not be valid in a real environment. However, the development of appropriate mechanisms to provide incentives for agents to be truthful and willing to cooperate goes beyond the remit of this thesis.

Aiding agents to achieve goals

The first example is illustrated in Figure 6.8, in which Agent A is equipped with sensors for communication with both 802.11b-capable devices as well Bluetooth devices, while Agent B can only communicate with Bluetooth devices. This means that the agents share a *Mutually Viewable Environment*, enabling them to communicate, but at the same time A can view other aspects of the environment that B cannot. Now, assume B has the goal of locating an information service that can provide information about local restaurants. In order to achieve this, it must first find a broker able to match the required service profile to existing services. However,

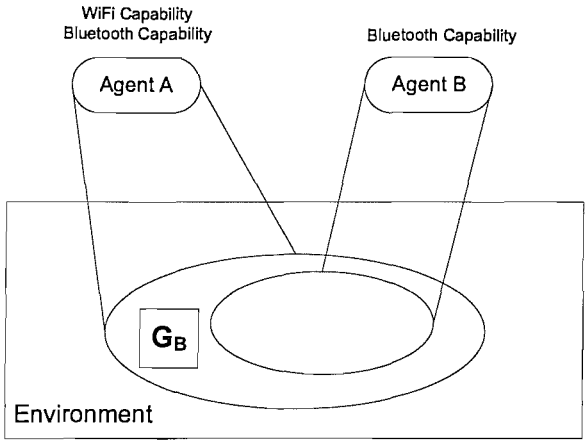

FIGURE 6.8: Aiding agent to achieve query goal

since neither brokers nor the information service are Bluetooth-enabled, B has a query goal that is outside its own *Viewable Environment*. Agent A, on the other hand, is able to communicate with a broker, so the relationship analysis agent can request that A adopts B's goals and performs the required actions for it. Identifying such a situation, the relationship analysis agent can generate an optional regulation, with the goals of locating a broker, querying it about available information services and providing the results to B.

Controlling access to devices

In a room in which presentations take place, we would like that any device the presenter is using at the time (projector, printer, laptop) of the presentation is not used by anyone else at that location, so as to ensure that the presentation is not interrupted in any way. The *RA* agent must therefore identify all agents that have a *MutualInfluence* relationship with the presenter's agent and create a regulation preventing access to any devices within that *Region of Influence*. We point out that the generality of this rule means that we do not need to define the exact devices in question, since this may change from location to location.

6.9 Agent Construction

The discussion in this chapter of infrastructure agents and the specific application scenario has not dealt with the low-level implementation details in any real depth. However, the architectures described have been developed using an implementation of *act*SMART in Java, and here we describe some of the implementation's more salient features. The toolkit consists of a set of

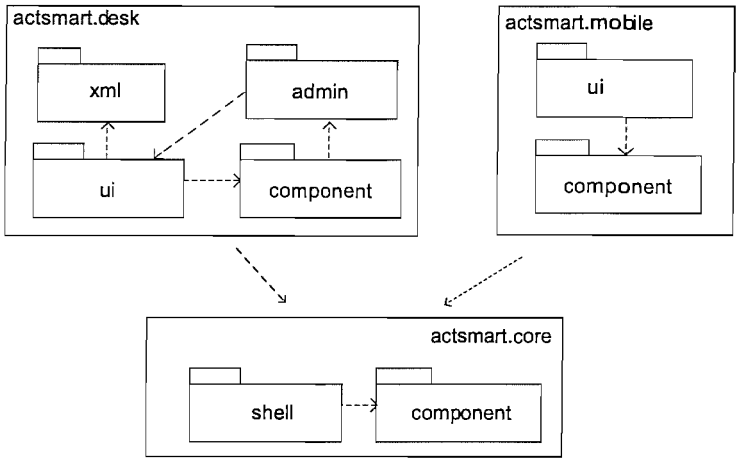

FIGURE 6.9: *act*SMART implementation

applications programming interfaces (APIs), that provide access to the basic code required for defining a shell, components and links between components. These APIs are separated into three different packages, as shown in Figure 6.9, and the `actsmart.core` packages are described below.

Shell The `shell` package contains the main `Shell` class, which implements the functionality for adding and removing components, as well as defining the links between components and the execution sequence of components. The other significant classes in this package are the `Link` class and the `ExecutionSequence` class, each defining the required methods for handling links and the execution sequence.

Component The main class within the `component` package is the `Component` class, which is the base class that all components must extent to implement their specific functionality. It defines the methods that each component should implement, so that they can all be manipulated in the same way by the shell. In addition, it provides some functionality for storing, providing and accepting statements when called to do so. Finally, the `Statement` class defines methods through which statements can be manipulated and extensions for the three different types of statements that we support.

This core set of APIs has been programmed using only classes supported within the Mobile Information Device Profile (MIDP) of the Java 2 Micro Edition [107]. Thus, the implementation of *act*SMART and, by consequence, the way developers access and make use of its basic concepts, remains the same, irrespective of whether development is targeted towards workstations or limited-capability mobile phones. This is in line with our aim of providing both a clear path

from abstract concepts to implementation, and a consistent set of ideas to support development for a variety of agent architectures and operating environments.

This `actsmart.core` package is then extended with more specialised implementations for the devices at hand. In our case, we provide two broad extensions, as illustrated in Figure 6.9, for workstations (`actsmart.desk`) and for mobile devices (`actsmart.mobile`). The `actsmart.desk` extension is aimed for use on typical workstations that can also take advantage of the more extensive capabilities afforded by the Java 2 Standard Edition. The functionality we added through extension is mainly geared towards enabling the debugging of architectures and their manipulation through a basic graphical environment (discussed in Section 6.9.1). The `actsmart.mobile` extension provides some rudimentary debugging capabilities and a set of components that enable agents to perform basic tasks such as making use of data storage facilities on a mobile device. In the next section, we describe in some more detail the `actsmart.desk` extension.

6.9.1 *act*SMART Development Environment

The extension of the core *act*SMART implementation for desktop computers and, in general, more powerful devices, takes advantage of the extensive functionality provided by the Java 2 Standard Edition APIs, as opposed to the limited capabilities of J2ME. This additional functionality is, of course, particularly useful for developing powerful and generic components that handle issues cutting across different application domains such as communication or planning. However, our main focus has been the extension of the core *act*SMART APIs to provide features that are useful in the *development* phase of agent construction by allowing easier access to the underlying features of *act*SMART. These extensions can be accessed either directly through the APIs or through a basic graphical user interface development environment. A description of the features we have built on top of the core APIs follows.

- Components can be loaded into the shell at run-time and the links between components can be defined dynamically. This allows developers to easily test different configurations of architectures in order to identify the best suited for the application at hand.

- The developer can instantiate just parts of the architecture, which enables them to focus on the interactions between a small number of specific components.

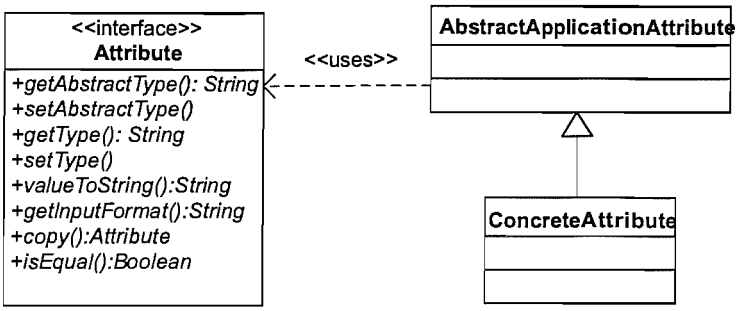

FIGURE 6.10: Attribute implementation

- Developers can *step through* the execution sequence of an agent, monitoring at each point the exchanges of statements between components. This allows developers to identify where problems within the architecture may occur.

- The development environment can also produce an XML file detailing the components that make up the architecture as well as the links between components and the execution sequence of the components. Furthermore, such descriptions can then be loaded back into the development environment to quickly instantiate architectures. This enables developers to save and catalogue different types of configurations for reuse.

These basic features are particularly useful for debugging an agent architecture, and illustrate what is possible when a *principled* approach to agent development is followed, which enables us to *generalise* the manner in which architectures are constructed and debugged.

In order to provide some more details on the implementation of the system, we provide below a description of the implementation of the main concepts of attributes, components, statements, links, and the shell.

6.9.2 Attributes

The implementation of attributes required a structure that is specific enough to actually be of use while maintaining the required flexibility that would allow it to be used in any type of situation. This concern leads to the specification of an *Attribute* interface that defines the methods necessary for manipulating attributes with minimal knowledge of their specific implementation in the context of an application domain.

The suggested way of using the *Attribute* interface is shown in the UML diagram of Figure

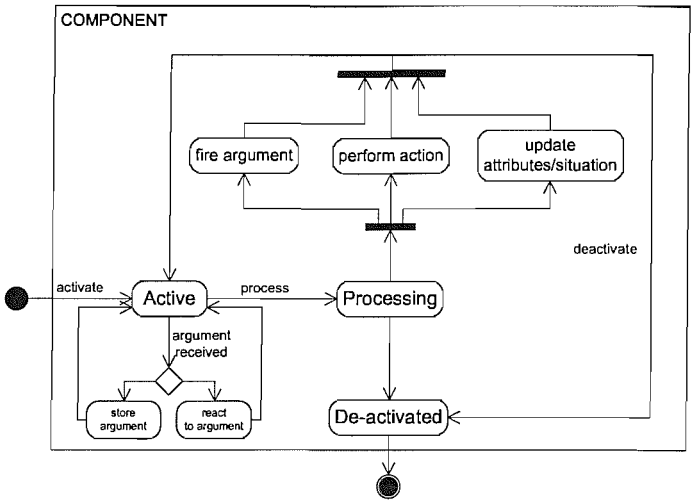

FIGURE 6.11: Component Activity

6.10. The interface can be implemented at either the architecture or application level (or both) and then extended with specific types of attributes.

In *act*SMART, attributes have an `AbstractType` and a `Type`. The abstract type refers to the general type of attribute (e.g., a Location or a Profile) while the type refers to the specific instantiation of an abstract type (e.g., CurrentLocation or Mike_Profile). The actual information stored within attributes is implementation-specific and may depend on arbitrarily complex data structures. Therefore, for them to be manipulated and changed, knowledge is required about the data structures that represent them. Nevertheless, some standard functions can be performed without such knowledge. Thus, the `Attribute` interface stipulates that the following functionalities should be provided by an implementation: testing attributes for equality through the `isEqual()` method[4]; copying attributes through a `copy()` method[5]; and, supporting presentation through the `valueToString()` method that converts the value of an attribute to a textual representation.

6.9.3 Components

As with attributes, the essential methods that components should implement are described in a *Component* interface. However, in addition to an interface definition, we also provide a skeleton

[4]Java provides an `equals()` method which all objects inherit, but for our purposes the semantics of equality are usually more demanding. Two attributes are not equal if they simply refer to the same object but if the values in their data structures are equal.

[5]Once more the semantics are different to the `clone()` method Java provides. A copy of an attribute is a copy of all the data referred within that object as well.

implementation of the interface so that developers only need to define the methods carrying the specific functionality of the components. The implemented functionality provides the ability to store and manage incoming and outgoing statements, and to provide access to statements currently being processed by the component.

The generic activity of components is shown in Figure 6.11. The main states of a component are *active*, *processing* and *deactivated.* In the *active* state, the component is simply listening for statements. Once a statement is received, a component can either store it and react to it later, or react to it immediately. The choice between the two types of behaviour depends on the application requirements as discussed throughout the various architecture examples presented in this chapter and in Chapter 4. For example, if tight control of component scheduling is desired, all statements should be stored upon receipt and acted upon in the *processing* state. This state is where the component does the bulk of the work by stepping through statements and deciding how to deal with each one. The choices, which do not exclude each other, are to produce a statement in response, perform an action or, in the case of an infostore, update stored attributes. Finally, the component may be called to enter its *deactivated* state. At this point, the component implementation should perform any operations required to ensure that deactivation is handled gracefully. Here, the component will still receive statements, unless the links with other components have been severed, but it will not act on them nor store them.

Statements stored within components are placed in a typical `Vector` object, which also has the capability of notifying a listener to changes to the vector. This functionality is used to facilitate debugging, since the developer can monitor statements being processed by a component through the graphical interface. Since these statements represent the state of the components, it means that this state can also be stored for transfer to a different entity configuration or can be used to synchronise entities between desktop and mobile environments.

6.9.4 Statements

Statements are implemented in *act*SMART through a generic `Statement` class that is then extended to provide specific types of statements, such as those described in Chapter 4. In order to add more flexibility to the framework, and to allow for its easy extension to include more statement types, statements are created through a `StatementFactory` that can also perform checks on the validity of the requested argument structures. Currently, the `SmartStatement-Factory` supports the creation of statements for INFORM, REQUEST and EXECUTE. Ap-

plications needing more specialised statement types can define their own factories or extend the existing factory.

6.9.5 Links

In our implementation of *act*SMART, a `Link` class is provided for each component that produces statements. This class holds *paths* between the sender and the receiver of a statement. The `Path` class defines a *sender* of a statement, a *receiver* and the statement to be sent.

The `Link` class is where information is stored about where statements produced by a component should be routed. Therefore, this information is completely decoupled from the components themselves, and can be managed by the shell.

6.9.6 Shell

The shell manages components, the links between them and the execution sequence. It can also hold attributes that have an entity-wide scope. Components in a shell are held in a vector structure and each component is accessible via a `componentID` and can be observed through the graphical interface (similarly to statements within components). The shell links components by creating a new `Path` object and placing it within the `Link` object of the statement-producing component. The components are executed in the sequence defined in the `ExecutionSequence` object, which is configured by the developer. As we mentioned in Chapter 4, currently we only support a sequential execution sequence, so the `ExecutionSequence` object implementation is relatively simple. However, an alternative implementation of this class following a more sophisticated control-flow mechanism is possible, since the class is decoupled form the rest of the implementation of *act*SMART.

The object relationships diagram in Figure 6.12 provides an overview of the relationships between the key classes in our implementation of *act*SMART. The `Shell` manages `Link` and `Component` objects and refers to an `ExecutionSequence` object. The `Component` object produces `Statement` objects, which contain `Attribute` objects. A `Link` object refers to a `Component` object and contains `Path` objects. Finally, `Path` objects relate two `Component` objects to a `Statement`.

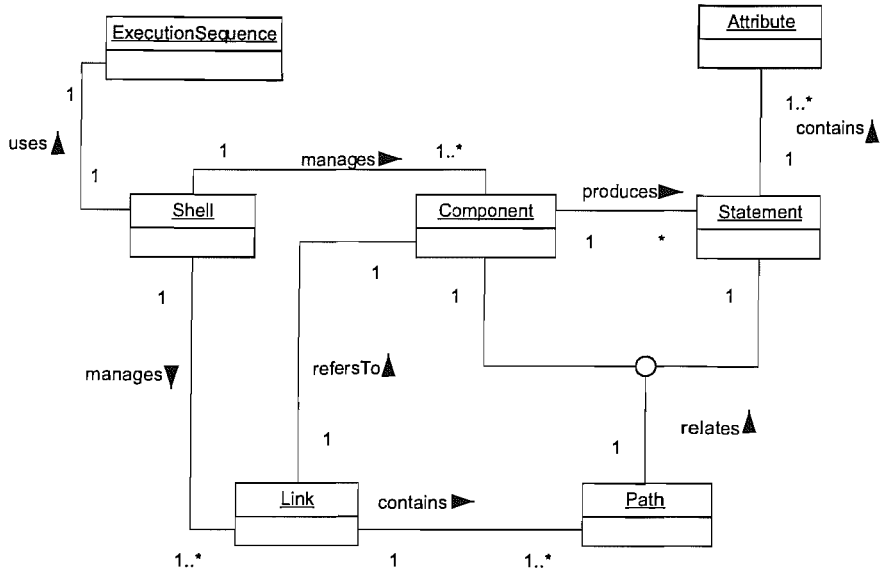

FIGURE 6.12: *act*SMART implementation class overview

6.10 Conclusions

In this chapter we have provided a view of all the models introduced in the thesis working together to support the development of an agent-based application. The main aim is to demonstrate that the models can provide real benefits to application development and that translation from abstract models to practical implementation is possible.

The implementation of several architectures in *act*SMART has provided useful experience as to the suitability of the model for agent construction within a practical application setting. Although the implementation of interactions with other sources was based on a simulation of the environment, the APIs used are those directly supported by the majority of high-end mobile phone devices.

The fine-grained control over every aspect of the agent aids significantly in testing and debugging, since components can be tested individually and, more importantly, they can be tested in connection with other components without requiring an instantiation of the entire architecture. Moreover, the state of each component, and the agent as a whole, is clearly defined, and changes to individual components and to the overall architecture are easy to achieve.

One of the central concerns has been that the model might place too many demands on a device, since a component-based approach is inevitably more *expensive* in processing requirements that more tightly-integrated implementations. Although more work is needed, both analytical and

experimental (or developmental), our tests on PCs and low-end PDAs indicate that the difference is not significant, especially when seen within the context of our ability to adapt architectures to the capabilities of devices.

From the multi-agent point of view, access to a consistent set of concepts has proven useful in enabling us to accurately model the different types of entities in an environment, and even facilitate run-time agent discovery based on such application-independent types. Furthermore, the relationship analysis agent represents a new type of infrastructure agent that can play an effective role in managing dynamic and heterogeneous multi-agent systems.

Compared to existing work on developing appropriate infrastructure for supporting ubiquitous computing (e.g. [190, 52], the two main benefits of our approach has to offer are a principled means for designing and describing agents, and extensive support for analysis of interactions between agents.

Chapter 7

Conclusions

"Education is a progressive discovery of our own ignorance."

Will Durant (1885 - 1981); US historian.

7.1 Introduction

The technological advances in device miniaturisation, increased processing power, and networking capabilities can support increasingly more complex and heterogeneous computing environments, where a range of devices can potentially communicate with and make use of services provided by others. In line with this profile, there is also an increasing demand for *integrating* the various different kinds of such devices in order to provide an environment in which access to information and services is available in a seamless manner, while transcending physical location and computing platform. However, application development for such environments poses two significant challenges. Firstly, developers must deal with a range of operating environments, requiring individual application components to be tailored to the demands and capabilities of individual devices, which inevitably increases the complexity of the design and development process, and makes the need for a consistent approach throughout essential. Secondly, developers must build systems in which disparate components are able to cooperate effectively and cope with changing application needs according to the state of the environment, creating the need for applications that can adapt dynamically at run-time.

Agent-based systems have a key role to play in the effort to provide and support such applications, since agents embody several of the required characteristics for effective and robust

operation in dynamic and heterogenous computing environments. However, there is a number of shortcomings relating to the use of the agent approach to application development. In particular, in this thesis we deal with the lack of clarity in existing agent models and address the need for models that can directly support practical application development. These are widely-accepted shortcomings that have been identified by a number of researchers in recent years [231, 189, 136, 32, 227]. Both issues are central to the effective application development in heterogeneous environments. The lack of clarity of conceptual models hinders the application development process, forcing developers to resort to ad-hoc methods, and constraints the ability of developers to have a consistent view of the entire system so as to better address problems when they arise. In addition, it makes the *reuse* of solutions across different applications harder, since there is no consistent way of describing such solutions. However, a conceptual model can only be useful if there is a clear path from that model to its practical implementation, providing true value for developers, who need to ensure that an abstract specification can be translated to practical, realisable systems.

In direct response to these challenges we have addressed the following specific issues in this thesis.

- Provide abstractions in support of the construction of individual agents, that can be used both during the conceptual elaboration and design of agent systems and during their practical implementation.

- Provide support for cooperation between agents through a model that enables us to firstly identify and subsequently reason about the relationships between agents.

An overarching aim of this is that any work developed should be *resuable* across a wide range of applications to support the transfer of knowledge across domains and reduce the development effort.

In this chapter we provide a summary of this work, highlight the specific contributions we believe we make in this thesis and, subsequently, discuss the limitations of the work along with the possible avenues for further work.

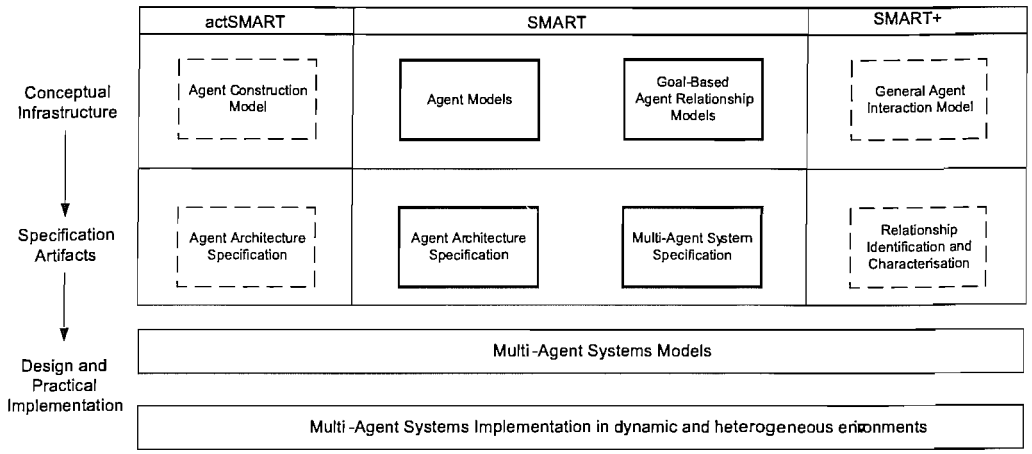

FIGURE 7.1: Overview

7.2 Summary

At the centre of the work presented in this thesis lies SMART, a conceptual framework adopted with the aim of refining and expanding it to achieve our aims. SMART provides us with the basic abstractions required to underpin development in agent systems, enabling us to describe both individual agents, without reference to specific agent architectures, and the relationships between agents.

However, SMART is lacking in two important aspects. Firstly, although the abstractions provided for describing individual agents are well-suited to our aims because they restrict us to specific architectures, they do not explicitly address the entire range of agent types we need to deal with for development and no paths are provided from those abstractions to the practical implementation of agents. Secondly, the abstractions in support of agent relationships are restricted to a particular class of relationships that are centred around agents attempting to achieve a common goal. Although such relationships play an important role in multi-agent systems, we also need to support the representation and reasoning about a wider range of relationships.

In response to these shortcomings, we have shaped the work in the thesis around a clearly defined plan of expanding and refining SMART, in order to align it with our aims. An overview of the extensions developed in this thesis is illustrated in Figure 7.1 (reproduced from Chapter 3). On the one hand, the agent construction model (*act*SMART) enables us to define specifications of architectures that can be directly implemented, and provides the required links between the abstract agent model and its practical implementation. This is also supported by the *translation* of *act*SMART into a set of APIs that can be used to support development. Throughout the thesis we

developed several examples of the use of *act*SMART to define a variety of architectures, ranging from a basic agent for auctions to agents performing negotiation and argumentation, infrastructure agents, and agents supporting users within a ubiquitous computing application scenario. On the other hand, SMART+ refines and extends the existing SMART models for agent relationships by introducing a more generalised model of interactions. This general model enables us to identify and characterise different types of relationships, both at design-time and at run-time, and we provided a generic typology of relationships so as to facilitate this characterisation. In addition, the interaction model also enables the characterisation of an agent's goals, allowing us to consider goals in relation to an agent's capabilities and other agents in the environment. We have illustrated the use of SMART+ through the definition of *interfering relationships* and the specification of an agent dedicated to relationship analysis in the context of a ubiquitous application scenario.

Underpinning this work was a careful consideration of the notion of agents as defined by SMART and a refinement to more closely describe the types of entities we are likely to encounter within a realistic application scenario. Below, we outline these refinements, before going on to discuss *act*SMART and SMART+.

7.2.1 Refining the Abstract Agent Model

For developers to be able to adopt an agent-oriented paradigm, there must be an unambiguous understanding of what constitutes an agent and, especially within the context of heterogenous application environments, an ability to *differentiate between* and *relate* types of agents. Although SMART offers clear definitions, it does not provide the level of granularity required to accommodate the different types of agents that may be encountered during application development.

We refined SMART's basic notions, and reconciled them with Wooldridge and Jennings's widely used *agent characteristics* [232], gaining the required level of granularity through the introduction of the notion of *self-direction* as distinct from the notion of autonomy. Self-direction is the ability to choose what actions to perform in order to achieve a given goal, while autonomy is the ability of an agent to generate its own goals. Agents that exhibit self-direction *actively* attempt to achieve goals, while agents that do not are *passive* agents, since their actions are entirely defined by the way they are manipulated.

7.2.2 *act*SMART: Agent Construction Model

The basic abstractions relating to individual agents, coming from SMART and our refinement of SMART, outlined above, support the definition of the agent construction model that allows us to develop specifications of agent architectures. The agent construction model (*act*SMART) was developed with the aim of addressing the need to construct agents for heterogeneous environments, where it is not realistic to assume that all agents will use the same type of architecture. Rather, the conceptual infrastructure should allow developers to create the most suitable architecture for the task at hand while providing consistency across architectures through a common set of underlying concepts. In addition, *act*SMART also enables architectures to adapt to changing needs, in line with the varying demands that heterogeneous and dynamic environments place on applications.

*act*SMART takes a component-based approach to agent development. *Components* represent different types of functionality within the agent architecture (sensing, acting, information storage, and decision-making) and are placed within a *shell* that manages the communication between them, and the sequence in which components execute. This approach enables us to distinguish between the *descriptive specification* of an agent using the SMART concepts of attributes, capabilities, goals and motivations, the *structural specification* of the agent as expressed through the different types of components that comprise the agent architecture, and the *behavioural specification* of the agent as defined by the ways in which the components interact.

In addition to enabling the comparison of agent architectures from different viewpoints, these distinctions allow us to reconfigure the agent architecture through the shell. Since components are independent of each other, we can change components and the way an agent architecture executes through the shell at run-time.

The use of *act*SMART is illustrated in Chapter 4 through several examples while some actual implementations of architectures are discussed in Chapter 6.

7.2.3 SMART+: Relationship Identification and Characterisation

Just as the agent construction model underpins the specification and development of individual agents, the *interaction model* underpins the development of multi-agent systems. Starting from the premise that in dynamic and heterogeneous systems we can never be sure that the only

relationships that are instantiated are those explicitly considered at design time, we have focused on developing a systematic means for *identifying* relationships, both at design-time and at run-time, and characterising those relationships in order to facilitate the choice of *relationship management* techniques.

The model of interaction of an individual agent with the environment only makes use of knowledge of an agent's actuator and sensor capabilities. This allows us to apply the model to a wide range of agents, since we make no assumption about their internal operation. The interaction model leads to the definition of *regions* of the environment that an agent is able to view (*Viewable Environment*) or affect (*Region of Influence*).

Using this interaction model we can investigate all possible relationships between two agents by examining how their individual *Regions of Influence* and *Viewable Environments* overlap. More specifically, these overlaps enable the creation of a typology of agent relationships to identify when two agents are able to view the same regions of the environment (*mutually viewable environment*), each other actions (*observable actions*), and when they are able to change the same regions of the environment (*mutual influence*).

Knowledge of all possible relationships between two agents is further enhanced by including in the model knowledge of an agent's goals. It is the agent's goals that ultimately determine which of all the possible relationships will be instantiated, enabling us to expand or restrict the possible relationships between agents, since we have an indication of the exact regions of the environment that agents can seek to influence. Thus, we have developed a typology of agent goals, relating goals to the *Region of Influence* and *Viewable Environment* of an agent.

These relationship analysis tools are illustrated through the definitions of particular types of relationships, such as *interfering relationships* and the development of a dedicated relationship analysis agent.

7.2.4 Implementation and Evaluation

The models for agent construction and relationship analysis were evaluated through a simulation of a ubiquitous computing application. The application includes middle agents for capability matching, and agents that are able to perform relationship analysis using the models developed in Chapter 5. Architectures for both agents have been specified using *act*SMART. We have provided examples of how relationship analysis agents can be used to identify relationships

between devices at particular locations and consequently generate rules to regulate interactions or inform agents of the possibilities for cooperation.

The evaluation is supported by the implementation of *act*SMART, which acted as the core of a desktop-based development tool that enables developers to modify the agent architecture by dynamically loading components and changing the execution sequence of the components. This also allows us to test specific aspects of the architecture, by instantiating just those components that we wished to check and also check the operation of the architecture by stepping-through the execution sequence.

7.3 Contributions

In this section we identify specific contributions that we have made through the work developed in this thesis. Several of these contributions have been described in a number of publications that have been presented in international workshops and conferences [7, 8, 9, 10, 11, 12].

7.3.1 Abstract Agent Model

We have refined SMART's model of agents which, although providing definitions for different types of agents, does not have the required level of granularity nor support for translating those definitions into structural and behavioural models of agents. We introduced the notion of *self-direction*, as distinct from the notion of *autonomy*. At the risk of repetition, self-direction refers to an agent's ability to choose how to achieve a goal, while autonomy refers to an agent's ability to choose a goal. In addition, through *act*SMART we provide a clear path from the specification of different entity types at an abstract level to the way such entities can be constructed at a practical implementation level.

This enables developers to proceed with system design with a clear understanding of what the concepts used imply for both design and implementation. Such clarity supports the reuse of solutions across domains and applications, which can eventually lead to reduced development costs.

7.3.2 Agent Construction Model

We have developed a conceptually grounded and architecturally-neutral model of agent construction, that enables the specification and development of modular and reconfigurable agents. The model is conceptually grounded through the abstract model of agents, discussed above, and is architecturally neutral since it does not restrict agents to any single architecture.

In order for agents to operate effectively in heterogeneous environments, their architectures must be tailored both to the demands of the application and to the demands and limitations the environment places on the application. This means that developers must deal with a number of different agent architectures for a single application, increasing the complexity of application design. Our agent construction model provides a consistent manner in which to specify and construct a range of architectures so as to reduce development effort, avoid the use of ad-hoc development methods, and enable reuse of solutions across applications. Furthermore, *act*SMART is equally relevant to development for both limited-capability devices as well as more powerful ones, providing the necessary consistency across the application domain.

Finally, *act*SMART provides the following secondary contributions.

Shell, component types, links and execution sequence The agent construction model makes use of: a shell as a *manager* of agent components; distinct components types as a means of encapsulating different types of agent functionality (reasoning, sensing, acting, information storage); links between components to support information-flow; and an execution sequence to define the order of execution of components. Each of these issues can be considered as an engineering construct to be reused in different contexts, supporting an agent construction model that can be based on several different abstract agent models, not just SMART. They are constructs that enable modular and reconfigurable agent architectures, regardless of the underlying agent model supporting construction.

Graphical Notation We have developed a graphical notation for describing agent architectures that enables us to illustrate the different types of components within the architecture and the information-flow between components.

7.3.3 Description, Structure and Behaviour

In support of the agent construction model we have provided a three-dimensional view of an agent, distinguishing between the *descriptive specification* of agents, in terms of their attributes, capabilities, goals and motivations, and the *structural* and *behavioural* specifications, providing benefits both at the design and at the implementation stage of agent architecture development.

At the design stage, it provides significant flexibility since it provides a developer different perspectives and the ability to move between them while refining the design of an agent architecture. We have illustrated this process through examples, such as the architectures for negotiating agents in Chapter 4. There we show how to move from a detailed descriptive specification which, in essence, provides a set of requirements for the agent, to more detailed structural and behavioural specifications which indicate how those requirements can be met and specified through *act*SMART.

At the implementation stage, it enables us to experiment with different structural and behavioural specifications by manipulating components and the links between them, as well as the execution sequence.

7.3.4 Linking Theory to Practice

The development of *act*SMART also makes an important contribution from a purely research-level perspective since it provides an example of the clear path from the abstract specification of agents in SMART to the elaboration of that specification to construct agents. We have demonstrated how a well-established theoretical model can be made more relevant to application development, while still providing access to the concepts of the abstract framework. Well-understood software engineering concepts such as components, and the refinement of components into component types, have shown how to use these concepts at the design stage to provide specifications for agent architectures without concern for the specific implementations of computational mechanisms to achieve the functionality of components.

7.3.5 Model of agent interaction

We have developed a *model of agent interaction* that is widely applicable, since it makes minimal assumptions about the internal structure of agents, focusing instead on the abilities of agents to

affect change or retrieve attributes from the environment. The interaction model makes use of SMART, and can be directly translated to a practical tool that agents can use at run-time, since we already have a clear path between the concepts of SMART and their practical implementation through *act*SMART.

The interaction model is a key contribution towards supporting cooperation between agents, since it enables us to model the possible interactions between agents in order to gain a better picture of the system and arrive at appropriate decisions about the best models to support cooperation. Such an analysis is important at design time, by revealing issues not considered explicitly, and also at run-time since, in heterogeneous and dynamic environments, agents may enter and leave the environment at any time.

In developing the model of agent interaction we also make the following secondary contribution.

Viewable Environment and Region of Influence We have introduced and formally defined the notion of *Viewable Environment* as the region of the environment an agent is able to view, and the notion of *Region of Influence* as the region of the environment as agent is able to affect.

7.3.6 Typology of Relationships

Using the model of agent interaction we have comprehensively characterised all the possible interactions between two agents through a typology of relationships, which allows us to take decisions about how to deal with different types of situations. It is especially useful for *automating* the reasoning about relationships, since the typology can be used directly by agents or by systems management tools at run-time to facilitate the application of relationship management based on the identified types of relationships.

7.3.7 Typology of Goals

The relationship typology is also supplemented with a typology of goals, which relates goals to their location within an agent's *Viewable Environment* and *Region of Influence*. Knowledge of an agent's goals enables us to expand or constrain the possible relationships between agents, since it indicates the exact regions of the environment that an agent will either attempt to sense or affect.

7.4 Limits and Further Work

The work presented in this thesis represents a significant step towards providing truly *reusable* models in support of agent-based systems development, both with respect to the construction of individual agents, and with respect to supporting cooperation in multi-agent systems. In fact, it is precisely this *comprehensive* aspect of the work that is key to its utility. The same set of concepts is used throughout, providing the necessary consistency in development that can ensure both reusability between applications and the ability to contrast alternative solutions, and facilitating progress towards robust application development.

From a research perspective the work is one of the few examples that adopts an *existing* conceptual framework and refines and extends it, providing a clear path from abstraction to construction. As such the work represents a clear progression from the current state-of-the-art.

Inevitably, there are limitations, relating both to the inherent difficulty of evaluation and to the multiple facets of agent development, ranging from methodologies to development toolkits. More importantly, the work provides promising avenues for further research that seem able to lead to further useful results. We discuss both limitations and possibilities for further work below.

7.4.1 Limits

Lack of methodology If our overarching aim is to support the development of agent-based applications, then perhaps the most serious limitation of this work is that it is not coupled to a principled *development methodology*. A methodology describes the steps developers should take to move from the definition of a problem to the specification and implementation of an agent-based application addressing the problem. Nevertheless, we recognise that a prerequisite to a methodology is a principled account of the models that form the space of discourse for the methodology. The work in this thesis provides such models and thus creates the necessary preconditions for developing a methodology.

Evaluation across domains Throughout the thesis we have provided several examples of the application of the models. However, more examples and more extensive evaluation across different domains would undoubtedly strengthen the arguments for the validity of the work. Unfortunately, the limited resources of development within the context of the thesis have meant that we attempted to focus effort on several key examples as best as possible.

Scalability of relationships models The relationship models presented provide a means to analyse relationships between agents and use that information to manage agent-based systems. However, we have not dealt with the implications of dealing with thousands or even hundreds of thousands of agents. In the context of the design for the relationship analysis agent presented in Chapter 6, the problems faced are similar to those faced by middle agents, where some results on scalability are available [129], but a closer analysis of the particular issues concerning relationship analysis and scalability are necessary.

Trust and security Application development in the context of heterogeneous and dynamic environments inevitably raises the need to ensure that any attempt to act maliciously is effectively controlled. Within the context of the work presented here such issues are particularly relevant, since when dealing with relationship analysis agents are required to share information about their capabilities and decisions are taken based on that information. An account of the implications of doing so within environments in which agents may be willing to deceive would enhance this work.

7.4.2 Further Work

Necessarily, the limitations discussed above must also act as pointers for further work. However, beyond work directed to addressing such limitations, further work is possible to extend and expand the relevance of the work both at the level of abstract models as well as at a practical implementation level.

Systematic comparison and evaluation techniques It would be useful to take advantage of the ability to describe, through SMART, *act*SMART and SMART+, a range of agent architectures and interactions across different domains in order to systematically analyse and compare different approaches. From a research perspective this can provide a better means for identifying progress, while from a development perspective it can provide guidelines in the form of construction patterns for individual agent architectures and multi-agent systems.

Analysis of multi-agent systems The relationship analysis and identification tools can allow us to analyse an entire multi-agent system, identifying the *level of potential cooperation or interference* between agents according to the types of relationships that are prevalent within the system. This could enable us to characterise different types of agent societies,

or identify particular agents within a system that are heavily relied upon or which are particularly damaging to the society as a whole, akin to the concepts discussed in the contexts of social dependence networks, but without the reliance on knowledge of plans, as discussed in Chapter 5.

Application of relationship analysis tools across domains The agent interaction model and the resulting relationship typology can find application in a number of areas. For example, within a market domain, relationship analysis can be used by agents to analyse the relationships between other agents, identifying situations that may indicate that two agents are either competing, collaborating or colluding. Using such information we can then make inferences about the trustworthiness of different agents. For example, if two agents are related by virtue of the fact that they are selling in the same market (common *Region of Influence*) and they belong in the same organisation, we could assume that they share their *Viewable Environment* as well, since they belong to the same organisation, and may attempt to collude to enhance their standing within the market. This line of research has been taken up by Sabater and Ramchurn, building on existing work on trust and reputation [174, 188].

Supporting service composition A topic of particular relevance in recent years has been the use of technologies to provide semantically-annotated descriptions of services to support service composition [205]. The combination of our relationship analysis techniques with such semantically-annotated services could provide an important tool to support development in this direction, since it could indicate some of the effects of composing different types of services offered by agents.

Integration with existing agent development tools The integration of the agent construction model within existing agent development tools, such as JADE [21], could prove a relatively cost-effective means of providing direct access to the models presented in this thesis in the context of a wider infrastructure supporting the development of multi-agent systems.

7.5 Conclusions

Any advances in agent research must be done with the recognition that existing work has reached a certain level of maturity, and there is a wealth of alternative proposals available. By basing

our work on an established existing model and clearly identifying its limitations and needs for refinement in order to better deal with dynamic, heterogeneous computing environments we hope to have ensured the relevance of the work to the state-of-the-art.

Agent-based development has an important role to play in shaping the way in which applications for distributed, heterogeneous environments are, and will be, developed and managed. However, for the paradigm to find wide application and become as mainstream as object-oriented development, we must ensure that our abstractions are presented in a clear manner, provide real utility to developers, and are related to practical implementation issues. In this thesis we have done just that, through the development of principled, reusable models for agent construction, in support of multi-agent systems.

Appendix A

Agent Architectures for the Demonstration Application

A.1 Supporting Infrastructure for Individual Agents

In order to gain a better understanding of the operation of agents within the demonstration application of Chapter 6, we describe here, in broad terms, the technical infrastructure in support of individual agents.

Irrespective of its specific functionality within an application, every agent is assumed to operate within the overall technological framework illustrated in Figure A.1. At the lowest level, an agent is considered to function through the support of a specific operating system that provides

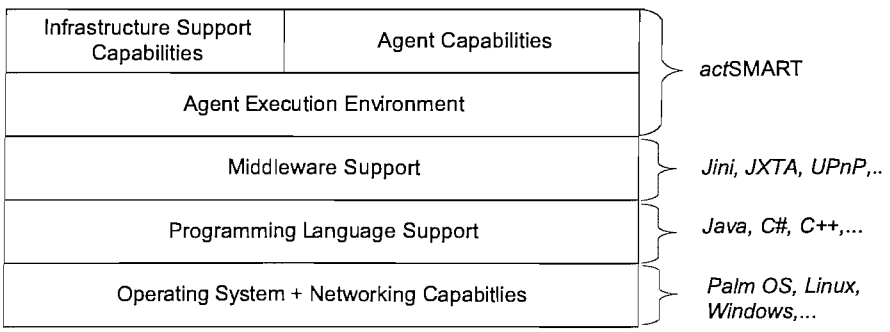

FIGURE A.1: Technological framework for conference agents.

low-level access to the network capabilities of the device hosting the agent, processing power, memory, and so forth. Agents are developed in a specific programming language, according to the support available on the device. In our case, this is always Java, since we take advantage of Java 2 Micro Edition, which is supported by the widest range of mobile devices. In addition, agents may benefit from access to specific middleware technologies such as Jini [5] or JXTA [156] to facilitate the discovery of other services and the exchange of messages. As mentioned earlier, we abstract out such details, and assume very basic middleware support for the exchange of messages. [1] The agents are contained within an agent execution environment, which provides some administration capabilities that are not dealt with directly by the agent architecture, such as logging of actions and administration of the agent lifecycle. Finally, the agent capabilities are the ones that make up the actual agent architecture. In our case the agent execution environment, the infrastructure support capabilities and the agent capabilities are all modeled through *act*SMART and are supported by our implementation of *act*SMART in Java. Infrastructure support capabilities are mainly those that enable the agent to interact with other agents in the environment through the use of specific communication protocols such as Bluetooth or 802.11b. Once more we point out that the application is simulated so the existence of different protocols is simply represented by different types of sensors or actuators representing different modes of interacting with other agents.

A.2 Broker Agent

A.2.1 Structural Specification

In the structural specification we provide a description of the components that make up the agent architecture, without referring to how the components interact together to provide the required behaviour for the agent.

The broker architecture has two sensors, which accept different types of messages from agents. The *ServiceRegistration* sensor is able to accept messages from agents wishing to register wanted or provider profiles. It produces INFORM statements that contain the profile to be registered. The *IncomingQueries* sensor is able to accept messages relating to queries about existing provider profiles. It produces appropriate INFORM statements with this information.

[1] The reader interested in the specifics of appying the SMART model in the context of middleware such as Jini can refer to our previous work on the subject [6, 8].

The infostores for the architecture simply maintain wanted and provider profiles. The *ProvidedServices* infostore contains all the provider profiles currently registered with the broker. It can reply to REQUEST statements for specific types of provider profile and accept INFORM statements to update the list of provider profiles. The *WantedServices* infostore contains all the wanted profiles currently registered with the broker. It can reply to REQUEST statements for specific types of wanted profiles and accept INFORM statements to update the list of wanted profiles.

The controllers manage the registration of profiles with the agent and match wanted to provider profiles. The *RegistrationManager* controller handles the maintenance and updating of registered profiles. It contains the necessary logic for parsing agent profiles to determine their validity, as well as ensuring that agents are notified when their profiles can no longer remain registered with the broker. It can produce REQUEST messages for provider or wanted profiles; accept INFORM statements with profiles to register; and produce EXECUTE statements for messages to be sent to registered agents. The *QueryManager* controller handles queries from agents. It contains the matching algorithm that is used to match wanted profiles against provider profiles. A profile is matched against a query if every attribute, capability, goal, and motivation in the profile submitted by the querying agent is found in a registered profile. The first profile that matches is returned. This is of course a very basic matching algorithm but sufficient for the purposes of our demonstration application. The component can produce REQUEST messages for provider or wanted profiles; accept INFORM statements with queries; and produce EXECUTE statements with replies to queries.

Finally, the actuators send messages to agents, informing them of either a match to their query or of the expiration of their registration. The *RegistrationMessages* actuator is used to send out messages relating to the administration of registrations of profiles by the broker. For example, an agent may be warned that its registration has expired and will be removed. It can accept EXECUTE statements with the message it should send to agents. The *QueryReplies* actuator is used to send out messages relating to replies to queries from agents. It can accept EXECUTE statements with the replies it should send to querying agents.

A.2.2 Behavioural Specification

The overall architecture of the broker is shown in Figure A.2. Messages for registering services arrive at the *ServicesRegistration* sensor, where they are parsed for syntactic validity and passed

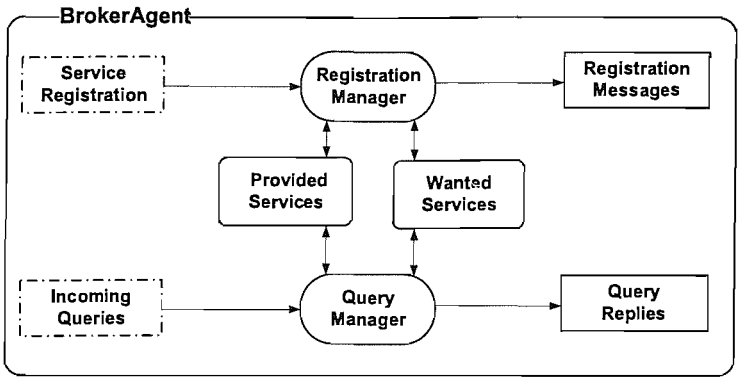

FIGURE A.2: Broker architecture

on to the *RegistrationManager*. The *RegistrationManager* queries the *ProvidedServices* infostore to ensure that registrations are not duplicated, and subsequently updates it appropriately. Messages relating to queries from other agents about registered services arrive at the *IncomingQueries* sensor. They are then passed on to the *QueryManager* which, according to the query, attempts to match the wanted profile with the registered profiles. If no match is found, the request is stored in the *WantedServices*, which the *QueryManager* periodically checks against provider profiles to determine whether a match is available. If a match is found, a reply is sent to the requesting agent via the *QueryReplies* actuator. Both wanted and provider profiles are removed from the infostore once their registration time period expires.

The execution sequence for this architecture combines both an event-driven and a sequential model. When a service registration or a query for a service arrives, it generates an event that causes a cycle of execution to commence. At the same time, the shell periodically causes first the *RegistrationManager* controller to execute so as to check that registrations have not expired, and second the *QueryManager* to check new provided services that may satisfy wanted services.

A.3 Relationship Analysis Agents

A.3.1 Structural Specification

The sensors of the *RA* agent simply allow it register agents and accept profiles or updates to profiles. The *AgentRegistration* sensor accepts agent profiles from agents registering with the

RA agent. It produces INFORM statements with the received profiles. The *UpdateInformation* sensor accepts profiles containing just the current attributes and goals of an agent. It produces INFORM statements with this information.

The agent's infostores store the various types of information relating to information about the agents being analysed, domain information and regulation information. The *RegionStore* infostore maintains information about an agent's *Viewable Environment* and *Region of Influence*. It can reply to REQUEST statements to provide information, and accept INFORM statements to update information. The *GoalStore* infostore maintains information about the goals of an agent categorised along the lines of the goal typology. It can reply to REQUEST statements to provide goal information about a specific agent, and it can accept INFORM statements to update the information stored. The *DomainInformation* store maintains information that maps capabilities to the attributes they can influence or view in an environment. It can reply to REQUEST statements to provide information about a specific capability. Domain information is supplied at design-time, although more dynamic ways of updating and maintaining such information are possible.

The controllers deal with relationship identification and analysis. The *RegionIdentification* component can analyse an agent profile in order to identify the set of attributes that form an agent's *Viewable Environment* and *Region of Influence*. It can accept INFORM statements with an agent profile and produce REQUEST statements for domain information relating to capabilities. It can produce INFORM statements with the sets of attributes that define the *Viewable Environment* and *Region of Influence* for an agent, along with an identification of that agent. The *GoalCategorisation* controller can analyse an agent profile containing goals of an agent and make use of information about that agent's *Region of Influence* and *Viewable Environment* in order to categorise goals according to the goal typology of Chapter 5. It can accept INFORM statements with an agent profile containing information about current goals and attributes; produce REQUEST statements requesting information about the agent's *Region of Influence* and *Viewable Environment*; and produce INFORM statements with the goal categories and the identification of the agent in question. The *RelationshipAnalyser* compares the *Viewable Environment* and *Region of Influence* of an agent against those of other agents and produces the relationship table. It can produce REQUEST statements to request information about an agent's regions and produce INFORM statements containing the relationship table. The *ConflictAnalyser* controller can identify conflicts between agents and accordingly identify whether any regulations apply to the situation. It makes use of the relationship table for the agent under analysis and any relevant goal infor-

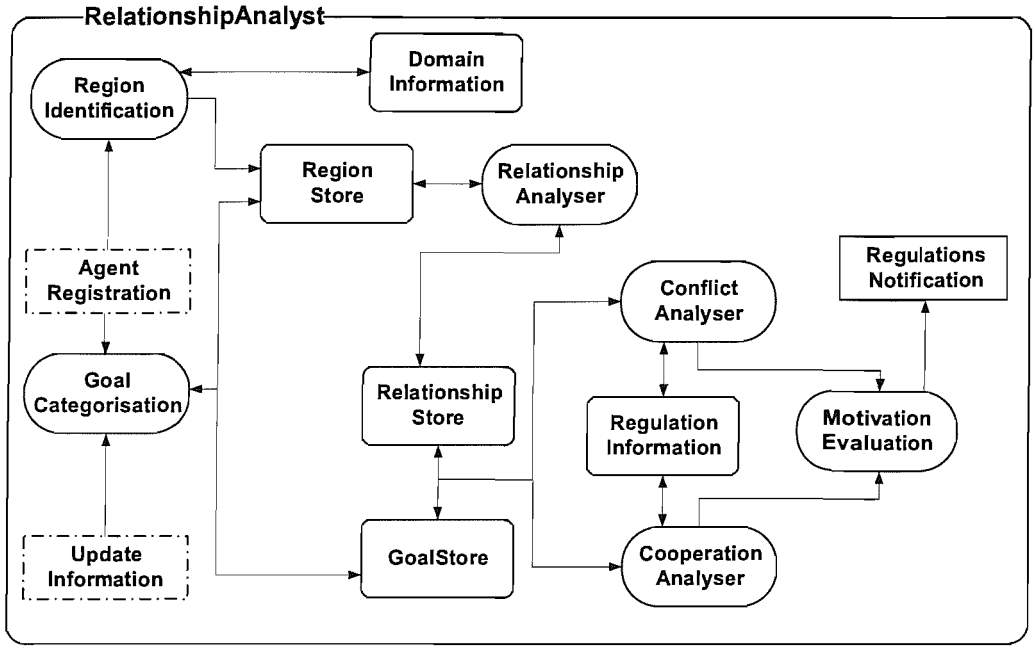

FIGURE A.3: Relationship analysis agent architecture

mation about the agent. The component can produce REQUEST statements for the relationship table, goal and regulation information; and, INFORM statements with the relevant regulations. The *CooperationAnalyser* uses the same types of information as the *ConflictAnalyser*, but with the aim of identifying possibilities for cooperation. It produces the required REQUEST statements to get information about regulations and INFORM statements with the regulations that apply to the situation under consideration. Finally, the *MotivationEvaluation* controller evaluates the regulations to determine which would offer the greatest utility to the agent. It accepts INFORM statements with the relevant regulations and produces EXECUTE statements with the regulations to be sent to agents.

The only actuator for the *RA* agent is the *RegulationsNotification* actuator that accepts EXECUTE statements and sends messages to agents for which relevant regulations have been identified.

A.3.2 Behavioural Specification

The links between components and the overall architecture are shown in Figure A.3. The execution sequence for this agent is a combination of a sequential and an event-based model. Each component is called to execute whenever it receives a statement from another component. This means that the agent first begins operation when a profile arrives at the *AgentRegistration* sensor. This sensor then notifies the *RegionIdentification* controller which in turn sends statements and

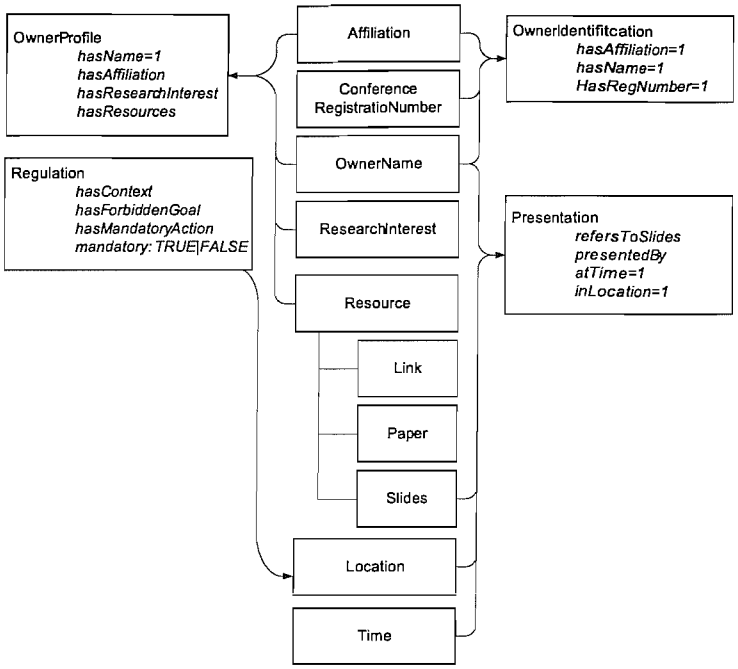

FIGURE A.4: *act*SMART Implementation

causes the activation of the other components it is connected to, and so on. The event-based execution sequence drives this architecture, since registrations arrive at irregular times and the agent might constantly cycle through execution sequence otherwise. In addition, at regular intervals the architecture cycles through an execution of the controllers (to which infostores may respond) to deal with any new information.

A.4 User Agent Architecture

A.4.1 Descriptive Specification

Attributes

The attributes describing the user agent represent features such as the name of the owner, the user agent's current location, the research interests of the owner, and so on. Now, it is necessary to make clear exactly how can attributes be represented at the implementation level so as to make clear the correspondence between the abstract notion of attributes in SMART and their technical

realisation. To this end, we adopt the approach provided by OWL [143] for describing concepts. In essence, OWL is an extension of RDF [85] to provide the capability to specify ontologies composed of taxonomies of classes and inference rules.

Attributes in SMART are mapped to the notion of *individuals* in OWL, where an individual represents a describable thing in the domain we are interested in. For example, the attribute *RonaldAshri* is an individual. Individuals, can be related through properties, which are binary relations between individuals. Therefore we could say that the property *hasResearchInterest* links the individual *RonaldAshri* to an individual *AgentModels*. Now, individuals can be grouped within classes which can state precisely the requirements for membership to a class. We could, for example, define a class *OwnerName* and indicate that *RonaldAshri* belongs to that class. Classes, therefore, enable us to define *concepts* which individuals can then instantiate.

Furthermore, we can define the *OwnerProfile* class to have a property *hasResearchInterest*, which refers to the *ResearchInterest* class. An instantiation of these classes with specific individuals, such as *RonProfile–hasResearchInterest–AgentModels* would define an attribute as understood in SMART.

All the classes used to define attributes and the relationships between them are illustrated in Figure A.4. Note that we can also place restrictions on the cardinality of the attributes that can be referred to by a property. For example, in the case of *OwnerProfile*, the property *hasName* is restricted by the fact that one and only one name *must* exist for that attribute to be valid. If a cardinality is not defined, then any number of instances of an attribute could be defined and related through the property. Note that we use only a very limited set of the capabilities of OWL, since our aim is simply to illustrate that such a technology has an important role to play in aiding the specification of multi-agent systems and facilitating run-time operation. Below, we provide a short explanation of each attribute, along with some simplifying assumptions we make about how such information is treated within the context of the application.

Affiliation The *Affiliation* attribute refers to the organisation, such as University of Southampton, that the agent is affiliated with.

Conference Registration Number The *ConferenceRegistrationNumber* attribute is an identifying number issued by the conference site at the moment that the user registers at the site.

OwnerName The *OwnerName* is simply the name of the owner of the agent.

ResearchInterest The *ResearchInterest* attribute refers to the research interests of the owner of the agent.

Resource The *Resource* attribute describes resources that are available through the agent. There are three types of resources in our case. The *Link* attribute represents URLs that a user may wish to make available to other users. The *Paper* attribute refers to publications that the agent can make directly available from the device it is operating. Finally, the *Slides* attribute refers to slides of presentations that the agent can make available. Clearly, all these attributes could be further elaborated to provide more information about the exact content of the resources. However, such a level of detail in not required for the purposes of our demonstration application.

Location The *Location* attribute refers to the current location of the agent. Once more, we do not attempt to provide detailed information about the location of the agent. The value of the *Location* attribute changes as the user changes rooms within the conference site, as long as there are devices within the room that can provide the agent with its new location, and the agent itself is able to communicate with such devices. In the case where the location cannot be ascertained, the value of the attribute should simply be the name of the entire conference site, set upon registration with the conference.

Time The *Time* attribute is simply a reference to a point in time providing the day, month, year and and time of day.

OwnerIdentification The *OwnerIdentification* attribute is a composite attribute whose purpose is to identify the owner of the agent. The attribute has three properties: the *hasAffiliation* property refers to one *Affiliation* attribute; the *hasName* property refers to one *Name* attribute; and, the *hasRegNumber* property refers to one *RegistrationNumber* property.

Presentation The *Presentation* attribute refers to a presentation that the owner may be giving during the conference. The attribute is composite and has the following properties: the *refersToSlides* property can be used to describe the slides that will be used at the presentation through the *Slides* attribute; the *presentedBy* property can be used to refer to the owner name through the *OwnerName* attribute; the *atTime* property can provide the time of the presentation through the *Time* attribute; and, the *Location* property can be used to provide the location of the presentation through the *Location* attribute.

OwnerProfile The *OwnerProfile* attribute is used to provide a description of the owner, includ-

ing research interests and resources that the owner has on the device. It is a composite attribute with properties for the name of the owner, their affiliation, their research interests and resources.

Regulations Finally, the *Regulations* attribute describes what the agent is allowed or not allowed to do with respect to interacting with other agents and services within the conference site. Each regulation has a set of mandatory goals and forbidden goals within a specific context. The *hasContext* simply refers to the location within which the regulation is applicable, since relationship analysis agents operate within specific locations.

Capabilities

The capabilities of the agent can broadly be divided into the discovery and access to physical devices and information services and the provision and access to information and resources from other agents. We avoid here a lengthy description since we have discussed these issues throughout Chapter 6 and will touch upon them later on.

Goals

The goals of the agent can also broadly be described as assisting the user in discovering and employing physical devices and information services, and handling the interaction with other user agents by exchanging information and resources.

A.4.2 Structural Specification

The components that make up the agent architecture for the user agent are shown in Figure A.5. The links between components are not drawn since they would overly complicate the figure, and components are grouped according to the functionality they collectively offer the agent. We avoid here a lengthy description of each individual component, but rather describe the broad functionality offered by sets of components as indicated by their divisions within squares in the figure.

Sensors There are four sensors for the user agent: the *UserInput* sensor transmits the information provided by the user; the *WiFiDeviceIncomingMessages* sensor can accept messages

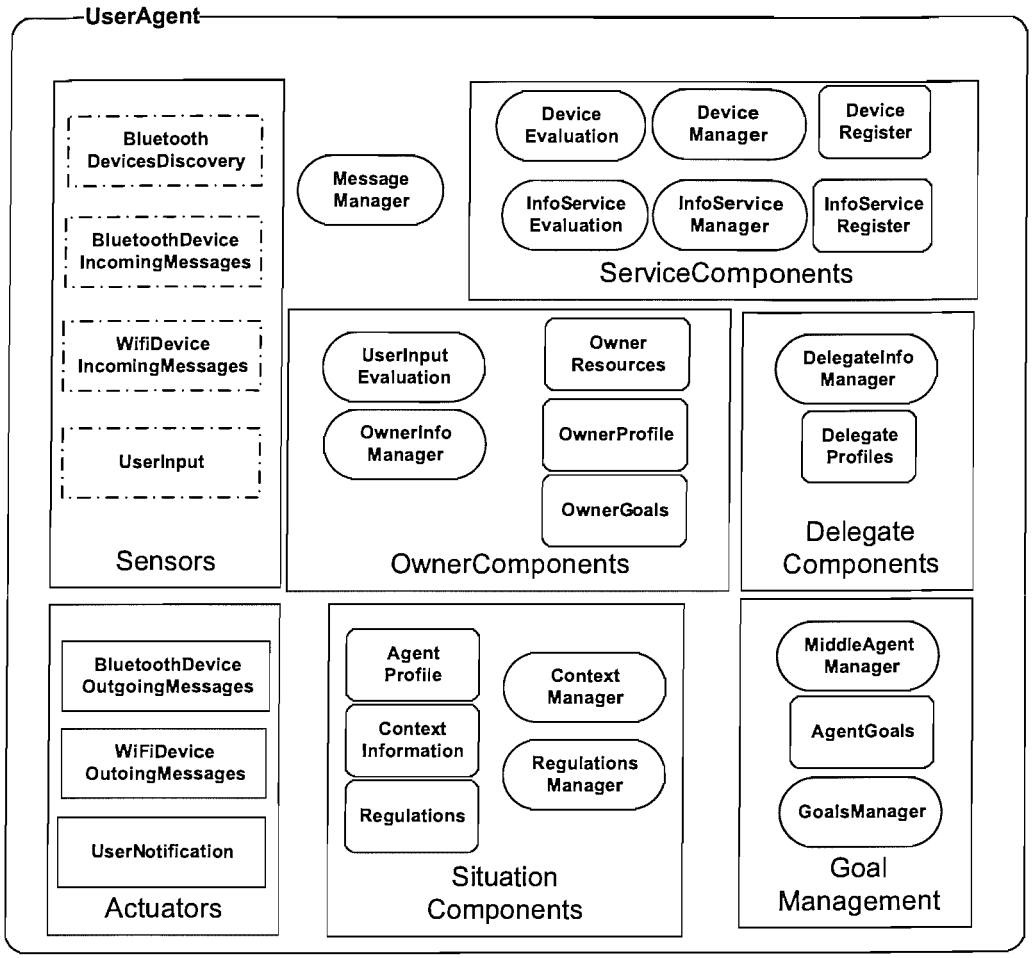

FIGURE A.5: User Agent Architecture

from devices using the 802.11b wireless protocol; the *BluetoothDeviceIncomingMessages* sensor accepts messages from Bluetooth devices; and the *BluetoothDevicesDiscovery* sensor handles and reports on the discovery of Bluetooth devices.

Actuators The actuators of the user agent essentially mirror the sensors. The *BluetoothDeviceOutgoingMessages* actuator is used to send messages to Bluetooth devices, the *WiFiDeviceOutgoingMessages* actuator is used for communication with 802.11b devices, while *UserNotification* actuator is used to send messages to the user through the device screen.

Owner Components The owner components handle information about the owner and the input from the owner. The *OwnerResources* infostore stores resources such as papers, slides and links; The *OwnerProfile* infostore stores information about the owner such as their affiliation and research interes; the *OwnerGoals* infostore stores the user's goals based on the user input; and, finally, the *UserInputEvaluation* controller evaluates input from the user and updates the relevant information stores, while the *OwnerInfoManager* controller

manages the updating of the owner profile and owner goals.

Situation Components The situation components manage information relating to the current context of the user agent, the agent profile, and the regulations that apply to that agent. The context is defined as simply the location of the agent, and the current time. The agent profile defines the information that can be sent to a middle agent to indicate the services offered by the agent. Finally, regulations indicate what goals are mandated and prohibited in a specific context. The *RegulationsManager* and *ContextManager* controllers update this information.

Service Components The service components are dedicated to handling the interactions with either physical devices or information services: the *DeviceEvaluation* and *InfoServiceEvaluation* controllers evaluate descriptions of information services against the required devices; the *DeviceRegister* and *InfoServiceRegister* infostores store descriptions of services; and, finally, the *DeviceManager* and *InfoServiceManager* controllers handle interactions with the registered services.

Delegate Components The delegate components simply store profiles provided by other user agents. The *DelegateInfoManager* is provided with such profiles and updates the *DelegateProfiles* infostore.

Goal Management The goal management components coordinate the execution of other components based on the agent's current goals. They also control interaction with middle agents when use of other agents is required. The *AgentGoals* infostore contains the agent's current goals, while the *GoalManager* controller uses information on current goals and sends appropriate statements to other components in order to achieve those goals. The *MiddleAgentManager* controller creates appropriate messages to request information on services from the middle agent.

MessageManager Finally, the *MessageManager* controller handles the routing of incoming messages to appropriate components within the architecture. It uses a set of basic rules that define which controller should first handle a message based on where the message is coming from.

A.4.3 Behavioural Specification

The execution sequence of the agent is a combination of a periodic execution of certain components, along with an event-based execution of components. The event-based behaviour is mainly used to handle interaction that is initiated by other agents or the user. For example, a user can enter information at any time, which may cause the owner profile, resources or owner goals to be updated. Similarly, Bluetooth devices may be discovered at any time as the agent changes locations, and any relevant information is registered at the *DeviceRegister*. In addition, all infostores execute as soon as they receive a statement.

The *GoalManager* periodically executes to check whether there are any goals within the *OwnerGoals* infostore. If there are, then they are retrieved and placed within the *AgentGoals* infostore, indicating that they are now active goals that the agent will attempt to achieve. If the goal relates to the use of a physical device or an information service, the *GoalManager* queries the *DeviceRegister* or *InfoServiceRegister* in order to identify whether the agent has access to a device or service able to satisfy the goal. If such a device does not exist, then the *GoalManager* makes use of the *MiddleAgentManager* to submit a query for the required service. Once a reply is received, it is directed, through the *MessageManager* to the middle agent, which informs the *DeviceEvaluation* controller, in turn updating the *DeviceRegister*. When the *Goalmanager* executes once more, it identifies whether an appropriate device has been found and instructs the *DeviceManager* to interact with the device requesting the appropriate action to be taken. In the case of information services, the replies from the queries are then sent through the *UserNotification* component to the user's screen.

If the goal relates to collecting information about specific types of users, such as all users with the same interests as the owner, then the *DelegateInfoManager* component and the *MiddleAgentManager* are used to gain that information.

Interaction with other user agents can take place when a user agent directly contacts another user agent, or whenever a user agent is notified about other user agents. The *OwnerInfoManager* controller handles request for information about the agent's owner, while the *DelegateInfoManager* controller handles messages providing information about other agents.

Bibliography

[1] M. P. Singh A. K. Jain, M. Aparico. Agents for process coherence in virtual enterprises. *Communications of the ACM*, 42(3):62–69, 1999.

[2] J.-R. Abrial, E. Börger, and H. Langmaack, editors. *Formal Methods for Industrial Applications: Specifying and Programing the Steam Boiler*. Springer, 1996.

[3] B. Adams, C. Breazeal, R. A. Brooks, and B. Scassellati. Humanoid Robots: A New Kind of Tool. *IEEE Intelligent Systems*, 15(4):25–31, 2000.

[4] A. Ankolenkar, M. Burstein, J. R. Hobbs, O. Lassila, D. L. Martin, D. McDermott, S. A. McIlraith, S. Narayanan, M. Paolucci, T. R. Payne, and K. Sycara. DAML-S: Web Service Description for the Semantic Web. In I. F. Cruz, S. Decker, J. Euzenat, and D. L. McGuinness, editors, *The First Semantic Web Working Symposium*, pages 411–430. Stanford University, California, 2001.

[5] K. Arnold, B. O'Sullivan., R. W. Scheifler, J. Waldo, and A. Wollrath. *The Jini Specification*. Addison-Wesley, 1999.

[6] R. Ashri and M. Luck. Paradigma: Agent implementation through Jini. In A. M. Tjoa, R.R. Wagner, and A. Al-Zobaidie, editors, *Eleventh International Workshop on Databases and Expert System Applications*, pages 453–457. IEEE Computer Society, 2000.

[7] R. Ashri and M. Luck. An Agent Construction Model for Ubiquitous Computing Devices. In *Proceedings of the Fifth Agent-Oriented Software Engineering Workshop*, 2004 (to appear).

[8] R. Ashri, M. Luck, and M. d'Inverno. Infrastructre Support for Agent-based Development. In M. d'Inverno, M. Luck, M. Fisher, and C. Preist, editors, *Foundations and Applications of Multi-Agent Systems*, volume 2403 of *LNAI*, pages 73–88. Springer, 2002.

[9] R. Ashri, M. Luck, and M. d'Inverno. On Identifying and Managing Relationships in Multi-Agent Systems. In G. Gottlob and T. Walsh, editors, *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, pages 743–748. Morgan Kaufmann Publishers, 2003.

[10] R. Ashri, M. Luck, and M. d'Inverno. A typology of relationships and goals for regulation and coordination. In *Proceedings of ECAI Workshop in Coordination in Emergent Agent Societies*, 2004 (to appear).

[11] R. Ashri, M. Luck, and M. d'Inverno. Identifying opportunities and constraints for goal achievement through relationship analysis. In *Poster Proceedings of the 3rd International Conference on Autonomous Agents and Multi-Agent Systems*, 2004 (to appear).

[12] R. Ashri, I. Rahwan, and M. Luck. Architectures for Negotiating Agents. In V. Marik, J. Muller, and M. Pechoucek, editors, *Mutli-Agent Systems and Applications III*, volume 2691 of *LNAI*, pages 136–146. Springer, 2003.

[13] R. Axelrod. An evolutionary approach to norms. *The American Political Science Review*, 80(4):1095–1111, 1986.

[14] K.S. Barber, R. McKay, M. MacMahon, C.E. Martin, D.N. Lam, A. Goel, D.C. Han, and J. Kim. Sensible Agents: An Implemented Multi-Agent System and Testbed. In *Proceedings of the Fifth International Conference on Autonomous Agents*, pages 92–99. ACM Press, 2001.

[15] C. Bartolini, C. Preist, and N. R. Jennings. Architecting for Reuse: A Software Framework for Automated Negotiation. In F. Giunchiglia, J. Odell, and G. Weiß, editors, *Agent-Oriented Software Engineering III*, volume 2585, pages 88–100. Springer, 2002.

[16] B. Bauer, J .Muller, and J. Odell. Agent UML: A formalism for specying multiagent software systems. *International Journal on Software Engineering and Knowldge Engineering*, 11(3):207–230, 2001.

[17] J. Baumann, F. Hohl, K. Rothermel, M. Strasser, and W. Theilmann. MOLE: A mobile agent system. *Software - Practice and Experience*, 32(6):575–603, 2002.

[18] P. Bellavista, A. Corradi, and C. Stefanelli. A secure and open mobile agent programming environment. In *Proccedings of the Fourth International Symposium on Autonomous Decentralized Systems*, pages 238–245. IEEE Computer Society Press, 1999.

[19] P. Bellavista, A. Corradi, and C. Stefanelli. A mobile agent infrastructure for the mobility support. In *Proceedings of the 2000 ACM symposium on Applied computing*, pages 539–545. ACM Press, 2000.

[20] F. Bellifemine, A. Poggi, and G. Rimassa. Developing multi-agent systems with a FIPA-compliant agent framework. *Software - Practice and Experience*, 31(2):103–128, 2001.

[21] F. Bellifemine, A. Poggi, and G. Rimassa. Developing Multi-agent Systems with JADE. In C. Castelfranchi and Y. Lespérance, editors, *Intelligent Agents VII. Agent Theories Architectures and Languages*, volume 1986 of *LNCS*, pages 89–103. Springer, 2001.

[22] F. Bellifemine, A. Poggi, and G. Rimassa. JADE: A FIPA2000 compliant agent development environment. In 216-217, editor, *Proceedings of the 5th International Conference on Autonomous Agents*. ACM Press, 2001.

[23] T. Bellwood, L. Clement, and C. von Riegen. UDDI Version 3.0.1- UDDI Spec Technical Committee Specification. Technical report, OASIS, 2003.

[24] F. Bergenti and A. Poggi. Ubiquitous Information Agents. *International Journal of Cooperative Information Systems*, 11(3–4):231–244, 2002.

[25] T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, May 2001.

[26] D. Bert, J.P. Bowen, M.C. Henson, and K. Robinson, editors. *ZB 2002: Fomal Specification and Development in Z and B: 2nd International Conference of B and Z Users*, volume 2272 of *LNCS*. Springer, 2002.

[27] D. Bert, J.P. Bowen, S. King, and M. Walden, editors. *ZB 2003: Formal Specification and Development in Z and B: Third International Conference of B and Z Users*, volume 2651 of *LNCS*. Springer, 2003.

[28] C. Bicchieri. Norms of cooperation. *Ethics*, 100(4):838–861, 1990.

[29] R. H. Bordini, A. L. C. Bazzan, R. d. O. Jannone, D. M. Basso, R. Maria Vicari, and V. R. Lesser. AgentSpeak(XL): efficient intention selection in BDI agents via decision-theoretic task scheduling. In *The First International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 1294–1302. ACM Press, 2002.

[30] J. Bowen. *Formal Specification and Documentation using Z: A case study approach.* International Thomson Computer Press, 1996.

[31] J. P. Bowen and M. G. Hichley. Seven more myths of formal methods. *IEEE Software*, 12(4):34–41, 1995.

[32] J. M. Bradshaw, M. Greaves, H. Holmack, T. Karygiannis, W. Jansen, B. G. Silverman, N. Suri, and A. Wong. Agents for the Masses. *IEEE Intelligent Systems*, 14(2):53–63, 1999.

[33] M. E. Bratman, D. Israel, and M. E. Pollack. Plans and Resource-Bounded Practical Reasoning. *Computational Intelligence*, 4(4):349–355, 1988.

[34] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler. Extensible Markup Language (XML) 1.0 (Second Edition). Technical report, W3C, 2000.

[35] F.M. T. Brazier, C.M. Jonker, and J. Treur. Principles of Component-Based Design of Intelligent Agents. *Data and Knowledge Engineering*, 41:1–28, 2002.

[36] F.M. T. Brazier, C.M. Jonker, J. Treur, and N.J.E. Wijngaards. Compositional Design of a Generic Design Agent. *Design Studies Journal*, 22:439–471, 2001.

[37] F.M.T. Brazier, B.M. Dunin-Keplicz, N.R. Jennings, and J. Treur. DESIRE: Modelling multi-agent systems in a compositional formal framework. *International Journal of Cooperative Information Systems*, 6(1):67–94, 1997.

[38] W. Briggs and D. Cook. Flexible Social Laws. In C. Mellish, editor, *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, pages 688–693. Morgan Kaufmann, 1995.

[39] M. L. Brodie. The promise of distributed computing and the challenges of legacy information systems. In David K. Hsiao, Erich J. Neuhold, and Ron Sacks-Davis, editors, *Proceedings of the IFIP WG 2.6 Database Semantics Conference on Interoperable Database Systems (DS-5)*, IFIP Transactions, pages 1–31. North-Holland, 1993.

[40] R. A. Brooks. A robust layered control system for a mobile robot. *IEEE Jouranl of Robotics and Automation*, 1(2):14–23, 1986.

[41] R. A. Brooks. Intelligence without representation. *Artificial Intelligence*, 47(1–3):139–159, 1991.

[42] J. Bryson and L. A. Stein. Architectures and Idioms: Making Progress in Agent Design. In C. Castelfranchi and Y. Lespérance, editors, *Intelligent Agents VII. Agent Theories Architectures and Languages*, volume 1986, pages 73–88. Springer, 2001.

[43] K. Bryson, M. Luck, M. Joy, and D. Jones. Agent Interaction for Bioinformatics Data Management. *Applied Artificial Intelligence*, 15(10):917–947, 2001.

[44] P. Busetta, R. Rnnquist, A. Hodgson, and A. Lucas. JACK Intelligent Agents - Components for Intelligent Agents in Java. *Agentlink News*, (2):2–5, 1999.

[45] G. Caire, R. Evans, P. Massonet, W. Coulier, F. J. Garijo, J. Gomez, J. Pavon, F. Leal, P. Chainho, P.E. Kearney, and J. Stark. Agent-Oriented Analysis Using MESSAGE/UML. In M. Wooldridge, G. Weiss, and P. Ciancarini, editors, *Agent-Oriented Software Engineering II*, volume 2222 of *LNCS*, pages 119–135. Springer, 2001.

[46] J. Camp and Y.T. Chien. The internet as public space: concepts, issues, and implications in public policy. *ACM SIFAS Computers and Society*, 30(3):13–19, 2000.

[47] A. Cassandra, D. Chandrasekara, and M. Nodine. Capability-based agent matchmaking. In C. Sierra, M. Ginia, and J. S. Rosenschein, editors, *Proceedings of the Fourth International Conference on Autonomous Agents*, pages 201–202, 2000.

[48] C. Castelfranchi. From individual intentions to gorups and organisations. In V. Lesser and L. Gasser, editors, *Proceedings of the First International Conference on Multi-Agent Systems*, pages 196–196. AAAI Press/MIT Press, 1995.

[49] C. Castelfranchi, F. Dignum, C. M. Jonker, and J. Treur. Deliberative Normative Agents. Principles and Architecture. In N. Jennings and Y. Lesperance, editors, *Intelligent Agents IV (ATAL99)*, volume 1757 of *LNCS*, pages 364–378. Springer, 2000.

[50] C. Castelfranchi, M. Miceli, and A. Cesta. Dependece relations among autonomous agents. In E. Werner and Y. Demazeau, editors, *Decentralised Artificial Intelligence*, pages 215–231. Elsevier, 1992.

[51] H. Chalupsky, Y. Gi, C. A. Knoblock, K. Lerman, J. Oh, D. Pynadath, T. A. Russ, and M. Tambe. Electric Elves : Applying Agent Technology to Support Human Organizations. In H. Hirsch and S. Chien, editors, *International Conference of Innovative Application of Artificial Intelligence (IAAI'01)*, pages 51–58. AAAI, 2001.

[52] H. Chalupsky, Y. Gil, C. A. Knoblock, K. Lerman, J. Oh, D. Pynadath, T. A. Russ, and M. Tambe. Electric Elves: Applying Agent Technology to Support Human Organisations. In H. Hirsh and S. Chien, editors, *Proceedings of the Thirteenth Innovative Applications of Artificial Intelligence Conference*, pages 51–58. AAAI, 2001.

[53] J. Cheesman and J. Daniels. *UML Components: A Simple Process for Specifying Component-Based Software*. Addison-Wesley, 2000.

[54] B. Chellas. *Modal Logic: An Introduction*. Cambridge University press, 1980.

[55] P. Ciancarini, R. Tolksdorf, and F. Zambonelli. A Survey of Coordination-Middleware for XML-Centric Applications. *The Knowledge Engineering Review*, 17(4), 2003.

[56] E. Clarke and J. Wing. Formal Methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996.

[57] Philip R. Cohen and Hector J. Levesque. Intention is choice with commitment. *Artificial intelligence*, 42(2–3):213–261, 1990.

[58] B.P. Collins, J.E. Nicholls, and I.H. Sorensen. Introducing Formal Methods: The CICS experience with Z. In B. Neumann, D. Simpsona, and G. Slater, editors, *Mathematical Structures for Software Engineering*. Oxford University Press, 1991.

[59] R. Conte and C. Castelfranchi. *Cognitive and Social Action*. UCL Press, 1995.

[60] R. Conte and C. Castelfranchi. Norms as mental objects. From normative beliefs to normative goals. In C. Castelfranchi and J. P. Muller, editors, *From Reaction To Cognition*, volume 957 of *LNCS*, pages 186–196. Springer, 1995.

[61] R. Conte and C. Castelfranchi. Simulating multi-agent interdependencies: A two-way approach to the macro-micro link. In K. Troitzsch, U. Mueller, N. Gilbert, and J.E. Doran, editors, *Social Science Microsimulation*, pages 394–415. Springer, 1998.

[62] R. Conte, C. Castelfranchi, and F. Dignum. Autonomous norm-acceptance. In J. Muller, M. Singh, and A. Rao, editors, *Intelligent Agents V (ATAL98)*, volume 1555 of *LNCS*, pages 319–333. Springer, 1999.

[63] D. D. Corkill and V. Lesser. The use of meta-level control for coordination in a distributed problem solving network. In A. H. Bond and L. Gasser, editors, *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, pages 748–756. Morgan Kaufmann Publishers, 1983.

[64] Microsoft Corporation. Universal plug and play device architecture. Technical Report V1.0, Microsoft Corporation, 2000.

[65] M. Cossentino and C. Potts. A Case Tool Supported Methodology for the Design of Multi-Agent Systems. In *The 2002 International Conference on Software Engineering Research and Practice*, 2002.

[66] R. Davis. What are intelligence? And Why? *AI Magazine*, 19(1):91–111, 1998.

[67] R. Davis and R.G. Smith. Negotiation as a metaphor for distributed problem solving. *Artificial Intelligence*, 20(1):63–109, 1983.

[68] D. de Roure, N. R. Jennings, and N. Shadbolt. The Semantic Grid: A future e-Science infrastructure. In F. Berman, G. Fox, and A.J.G. Hey, editors, *Grid Computing: Making the Gloab Infrastructure a Reality*, pages 437–470. Wiley, 2003.

[69] K. Decker, K. Sycara, and M. Williamson. Middle-agents for the internet. In *Proceedings of the 15th Joint Conference on Artificial Intelligence*, volume 1, pages 573–578. Morgan Kaufmann, 1997.

[70] S. Deering and R. Hinden. Internet Protocol, Version 6 (IPv6) Specification. Technical Report RFC2460, IPV6.org, 1998.

[71] S. A. DeLoach, M. F. Wood, and C. H. Sparkman. Multiagent Systems Engineering. *The International Journal of Software Engineering and Knowledge Engineering*, 11(3), 2001.

[72] G. DeSanctis and B. M. Jackson. Co-ordination of information technology management: Team based structures and computer-based communication systems. *Journal of Management Information Sciences*, 4(10):85–110, 1994.

[73] F. Dignum. Autonomous Agents with Norms. *Artificial Intelligence and Law*, (7):69–79, 1999.

[74] V. Dignum and F. Dignum. The knowledge market: Agent-mediated knowledge sharing. In V. Marík and J. Müller, editors, *Multi-Agent Systems and Applications III*, volume 2691 of *LNCS*, pages 168–179. Springer, 2003.

[75] M. d'Inverno, D. Kinny, M. Luck, and M. Wooldridge. A Formal Specification of dMARS. In M. P. Singh, A. S. Rao, and M. Wooldridge, editors, *Intelligent Agenrs IV: Proceedings of the Fourth International Workshop on Agent Theories, Architectures and Languages*, volume 1365 of *LNCS*, pages 155–176. Springer, 1996.

[76] M. d'Inverno and M. Luck. A Formal View of Dependece Networks. In C. Zhang and D. Lukose, editors, *Distributed Artificial Intelligence Architecture and Modelling: Proceedings of the First Australian Workshop on Distributed Artificial Intelligence*, volume 1087 of *LNCS*, pages 155–129. Springer, 1996.

[77] M. d'Inverno and M. Luck. Formalising the contract-net as a goal-directed system. In W. Van de Velde and J. W. Perram, editors, *Agents Breaking Away, 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, volume 1038 of *LNCS*, pages 72–85. Springer, 1996.

[78] M. d'Inverno and M. Luck. Development and Applications of a Formal Agent Framework. In *First IEEE International Conference on Formal Engineering Methods*, pages 222–231. IEEE Computer Society, 1997.

[79] M. d'Inverno and M. Luck. Engineering AgentSpeak(L): A Formal Computational Model. *Journal of Logic and Computation*, 8(3):233–260, 1998.

[80] M. d'Inverno and M. Luck. Formal agent development: Framework to system. In J.L. Rash, C.A. Rouffand, W. Truszkowski, D. Gordon, and M.G. Hinchey, editors, *Formal Approaches to Agent-Based Systems: First International Workshop*, volume 1871 of *LNCS*, pages 133–147. Springer, 2001.

[81] M. d'Inverno and M. Luck. *Understanding Agent Systems*. Springer, 2001.

[82] M. d'Inverno and M. Luck. *Understanding Agent Systems*. Springer, 2nd edition, 2004.

[83] D. D'Souza and A. Wills. *Objects Components and Frameworks with UML*. Addison-Wesley, 1998.

[84] N. Dulay, N. Damianou, E. Lupu, and M. Sloman. A policy language for the management of distributed agents. In M. J. Wooldridge, G. Weiss, and P. Ciancarini, editors, *Agent-Oriented Software Engineering II*, volume 2222, pages 84–100. Springer-Verlag, 2001.

[85] Dave Beckett (ed). RDF/XML Syntax Specification. Technical report, W3C, 2003.

[86] T. Eiter and V.S. Subrahmanian. Heterogeneous Active Agents, II: Algorithms and Complexity. *Artificial Intelligence*, 108(1–2):257–307, 1999.

[87] T. Eiter, V.S. Subrahmanian, and G. Pick. Heterogeneous Active Agents, I:Semantics. *Artificial Intelligence*, 108(1–2):179–255, 1999.

[88] T. Eiter, V.S. Subtahmanian, and T. Rogers. Heterogeneous Active Agents, III:Polunomially implementable agents. *Artificial Intelligence*, 117(1):107–167, 2000.

[89] M. Esteva, D. de la Cruz, and C. Sierra. ISLANDER: an electronic institutions editor. In *The First International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 1045–1052. ACM Press, 2002.

[90] Peyman Faratin. *Automated Service Negotiation Between Autonomous Computational Agents*. PhD thesis, UCL, Queen Mary and Westfield, Dept. of Electronic Engineering, 2000.

[91] S. Fatima, M. Wooldridge, and N. R. Jennings. Multi-issue negotiation under time constraints. In C. Castelfranchi and L. Johnson, editors, *Proceedings of the Second International Conference on Autonomous Agents and Multi-Agent Systems*, pages 143–150. ACM Press, 2002.

[92] I. A. Ferguson. *TouringMachines: An Architecture for Dynamic, Rational, Mobile Agents*. PhD thesis, Clare College, Cambridge University, 1992.

[93] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. Technical report, The Internet Society, 1999.

[94] T. Finin, A. Joshi, L. Kagal, O. V. Patsimor, S. Avancha, V. Korolev, H. Chen, F. Perich, and R. S. Cost. Intelligent Agents for Mobile and Embedded Devices. *International Journal of Cooperative Information Systems*, 11(3–4):205–230, 2002.

[95] T. Finin, Y. Labrou, and J. Mayfield. KQML as an agent communication language. In Jeffrey Bradshaw, editor, *Software Agents*. MIT Press, 1997.

[96] K. Fischer, J. P. Müller, and M. Pischel. A pragmatic BDI architecture. In M. Wooldridge, J. P. Müller, and M. Tambe, editors, *Intelligent Agents II, Agent Theories, Architectures, and Languages*, volume 1037, pages 203–218. Springer, 1996.

[97] T. L. Fox, R. Pedigo, and S. W. Remingon. Building the Virtual Organization with electronic communication. *EM - Electronic Contracting*, 8(3):43–45, 1998.

[98] F. Gandon and N. Sadeh. Semantic Web Technologies to Reconcile Privacy and Context Awareness. *Web Semantics Journal*, 1(3), 2004.

[99] A. F. Garcia, C. J. P. de Lucena, F. Zambonelli, A. Omicini, and J. Castro, editors. *Software Engineering for Large-Scale Multi-Agent Systems, Research Issues and Practical Applications*, volume 2603 of *LNCS*. Springer, 2006.

[100] M. R. Genesereth and S. P. Ketchpel. Software agents. *Communications of the ACM*, 37(7):48–53, 1994.

[101] M. P. Georgeff and A. S. Rao. A profile of the Australian AI Institute. *IEEE Expert*, 11(6):89–92, December 1996.

[102] M.P. Georgeff and A.L. Lansky. Reactive reasoning and planning. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, pages 677–682. AAAI Press/MIT Press, 1987.

[103] F. Giunchiglia, J. Mylopoulos, and A. Perini. The TROPOS Software Development Methodology: Processes, Models and Diagrams. In C. Castelfranchi and W. Johnson, editors, *Proceedings of the First International Joint Conference on Autonomous Agents and Multi Agent Systems*, pages 35–36. ACM Press, 2002.

[104] F. Giunchiglia and L. Serafini. Multilanguage hierarchical logics (or: How we can do without modal logics). *Artificial Intelligence*, (65):29–70, 1994.

[105] J. Graham and K. Decker. Towards a Distributed Environment-Centered Agent Framework. In N.R. Jennings and Y. Lesperance, editors, *Intelligent Agents VI Agent Theories, Architectures, and Languages*, volume 1757 of *LNCS*. Springer, 1999.

[106] R. Gray, D. Kotz, G. Cybenko, and D.a Rus. D'Agents: Security in a multiple-language, mobile agent system. In G. Vigna, editor, *Mobile Agents and Security*, volume 1419 of *LNCS*, pages 154–187. Springer-Verlag, 1998.

[107] JSR 118 Expert Group. Mobile Information Device Profile for the Java 2 Micro Edition - Version 2.0. Technical report, Java Community Press, 2002.

[108] W3C HTML Working Group. HTML 4.01 Specification. Technical report, W3C, 1999.

[109] P. Gupta and D. Moitra. Evolving a pervasive IT Infrastructure: a technology integration approach. *Personal and Ubiquitous Computing*, 8(1):31–41, 2004.

[110] Rune Gustavson. Agents with power. *Communications of the ACM*, 42(3):41–47, 1999.

[111] S. Hadjiefthymiades, V. Matthaiou, and L. Merakos. Supporting the www in wireless communications through mobile agents. *Mobile Networks and Applications,*, 7(4):305–313, 2002.

[112] K. Z. Haigh, J. Phelps, and C. W. Geib. An open agent architecture for assisting elder independence. In *Proceedings of the first International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 578–586. ACM Press, 2002.

[113] A. Hall. Seven myths of formal methods. *IEEE Software*, 7(5):11–19, 1990.

[114] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, (21):666–677, 1978.

[115] L. J. Hoffman and L. Cranor. Internet Voting for Public Officials. *Communications of the ACM*, 44(1):69–71, 2001.

[116] N. Howden, R. Runnquist, A. Hodgson, and A. Lucas. JACK Intelligent Agents – Summary of an Agent Infrastructure. In T. Wagner and O. F. Rana, editors, *The 5th Intenational Coference on Autonomous Agents, Workshop on Infrastructure for Agents, MAS and Scalable MAS*, pages 251–257, 2001.

[117] M. J. Hubber. JAM: A BDI-Theoretic mobile agent architecture. In *Proceedings of the Third International Conference on Autonomous Agents*, pages 236–243. ACM Press, 1999.

[118] N. R. Jennings. Specification and implementation of belief, desire, joint-intention architectures for collaborative problem solving. *Journal of Intelligence and Cooperative Information Systems*, 2(3):289–318, 1993.

[119] N. R. Jennings. On being responsible. In *Proceedings of the First International Conference on Multi-Agent Systems*, pages 133–139. AAAI Press/MIT Press, 1995.

[120] N. R. Jennings. On agent-based softare engineering. *Artificial Intelligence*, 117(2):277–296, 2000.

[121] N. R. Jennings, P. Faratin, A. R. Lomuscio, S. Parsons, C. Sierra, and M. Wooldridge. Automated negotiation: prospects, methods and challenges. *International Journal of Group Decision and Negotiation*, 10(2):199–215, 2001.

[122] N. R. Jennings, S. Parsons, P. Noriega, and C. Sierra. On argumentation-based negotiation. In *Proc. of the Int. Workshop on Multi-Agent Systems*, pages 1–7, Boston, USA, 1998.

[123] N. R. Jennings and T. Wittig. ARCHON: Theory and Practice. In N. M. Avouris and L. Gasser, editors, *Distributed Artificial Intelligence: Theory and Praxis*, pages 179–195. Kluwer Academic Press, 1992.

[124] C. B. Jones. *Systematic Software Development using VDM.* Prentice Hall, 2nd edition, 1990.

[125] A. Joshi, T. Finin, and Y. Yesha. Me-Services: A Framework for Secure and Personalized Discovery, Composition and Management of Services in Pervasive Environments. In C. Bussler, R. Hull, S. McIlraith, M.E. Orlowska, B. Pernici, and J. Yang, editors, *Web Services, E-Business, and the Semantic Web: CAiSE 2002 International Workshop, WES 2002, Toronto, Canada, May 27-28, 2002. Revised Papers*, volume 2512 of *LNCS*, pages 248–259. Springer, 2002.

[126] A. Joshi, T. Finin, and Y.Yesha. Agents, Mobility, and M-services: Creating the Next Generation Applications and Infrastructure on Mobile Ad-Hoc Networks. In B. Knig-Ries, K. Makki, S.A.M. Makki, N. Pissinou, and P. Scheuermann, editors, *Developing an Infrastructure for Mobile and Wireless Systems : NSFWorkshop IMWS 2001 Scottsdale, AZ, October 15, 2001. Revised Papers*, volume 2538 of *LNCS*, pages 106–118. Springer, 2002.

[127] T. Juan, A. R. Pearce, and L. Sterling. ROADMAP: extending the GAIA methodology for complex open systems. In *The First International Joint Conference on Autonomous Agents & Multiagent Systems*, pages 3–10. ACM Press, 2002.

[128] L. Kagal, T. Finin, and A. Joshi. A Policy Based Approach to Security for the Semantic Web. In D. Fensel, K. Sycara, and J. Mylopoulos, editors, *Proceedings of the 2nd International Semantic Web Conference*, volume 2870 of *LNCS*, pages 402–418. Springer, 2003.

[129] M. Kahn and C. D. T. Cicalese. CoABS Grid Scalability Experiments. In T. Wagner and O. F. Rana, editors, *Infrastructure for Agents, MAS and scalable MAS, Workshop in Autonomous Agents 2001*, pages 145–152, 2001.

[130] M. Kahn and P. Sage. DARPA Control of Agent-Based Systems Tutorial. In J.M. Bradshaw, editor, *PAAM 2000*, 2000.

[131] S. Kent and K. Seo. Security Architecture for the Internet Protocol. Technical Report RFC2401bis-01, IETF, 2004.

[132] D. Kinny, M. P. Georgeff, and A. S. Rao. A methodology and Modelling Technique for Systems of BDI Agents. In W. Van de Velde and J. W. Perram, editors, *Agents Breaking Away, 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World, Eindhoven, The Netherlands, January 22-25, 1996, Proceedings*, volume 1038 of *LNCS*, pages 56–71. Springer, 1996.

[133] Danny Lange and Mitsuru Oshima. *Programming and Deploying Java(tm) Mobile Agents with Aglets(tm)*. Addisson-Wesley, 1998.

[134] Kevin Lano. *The B Language and Method: A guide to Practical Formal Development*. Springer-Verlag, 1996.

[135] J. Lawrence. LEAP into Ad-Hoc Networks. In T. Finin and Z. Maamar, editors, *Ubiquitous Agents on embedded, wearable and mobile devices*, 2002.

[136] J. Lind. Issues in Agent-Oriented Software Engineering. In P. Ciancarini and M. Wooldridge, editors, *Agent-Oriented Software Engineering*, volume 1957 of *LNCS*, pages 45–48. Springer, 2001.

[137] M. Luck. From definition to development: What next for agent-based systems. *Knowledge Engineering Review*, 14(2):119–124, 1999.

[138] M. Luck, R. Ashri, and M. d'Inverno. *Agent-Based Software Development*. Artech House, 2004.

[139] M. Luck and M. d'Inverno. Engagement and cooperation in motivated agent modelling. In C. Zhang and D. Lukose, editors, *Proceedings of the First Australian DAI Workshop*, volume 1087 of *LNCS*, pages 70–84. Springer, 1996.

[140] M. Luck, P. McBurney, and C. Preist. Agent Technology: Enabling Next Generation Computing (A roadmap for Agent-Based Computing). Technical report, Agentlink, 2003.

[141] E. Lupu and M. Sloman. Towards a role-based framework for distributed systems management. *Journal of Network and Systems Management*, 5(1):5–30, 1997.

[142] K. Matsuda, T. Miyake, and H. Kawai. Culture formation and its issues in personal agent-oriented virtual society: "PAW 2". In *Proceedings of the 4th International Conference on Collaborative Virtual Environments*, pages 17–24. ACM Press, 2002.

[143] D. L. Mcguinness and F. van Harmelen. OWL Web Ontology Language: Overview. http://www.w3.org/TR/2003/PR-owl-features-20031215/.

[144] M .Miceli, A. Cesta, and P. Rizzo. Distributed Artificial Intelligence from a socio-cognitive standpoint: Looking at reasons for interaction. *Artificial Intelligence and Society*, 9:287–320, 1996.

[145] D. Milojicic, A. Messer, P. Bernadat, I. Greenberg, O. Spinczyk, D. Beuche, and W. Schrder-Preikschat. Psi - pervasive services infrastructure. In F. Casati, D. Georgakopoulos, and M.-C. Shan, editors, *Technologies for E-Services: Second International Workshop, TES 2001, Rome, Italy, September 14-15, 2001, Proceedings*, volume 2193 of *LNCS*, pages 187–199. Springer, 2001.

[146] L. Moreau, N. M. Zaini, D. Cruishanck, and D. De Roure. SoFAR: An Agent Framework for Distributed Information Management. In *Intelligent Agent Software Engineering*, pages 49–67. Idea Group Publishing, 2003.

[147] A. Mowshowitz. Virtual Organization. *Communications of the ACM*, 40(9):30–37, 1997.

[148] S. Munroe, M. Luck, and M. d'Inverno. Towards a motivation-based approach for evaluating goals. In *The Second International Joint Conference on Autonomous Agents & Multiagent Systems*, pages 1074–1075. ACM Press, 2003.

[149] M. A. Mupoz, M. Rodriguez, J. Favela, A. I. Martinez-Garcia, and V. M. Gonzalez. Context-aware mobile communications in hospitals. *Computer*, 36(9):38–46, 2003.

[150] J. Mylopoulos, M. Kolp, and J. Castro. UML for Agent-Oriented Software Engineering: The TROPOS Proposal. In M. Gorgolla and C. Kobryn, editors, *UML 2001 - The Unified Modeling Language, Modeling Languages, Concepts and Tools 4th International Conference*, volume 2518 of *LNCS*. Springer, 2001.

[151] R. Nair, M. Tambe, S. Marsella, and T. Raines. Automated assistants for analyzing team behaviours. *Journal of Autonomous Agents and Multi-Agent Systems*, 8(1), 2004.

[152] M. H. Nodine and A. Unruh. Facilitating Open Communication in Agent Systems: The InfoSleuth Infrastructure. In M. P. Singh, A. S. Rao, and M. Wooldridge, editors,

Intelligent Agents IV, Agent Theories, Architectures, and Languages, volume 1365 of *LNCS*. Springer, 1998.

[153] P. Noriega. *Agent-Mediated Auctions: The Fishmarket Metaphor*. PhD thesis, Institut d'Investigacio en Intelligencia Artificial (IIIA), 1997.

[154] P. Noriega and C. Sierra. Towards Layered Dialogical Agents. In J.P. Muller, M. Wooldridge, and N.R. Jennings, editors, *Intelligent Agents III - Agent Theories, Architectures and Languages*, volume 1193 of *LNCS*, pages 173–188. Springer, 1997.

[155] H. Nwana, D. Ndumu, L. Lee, and J. Collis. ZEUS: A Tool-Kit for Building Distributed Multi-Agent Systems. *Applied Artifical Intelligence*, 13(1):129–186, 1999.

[156] S. Oaks, B. Traversat, and L. Gong. *JXTA In a Nutshell*. O'Reilly and Associates, 2002.

[157] P.D. O'Brien and M.E. Wiegand. *Agents of Change in Business Process Management*, volume 1198 of *LNAI*, pages 132–145. Springer, 1997.

[158] D. O'Leary, D. Kuokka, and R. Plant. Artificial intelligence and virtual organizations. *Communications of the ACM*, 40(1):52–59, 1997.

[159] E. Oliveira and A. P. Rocha. Agents Advanced Features for Negotiation in Electronic Commerce and Virtual Organisations Formation Processes. In F. Dignum and C. Sierra, editors, *Agent Mediated Electronic Commerce, The European AgentLink Perspective*, volume 1991, pages 78–97. Springer, 2001.

[160] A. Omicini. SODA: Societies and infrastructures in the analysis and design of agent-based systems. In P. Ciancarini and M. Wooldridge, editors, *Agent-Oriented Software Engineering*, volume 1957 of *LNCS*, pages 185–193, 2001.

[161] A. Omicini, A. Ricci, M. Viroli, and G. Rimassa. Integrating Objective & Subjective Coordination in MultiAgent Systems. In *19th ACM Symposium on Applied Computing (SAC 2004)*, pages 485–491. ACM Press, 2004.

[162] A. Omicini and F. Zambonelli. Coordination of Mobile Information Agents in TuCSoN. *Internet Research: Electronic Networking Applications and Policy*, 8(5):400–413, 1998.

[163] L. Padgham and M. Winikoff. Prometheus: A Methodology for Developing Intelligent Agents. In F. Giunchiglia, J. Odell, and G. Weiß, editors, *Agent-Oriented Software Engineering III*, volume 2585, pages 174–185. Springer, 2002.

[164] M. Paolucci, Z. Niu, K. Sycara, C. Domashnev, S. Owens, and M. van Velsen. Matchmaking to support intelligent agents for portfolio management. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on on Innovative Applications of Artificial Intelligence*, pages 1125–1126. AAAI Press/MIT Press, 2000.

[165] S. Parsons, N. R. Jennings, J. Sabater, and C. Sierra. Agent Specification Using Multi-Context Systems. In M. d'Inverno, M. Luck, M. Fisher, and C. Preist, editors, *Foundations and Applications of Multi-Agent Systems*, volume 2403 of *LNCS*. Springer, 2002.

[166] S. Parsons, C. Sierra, and N. R. Jennings. Agents that reason and negotiate by arguing. *Journal of Logic and Computation*, 8(3):261–292, 1998.

[167] T. R. Payne, R. Singh, and K. Sycara. Browsing Schedules - An Agent-based approach to navigating the Semantic Web. In *International Semantic Web Conference*, pages 469–473, 2002.

[168] M. Pollack and M. Ringuette. Introducing the Tileworld: Experimentally evaluating agent architectures. In *Proceedings of the Eigth National Confenrence on Artificial Intelligence*, 1990.

[169] T. F. La Porta, T. Woo, K. K. Sabnani, and R. Ramjee. Experiences with network-based user agents for mobile applications. *Mobile Networks and Applications*, 3(2):123–141, 1998.

[170] B. Potter, J. Sinclair, and D. Till. *An Introduction To Formal Specification and Z.* International Series in Computer Science. Prentice Hall, 1991.

[171] H. Prakken and G. Vreeswijk. Logics for defeasible argumentation. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic*, volume 4, pages 219–318. Kluwer, 2nd edition, 2002.

[172] I. Rahwan, L. Sonenberg, and F. Dignum. Towards interest-based negotiation. In *Proceedings of the Second International Conference in Autonomous Agents and Multi-Agent Systems*. ACM Press, 2003.

[173] A. Rakotonirainy, J. Indulska, S. Wai Loke, and A. Zaslavsky. Middleware for Reactive Components: An Integrated Use of Context, Roles, and Event Based Coordination. In

R. Guerraoui, editor, *Middleware 2001 : IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg, Germany, November 12-16, 2001. Proceedings*, volume 2218 of *LNCS*, pages 77–98. Springer, 2001.

[174] S. D. Ramchurn, C. Sierra, L. Godo, and N. R. Jennings. A Computational Trust model for Multi-Agent Interactions based on Confidence and Reputation. In R. Falcone, S. Barber, L. Korba, and M. Singh, editors, *Workshop on Deception, Trust, and Fraud in the Second International Joing Conference in Autonomous Agents and Multi-Agent Systems*, pages 69–75, 2003.

[175] A. S. Rao. AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language. In W. Van de Velde and J. W. Perram, editors, *Agents Breaking Away, 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, volume 1038 of *LNCS*, pages 42–55. Spinger, 1996.

[176] A. S. Rao and M. P. Georgeff. Asymmetry Thesis and Side-Effect Problems in Linear-Time and Branching-Time Intention Logics. In John Mylopoulos and Raymond Reiter, editors, *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, pages 498–505. Morgan Kaufmann, 1991.

[177] A. S. Rao and M. P. Georgeff. An Abstract Architecture for Rational Agents. In B. Nebel, C. Rich, and W. R. Swartout, editors, *Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning*, pages 439–449. Morgan Kaufmann, 1992.

[178] A. S. Rao and M. P. Georgeff. A Model-Theoretic Approach to the Verification of Situated Reasoning Systems. In R. Bajcsy, editor, *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, pages 318–324. Morgan Kaufmann, 1993.

[179] A. S. Rao and M. P. Georgeff. BDI-agents: from theory to practice. In *Proceedings of the First International Conference on Multiagent Systems*, pages 312–319. AAAI Press/ The MIT Press, 1995.

[180] A. Ricci, A. Omicini, and E. Denti. Engineering agent societies: a case study in smart environments. In *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 1064–1065. ACM Press, 2002.

[181] J. A. Rodriguez-Aguilar. On the design and construction of Agent-Mediated Institutions. Monograph Series 11, Institut d'Investigacio en Intelligencia Artificial (IIIA), 2002.

[182] J. A. Rodriguez-Aguilar, F. J. Martin, P. Noriega, P. Garcia, and C. Sierra. Towards a test-bed for trading agents in electronic auction markets. *AI Communications*, 11(1):5–19, 1998.

[183] J. Antonio Rodriguez-Aguilar and C. Sierra. Enabling Open Agent Institutions. In K. Dautenhahn, A. H. Bond, L. Canamero, and B. Edmonds, editors, *Socially Intelligent Agents: Creating relationships with computers and robots*. Kluwer, 2002.

[184] A. Ross. *Directives and Norms*. Routledge and Kegan Paul Ltd, 1968.

[185] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Object Techology Series. Addison Wesley, 1998.

[186] S. J. Russel and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 1995.

[187] R. T. Rust and P.K. Kannan. E-service: a new paradigm for business in the electronic environment. *Communications of the ACM*, 46(6):36–42, 2003.

[188] J. Sabater and C. Sierra. REGRET: a reputation model for gregarious societies. In C. Castelfranchi and L. Johnson, editors, *Proceedings of the 1st International Joint Conference onAutonomous Agents and Multi-Agent Systems*, pages 475–482, 2002.

[189] J. Sabater, C. Sierra, S. Parsons, and N. R. Jennings. Engineering Executable Agents using Multi-context Systems. *Journal of Logic Computation*, 12(3):413–442, 2002.

[190] N. Sadeh, T. Chan, L. Van, O. Kwon, and K. Takizawa. A semantic web environment for context-aware m-commerce. In *Proceedings 4th ACM Conference on Electronic Commerce*, pages 268–269, 2003.

[191] D. Saha and A. Mukherjee. Pervasive Computing: A Paradigm for the 21st Century. *IEEE Computer*, 36(3):25–31, 2003.

[192] A. Sahai and C. Morin. Mobile agents for enabling mobile user aware applications. In *Proceedings of the Second International Conference on Autonomous Agents*, pages 205–211. ACM Press, 1998.

[193] G. Diez-Andino Sancho, R. M. Garcia Rioja, and C. Campo. Design of a FIPA-Compliant Agent Platform for Limited Devices. In *Mobile Agents for Telecommunication Applications*, volume 2881 of *LNCS*. Springer, 2003.

[194] Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, 60(1):51–92, 1993.

[195] Y. Shoham and M. Tennenholtz. On Social Laws for Artificial Agent Societies. *Artifial Intelligence*, 73(1–2):231–252, 1995.

[196] J.S. Sichman, Y. Demazeau, R. Conte, and C. Castelfranchi. A social reasoning mechanism based on dependence networks. In *11th European Conference on Artificial Intelligence*, pages 188–192. John Wiley and Sons, 1994.

[197] C. Sierra, N. R. Jennings, P. Noriega, and S. Parsons. A framework for argumentation-based negotiation. In M. Singh, A. Rao, and M. Wooldridge, editors, *Intelligent Agent IV: Proc. of ATAL 1997*, volume 1365 of *LNCS*, pages 177–192. Springer, 1998.

[198] M. P. Singh. A criticial examination of the Cohen-Levesque theory of intention. In *Proceedings of the Tenth European Conference on Artificial Intelligence*, pages 364–368, 1992.

[199] M.P. Singh. *Multiagent Systems: A Theoretical Framework for Intentions, Know-How, and Communications*, volume 799 of *LNCS*. Springer, 1994.

[200] M. Sloman. Policy driven management for distributed systems. *Network and Systems Management*, 2(4):333–360, 1994.

[201] J.M. Spivey. *The Z Notation*. Prentice Hall, 2nd edition, 1992.

[202] I. Srnkovic, B. Hnich, T. Jonsson, and Z. Kiziltan. Specification, Implementation and Deployment of Components. *Communications of the ACM*, 45(10):35–40, 2002.

[203] V.S. Subrahmanian, P. Bonatti, J. Dix, T. Eiter, S. Kraus, F. Ozcan, and R. Ross. *Heterogeneous Agent Systems*. MIT Press, 2000.

[204] N. Suri, M. Carvalho, J. Bradshaw, M. R. Breedy, T. B. Cowin, P. T. Groth, R. Saavedra, and A. Uszok. Enforcement of Communications Policies in Software Agent Systems through Mobile Code. In *4th IEEE International Workshop on Policies for Distributed Systems and Networks*, pages 247–250. IEEE Computer Society, 2003.

[205] K. Sycara, M Paolucci, A. Ankolenkar, and N. Srinivasan. Automated Discovery, Interaction and Composition of Semantic Web Services. *Journal of Web Semantics*, 1(1):27–46, 2003.

[206] K. Sycara, M. Paolucci, M. van Velsen, and J. Giampapa. The RETSINA MAS Infrastructure. *Autonomous Agents and MAS*, 7(1–2), 2003.

[207] K. Sycara, S. Widoff, M. Klusch, and J. Lu. LARKS: Dynamic Matchmaking Among Heterogeneous Software Agents in Cyberspace. *Autonomous Agents and Multi-Agent Systems*, (5):173–203, 2002.

[208] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[209] TAC. *The Trading Agent Competition*. World Wide Web, http://www.sics.se/tac/, 2003.

[210] C. Tessier, L. Chaudron, and H.-J. Muller, editors. *Conflicting Agents: Conflict Management in Multi-Agent Systems*. Kluwer Publishers, 2000.

[211] R. Titmuss, I.B. Crabtree, and C.S. Winter. *Agents, Mobility and Multimedia Information*, volume 1198 of *LNAI*, pages 146–159. Springer-Verlag, 1997.

[212] J. Tretmans, K. Wijbrans, and N. Chaudron. Software Engineering with Formal methods: The Development of a Storm Surge barrier Control System Revisiting Seven Myths of Formal Methods. *Formal Methods in System Design*, 19(2):195–215, 2001.

[213] M. Tsvetovat and K. Sycara. Customer Coalitions in the electronic marketplace. In *Proceedings of the 4th International Conference on Autonomous agents*, pages 263–264. ACM Press, 2000.

[214] R. Tuomela. *The Importance of Us: A Philosophical Sudy of Basic Social Norms*. Stanford University Press, 1995.

[215] R. Tuomela and M. Bonnevier-Tuomela. Norms and Agreements. *European Journal of law, Philosophy and Computer Science*, 5:41–46, 1995.

[216] A. Uszok, J. Bradshaw, R. Jeffers, N. Suri, P. J. Hayes, M. R. Breedy, L. Bunch, M. Johnson, S. Kulkarni, and J. Lott. KAoS Policy and Domain Services: Toward a Description-Logic Approach to Policy Representation, Deconfliction, and Enforcement. In *4th IEEE International Workshop on Policies for Distributed Systems and Networks*, pages 93–98. IEEE Computer Society, 2003.

[217] A. van Lamsweerde. Formal specification: A roadmap. In *Proceedings of The Future of Software Engineering*, pages 147–159. ACM Press, 2000.

[218] J. Vazquez-Salceda and F. Dignum. Modelling Electronic Organizations. In V. Marík and J. Müller, editors, *Multi-Agent Systems and Applications III*, volume 2691, pages 584–593. Springer, 2003.

[219] J. Venn. On the diagrammatic and mechanical representation of propositions and reasonings. *The London, Edinburgh and Dublin Philosophical Magazine and Journal of Science*, (9):1–18, 1880.

[220] W3C. Web Services Activity. http://www.w3.org/2002/ws/.

[221] A. Walker and M. Wooldridge. Undestanding the emergence of conventions in multi-agent systems. In V. Lesser and L. Gasser, editors, *Proceedings of the First International Conference on Multi-Agent Sysetms*. AAAI Press/MIT Press, 1995.

[222] D. N. Walton and E. C. W. Krabbe. *Commitment in Dialogue: Basic Concepts of Interpersonal Reasoning*. SUNY Press, Albany, NY, USA, 1995.

[223] M. Weiser. The computer for the twenty-first century. *Scientific American*, pages 94–104, September 1991.

[224] B. Wellman. Designing the internet for a networked society. *Communications of the ACM*, 45(5):91–96, 2002.

[225] G. J. Wickler. *Using Expressive and Flexible Action Representations to Reason about Capabilities for Intelligent Agent Cooperation*. PhD thesis, University of Edinburgh, 1999.

[226] G. Wills, H. Alani, R. Ashri, R. Crowder, Y. Kalfoglou, and S. Kim. Design issues for agent-based resource locator systems. In *Proceedings of the 4th International Conference on Practical Aspects of Knowledge Management (PAKM'02)*, 2002.

[227] M. Winikoff, L. Padgham, and J. Harland. Simplifying the Development of Intelligent Agents. In *AI2001: Advances in Artificial Intelligence. 14th Australian*, pages 557–568, 2001.

[228] H. Wong and K. Sycara. A Taxonomy of Middle-Agents for the Internet. In *Proceedings of the Fourth International Conferene on Multi-agent Systems*. ACM Press, 2000.

[229] J. Woodcock and J. Davies. *Using Z: Specification, Refinement and Proof*. Prentice Hall International, 1996.

[230] M. Wooldridge. This is MYWORLD: The Logic of an Agent-Oriented DAI Testbed. In M. Wooldridge and N. R. Jennings, editors, *Intelligent Agents, ECAI-94 Workshop on Agent Theories, Architectures, and Languages*, volume 890 of *LNCS*, pages 160–178. Springer, 1995.

[231] M. Wooldridge and P. Ciancarini. Agent-Oriented Software Engineering: The State of the Art. In P. Ciancarini and M. Wooldridge, editors, *Agent-Oriented Software Engineering*, volume 1957 of *LNCS*, pages 1–28. Springer, 2001.

[232] M. Wooldridge and N. R. Jennings. Intelligent agents: Theory and practice. *Knowledge Engineering Review*, 10(2):115–152, 1995.

[233] M. Wooldridge, N.R. Jennings, and D.Kinny. The GAIA Methodology for Agent-Oriented Analysis and Design. *Journal of Agents and Multi-Agent Systems*, 3(3):285–312, 2000.

[234] F. Lopez y Lopez, M. Luck, and M. d'Inverno. Constraining autonomy through norms. In *The First International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 674–681. ACM Press, 2002.

[235] E. Yu. Agent Orientation as a Modelling Paradigm. *Wirtschaftsinformatik.*, 43(2):123–132, 2001.

[236] F. Zambonelli, N. R. Jennings, A. Omicini, and M. Wooldridge. Agent-Oriented Software Engineering for Internet Applications. In A. Omicini, F. Zambonelli, M. Klusch, and R. Tolksdorf, editors, *Coordination of Internet Agents*, chapter 13, pages 326–346. Springer, 2001.