Agents, Agency and Autonomy: A Formal Computational Model

Mark Paul d’Inverno

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy of the University of London

Department of Computer Science
University College London

1998
Abstract

In recent years, there has been an explosion of interest in agents and multi-agent systems in a variety of areas including artificial intelligence, and software engineering. Agent technology, however, is still relatively young, and there is much debate and discussion over many important concepts and the relevant terminology. In particular, in a great deal of agent research, agents themselves are defined in wildly different ways, if at all, and this makes it extremely difficult to be explicit about their nature and functionality.

These problems have arisen in part due to the lack of a common structure and language for describing and reasoning about both single agents and multi-agent systems, which might facilitate a rigorous organisation of the field.

In response to this, a four-tiered theoretical formal framework for agent systems is proposed, which we use as a base from which to develop a detailed model of agents and their dimensions, the properties required by agents for effective operation, and the social organisation of multi-agent systems. This framework essentially comprises entities, objects, agents and autonomous agents, and specifies the relationships between them to provide a rigorous and detailed analysis of the structures underlying all such systems.

Key to the understanding of this work is our overarching concern as computer scientists, of building computational systems. The development of formal theories and systems as proposed above is inadequate if they are irrelevant to the needs of practitioners. While the construction of any theory or model is unavoidably somewhat removed from the realities of software development, we address this concern by ensuring that the tools used are standard software engineering ones that are accessible and support practical development.

In illustration of these ideas, and as a demonstration of the validity of the arguments made, we show how the framework and models developed can both provide a theoretical foundation and be applied directly to existing agent systems and theories: in particular, the Contract Net Protocol, AgentSpeak(L) and Social Dependence Networks.
Acknowledgements

I could not have completed this thesis without the support of my friend and colleague, Michael Luck, who has been a never-ending source of encouragement and critical evaluation. Some of the work in this thesis is based on joint work with Michael, and in particular, Chapter 3 is an extension to work developed together and presented in [102]. There have also been a series of joint papers which have resulted from collaboration [35, 36, 37, 38, 39, 40, 103] and which contain elements included in this thesis. However, the development of this work as a thesis, and the errors it contains are mine alone.

My thanks to my supervisor John Campbell who initially attracted me to the field of multi-agent systems. He has always been stimulating and encouraging, providing a dynamic cultural environment in which to develop the ideas in this thesis. I am particularly grateful for the incisive and constructive criticisms he has made during the preparation of this document.

In no particular order I would like to thank Pauline and Ray d’Inverno for their continual concern, Paul Rooney for trying to turn me from academia to theatre, Godolphin Bloggs for his technical help with Windows, Hazel Sack and Jennifer Goodwin for their proofreading, Beth d’Inverno, Natalie May, Eddie Kon, Colin Myers, Val Rooney, Paul Howells, Peter Morse, Michael Wooldridge, Dominic d’Inverno, Bill Rooney, Nick Jennings, Michael Georgeff, John Wölstencroft, Tim Norman, Giles Rooney, Anand Rao, Paul d’Inverno, David Kinny, Jon Crowcroft, Alison Rooney, Mark Priestley, George Justo, Chris Rooney, Chengqui Zhang, Derek Long, Maria Fox, Candida Horseburgh and Cristiano Castelfranchi.

This research was based on work carried out under a Science and Engineering Research Council studentship.

My thanks to the University of Westminster who have provided me with the time and opportunity to prepare this thesis.

I would also like to thank the Australia Artificial Intelligence Institute for hosting me for a month during which time I learned a great deal about their agent-based systems.

The specification contained in this thesis has been checked for type correctness using the fuzz package [155].
3.2.4 Specification Structure Diagrams ........................................... 43
3.3 Initial Concepts ........................................................................ 45
3.4 Entities ....................................................................................... 46
  3.4.1 Entity State ........................................................................... 47
  3.4.2 Entity Operations .................................................................. 47
3.5 Objects ........................................................................................ 48
  3.5.1 Object Behaviour ................................................................. 48
  3.5.2 Object State ........................................................................... 49
  3.5.3 Object Operations ................................................................. 49
3.6 Agency ......................................................................................... 50
  3.6.1 Introduction ........................................................................... 50
  3.6.2 Agent Specification ............................................................... 50
  3.6.3 Agent Perception ................................................................... 51
  3.6.4 Agent Action .......................................................................... 52
  3.6.5 Agent State ............................................................................ 52
  3.6.6 Agent Operations ................................................................. 53
3.7 Autonomy ....................................................................................... 54
  3.7.1 Introduction ........................................................................... 54
  3.7.2 Autonomous Agent Specification ......................................... 55
  3.7.3 Autonomous Agent Perception ........................................... 55
  3.7.4 Autonomous Agent Action .................................................. 56
  3.7.5 Autonomous Agent State ..................................................... 56
  3.7.6 Autonomous Agent Operations .......................................... 56
3.8 Applying the Framework: Tropistic Agents .................................. 57
  3.8.1 Tropistic Agents ................................................................. 57
  3.8.2 Reformulating Perception ................................................... 57
  3.8.3 Reformulating Action ......................................................... 58
  3.8.4 Discussion ........................................................................... 59
3.9 Summary ....................................................................................... 59
3.10 Related Work ............................................................................. 59
3.11 Conclusions ............................................................................... 62

4 Agency Relationships ......................................................................... 63
  4.1 Introduction ............................................................................... 63
2 Multi-Agent Systems ....................................................................... 64
  4.2.1 Multi-Agent System Definition .......................................... 64
  4.2.2 Multi-Agent System Specification ..................................... 65
  4.2.3 Server-Agents and Neutral-Objects .................................... 65
4.3 Goal Generation ......................................................................... 66
4.4 Goal Adoption ............................................................................. 68
  4.4.1 Goal Adoption by Neutral-Objects .................................... 69
  4.4.2 Goal Adoption by Server-Agents ....................................... 70
  4.4.3 Autonomous Goal Adoption ............................................. 72
  4.4.4 Autonomous Goal Destruction ...................................... 73
4.5 Engagement ............................................................................... 73
  4.5.1 Direct Engagement .............................................................. 73
  4.5.2 Direct Engagements in a Multi-Agent System .................... 74
  4.5.3 Engagement Chains ............................................................ 75
  4.5.4 Engagement Chains in a Multi-Agent System .................... 76
4.6 Cooperation ............................................................................... 77
  4.6.1 Cooperations in a Multi-Agent System .............................. 78
  4.6.2 Discussion and Example ..................................................... 78
4.7 The Agency Society .................................................................. 79
4.8 Agency Relations Taxonomy ...................................................... 79
  4.8.1 Direct Engagement Relation .............................................. 80
  4.8.2 Generic Engagement Relation ......................................... 80
7.6.2 Making Task Announcements ........................................ 134
7.6.3 Making Bids ....................................................... 134
7.6.4 Awarding Contracts .............................................. 135
7.6.5 Terminating Contracts ............................................ 136
7.7 Summary and Conclusions ........................................... 136

8 A Computational Architecture for AgentSpeak(L) ................. 138
  8.1 Introduction .................................................... 138
  8.2 AgentSpeak(L) .................................................. 139
  8.3 Types .......................................................... 139
    8.3.1 Beliefs .................................................... 140
    8.3.2 Events .................................................... 140
    8.3.3 Plans ..................................................... 141
    8.3.4 Intentions ................................................ 141
  8.4 AgentSpeak(L) Agents ........................................... 141
  8.5 AgentSpeak(L) Agent Operation ................................ 143
  8.6 Summary and Conclusions ...................................... 146

9 Evaluating Social Dependence Networks ............................. 147
  9.1 Introduction .................................................... 147
  9.2 Social Dependence Networks ..................................... 147
  9.3 External Descriptions .......................................... 149
  9.4 Action and Resource Autonomy .................................. 152
  9.5 Dependence Relations ........................................... 155
  9.6 Dependence Situations ......................................... 157
  9.7 Locally and Mutually Believed Dependence ....................... 159
  9.8 Summary and Conclusions ...................................... 161

10 Conclusions .......................................................... 162
  10.1 Introduction .................................................... 162
  10.2 Recapitulation ................................................ 162
    10.2.1 Generality ............................................... 163
    10.2.2 Application .............................................. 163
    10.2.3 Methodology ............................................. 164
  10.3 Contributions .................................................. 164
    10.3.1 Agent Hierarchy ......................................... 164
    10.3.2 Agency Relationships .................................... 165
    10.3.3 Agent Architectures ..................................... 165
  10.4 Limitations and Further Work .................................. 166
    10.4.1 Which Language? ......................................... 166
    10.4.2 Agent Mechanisms ....................................... 166
    10.4.3 Complexity ............................................... 166
    10.4.4 Applicability ............................................. 166
    10.4.5 Design Methodology ..................................... 167
  10.5 Concluding Remarks ........................................... 167

A Generic Z Definitions .............................................. 181
  A.1 Sets .......................................................... 181
  A.2 Relations ..................................................... 182
  A.3 Functions ..................................................... 183
  A.4 Sequences ..................................................... 183

B Schema Structure for Specifying the Sociological Agency Model 185
## List of Figures

2.1 IRMA Architecture ..................................... 21
2.2 GRATE* Implementation Structure .......................... 22
2.3 Subsumption Architecture .................................... 24
2.4 Agent Network Architecture .................................. 25
2.5 PRS Architecture ............................................ 26
2.6 TouringMachine Architecture .................................. 27
2.7 INTERRAP Architecture ...................................... 28
2.8 HEARSAY-II Architecture ..................................... 30
2.9 Network of HEARSAY-II Systems .............................. 31
2.10 Structure of an ARCHON Community ......................... 32
2.11 Functional View and Implementation Architecture of an ARCHON Agent .......... 33
2.12 CIDIM Community ........................................... 34
2.13 PLACA Interpreter ........................................... 35

3.1 Structuring Schemas ........................................ 44
3.2 Entity Hierarchy ............................................ 46
3.3 Schema Structure for Specifying the Agency Framework ............... 60
3.4 Schema Structure for Specifying Agency Framework Operation .......... 60

4.1 Modified Entity Hierarchy ..................................... 66
4.2 Goal Adoption in Neutral-Objects, Server-Agents and Autonomous Agents .......... 69
4.3 Schema Structure for Specifying Multi-Agent Systems and Goal Adoption ........... 74
4.4 Example: An Engagement Chain ................................ 75
4.5 Schema Structure for Specifying the Agency Society .................. 80
4.6 Schema Structure for Specifying the Agency Relations Taxonomy ............... 84

5.1 Alternative New Engagements ................................... 89
5.2 Schema Structure for Specifying New Engagements .......................... 91
5.3 Broken Engagements Spreading in Engagement Chains ............... 92
5.4 Schema Structure for Specifying Breaking Engagements ........................ 93
5.5 New Cooperations .............................................. 94
5.6 Schema Structure for Specifying New Cooperations .......................... 95
5.7 Leaving Cooperations ....................................... 96
5.8 Schema Structure for Specifying Leaving Cooperations ........................ 97
5.9 Agency Relations for Library Scenario I ................................ 98
5.10 Agency Relations for Library Scenario II .............................. 99
5.11 Agency Relations for Library Scenario III ......................... 100
5.12 Agency Relations for Library Scenario IV ........................... 102

6.1 Example: A Sociological Agent’s Model .......................... 123

7.1 Contract Net Protocol ....................................... 127
7.2 Example: Contract Net Configuration I .......................... 131
7.3 Example: Contract Net Configuration II ........................... 132
List of Tables

3.1 Summary of Z Notation .................................. 42
3.2 Example: Descriptions of a Robot and Cup in the Agency Framework .......... 55
6.1 Example: A Sociological Agent’s Evaluation of its Plans ................. 124
9.1 Example: An External Description ................................ 148
9.2 Original Definition of External Descriptions .......................... 149
9.3 Original Definition of Action and Resource Autonomy ................. 153
9.4 Original Formalisation of Action and Resource Autonomy ............. 153
9.5 Original Definition of Dependence Networks ........................ 155
9.6 Original Formalisation of Dependence Networks ...................... 155
9.7 Original Definition of Dependence Situations ........................ 158
Chapter 1

Introduction

1.1 Early Artificial Intelligence

Consider a household robot designed to take the routine daily drudgery out of the life of its owner by performing household chores. Such a system must function flexibly and robustly in a wide range of scenarios in a dynamic and sophisticated world; it is certainly not possible to design this robot by explicitly considering all scenarios, since they are potentially infinite. In order that the robot can provide a service to its owner it must be designed so that it exhibits what is taken to be intelligent behaviour. The problem of designing artificial systems intended to exhibit behaviours found in “naturally occurring intelligent entities” [65] is part of the field of artificial intelligence (AI).

The early years of AI saw many notable accomplishments in striving for intelligent behaviour. Whilst there was substantial progress in developing systems with such capabilities, these systems were limited in scope because they were designed only to function in idealised worlds. A typical example of such a capability is problem-solving or planning that is concerned with how to construct a sequence of actions which, when performed, achieve a given goal.

Shakey, a robot built predominantly at SRI in the 1960’s and 1970’s, was an attempt at constructing such a system [122]. It was able to move, to perceive using its sensors which included a monochrome TV camera, to perform actions using its effectors and to plan to achieve goals that were provided externally using a teletype. Shakey moved about in connected rooms containing painted blocks and wedges. It could move these objects by pushing them either to achieve a goal, or if they were in its way. Shakey modelled its world using first order predicate calculus, and planned in this domain by means of the STRIPS program which operated on the symbolic model it constructed [53]. However, these achievements were limited. The rooms were bare apart from the wedges and the blocks, the objects were painted in special ways so that they could be easily identified, and the walls were all uniformly coloured. In addition, the world was sufficiently simple for Shakey to construct a complete model of it.

Similarly, at MIT in the sixties, a vision system was built to match visual images of objects with stored information of models of objects. Again, the results were successful but limited since it was necessary for there to be very careful lighting, specific painting of the blocks, and well-chosen backgrounds. Winston’s copy-demo program provides another example of one of these early systems [167]. It could look at an arrangement of blocks, and then reconstruct it somewhere else. However, this was only possible in very specific circumstances. As soon as curved blocks were introduced or the lighting was varied, the system would fail. (These and other examples of successes and failures in the history of AI are discussed in an excellent survey by Brooks [9].) We can isolate and summarise several key problems with these systems that caused limitations in much early AI work. Each of these problems is outlined below.

Complete Knowledge – If a system is based on the assumption that it will always have complete knowledge of its environment, it will not be able to cope with real environments that cannot be modelled completely.
**Static Environments** – In general, static environments in which the only changes that can occur are the results of system actions, are not possible. Real environments will admit changes that arise through external factors, and any systems based on the static environment assumption are bound to fail.

**Guaranteed Effects** – The effects of actions cannot be guaranteed. A planner, for example, cannot therefore know with total certainty that actions always achieve their intended effects.

**Toy Worlds** – Constrained environments designed specifically to test particular systems do not scale up to real worlds. Such toy worlds were a major feature of early AI work, and solutions based on them tend to be domain specific and not generally applicable.

The assumptions on which early AI systems were based enabled some fundamental issues to be tackled, and some important research to be done. The limitations imposed by these assumptions, however, constrained the scope of the solutions that could be developed.

### 1.2 Agents

Brooks, in particular, recognised the limitations of these approaches, which are all based on the assumption that a complete model of the world can be built and manipulated [9]. The success of these systems had been made possible by devising very simple environments so that they could be easily mapped to some internal model. According to Brooks, intelligent systems can only be constructed in the context of actual participation in some world. This claim comes in two parts. First, rather than modelling their environment, intelligent systems should instead be situated in a real world that influences their behaviour in an immediate and direct way. Second, they must be grounded in their worlds by means of a physical presence. That is, they must be embodied and experience the world directly.

In contrast to the idealised environments of early AI discussed above, real environments are dynamic and continually changing, possibly as a result of other systems within them. They are typically large and complex, and it is therefore not possible to have complete or correct knowledge, resulting in uncertainty concerning the effects of actions. It is partly from these Brooksian ideas of embodiment and situatedness that the concept of an agent arises.

#### 1.2.1 Types of Agent

The notion of agents has provided a way of imbuing traditional computing systems with an extra degree of flexibility that allows them to be more resilient and robust in the face of more varied and unpredictable forms of interaction. Thus computational agents have been designed to perform in electronic and physical environments, to provide services and execute tasks, possibly by collaborating with users or with other agents. This is distinct from the traditional interaction that has prevailed between users and computer systems where the user directs execution. It is also distinct from tightly-coupled interaction that arises between different components in a single fixed architecture.

The range of applications for which agents have been constructed is large and growing fast. To illustrate this diversity of domain, we describe below some of the more commonly discussed systems.

**Software Agents** are described by Genesereth and Ketchpel as programs which can communicate using an expressive agent communication language [66]. More generally they can refer to software components that continually monitor (perceive) their environment and can respond by acting in it.

**Interface Agents** provide assistance to users working with particular applications, functioning as a ‘user-application go between’. Typically, such agents might be employed by untrained users who would not be able to initiate all necessary tasks or monitor all events [107].
Personal Assistant Agents fulfill the role often typically attributed to a personal assistant [112] and collaborate with the user in the same environment [107]. As an example, they may be responsible for organizing meetings and scheduling deadlines such as a meeting scheduling agent [95]. Such agents model users to understand their roles and priorities [71].

Believable Agents are agents specifically designed so that people will ascribe human qualities to them [1]. They are used for applications in computer games and virtual reality, and draw upon many aspects of culture such as paintings, mime and cartoons investigating the mechanisms and methods used to create the illusion of life.

Electronic Mail Agents decide the priorities of incoming mail and act according to those priorities by, for example, destroying the message or immediately alerting some user to its existence [107].

Information Agents are software agents designed to locate, retrieve and collate electronic information based on user queries. Such agents include those able to extract information from heterogeneous databases and format this information in ways specified by the user. These also include web agents designed specifically for traversing hypertext structures, such as the Ahoy agent for locating home pages on the web [144]. Indeed there are many agent systems that can be currently used on the web [54].

Teaching Agents include those developed by Selker [143] that offer users help and tutorial support for learning new programming languages. Teaching agents are adaptable agents that learn about individual users and are able to offer ‘opportunistic’ tailored advice to help them.

Examples of other application agents include conference agents [47], desktop agents [71] and market agents [116]. It should be clear from this brief review that the different categories of agents overlap in a significant way and that, in many cases, the terms are interchangeable.

As this list illustrates, agent applications tend to inhabit an electronic environment. Though this is typical, there is no reason why, for example, a mail agent could not be situated in a physical environment. However, electronic agents are much easier to implement than their physical counterparts and, as Norman points out [123], the cost, effort and expertise necessary to develop and experiment with software artifacts are relatively low. Consequently, the majority of existing agent applications are confined to the electronic rather than the physical world, though this is not an important distinction; the agent metaphor applies equally well in either domain.

It would seem that there are a vast number of different kinds of agent or names for agents. There is however, a danger here, since if any new item of software can be described as an agent then the term becomes redundant and any innovations specific to agent technology may be lost in a plethora of so-called agent systems. There is consequently a responsibility placed on researchers and developers to take care over use of the agent metaphor and also to specify agent functionality precisely. In what follows, we will use the terms agency and agenthood interchangeably to refer to the basic concept of being an agent.

1.2.2 Dimensions of Agency

If an agent embodied and situated in a dynamic environment is to be effective, then there are a number of capabilities (or equally, functionalities) that it will need. In much recent work, rather than defining what it means for something to be an agent, an agent system is often characterised according to various qualities that describe its essential functionality. We refer to all these as dimensions of agency, and survey some of them below.

Rationality is concerned with making sure agents ‘do the right thing’ [139] according to some logical model. Behaving ‘rationally’ means that agents act so as to accomplish their tasks and do not act to prevent their goals.
Autonomy is taken to mean that an agent can act without the intervention of others. Autonomy is often regarded as a relative notion; the more autonomous an agent is, the less supervision it needs. If it is not autonomous, then it may always be under the direct supervision of another system or user. Most agents fall some way between the two. A semi-autonomous agent [73] is therefore sometimes also considered. In addition, autonomy has also been used to refer to agents that maintain some control over their internal state [173].

Reflection characterises an agent able to reason about its behaviour.

Deliberation is the ability to manipulate symbolic representations of an environment. For example, Genesereth considers deliberative agents as those which, on every execution cycle, consider the external action to perform next [65]. Notice that reflection necessarily requires deliberation.

Reflexivity is the quality by which agents without a representation of the world act solely in a stimulus-response fashion. Their behaviour is typically rule-based. When the condition of a rule matches the current situation, the action part of the rule is then executed [137]. Reflexive agents are sometimes called tropistic agents [65]. Deliberative agents and reflexive agents are mutually exclusive.

Reactivity is now taken to be the ability of an agent to respond to changes in its environment within an appropriately small amount of time. The notions of reactive and reflexive agents are often confused [173] and reactive can be used to describe agents without any symbolic representations.

Pro-Activeness refers to agents having longer term agendas. As well as reacting to environmental changes such agents have goals that may not be satisfied immediately, and which will typically require planning.

Mobility can refer to properties of both software and physical agents. Software mobile agents are programs that can be dispatched from a computer and transported to a remote machine for execution [74]. Mobile agents can also refer to robots which, for example, can navigate between locations in a physical space [51].

1.2.3 Problems with Definition

While these properties illustrate the range and diversity both of the design and potential application of agents, they are inadequate for constructing a more detailed and precise analysis of the basic underlying concepts. In addition, if these dimensions characterise agents, then it would seem appropriate to define these dimensions in terms of fundamental components of agency.

However, it is now generally recognised that there is no agreement on what it is that makes something an agent, and it is standard, therefore, for many researchers to provide their own definitions. For example, Wooldridge and Jennings [173] survey the field and identify a weak notion of agency that involves autonomy or the ability to function without intervention, social ability by which agents interact with other agents, reactivity allowing agents to perceive and respond to a changing environment, and pro-activeness through which agents behave in a goal-directed fashion. They also describe a strong notion of agency, prevalent in AI which, in addition to the ‘weak’ characteristics, also uses mental components such as belief, desire, intention, knowledge and so on. This provides an indication of some of the problems which exist in that the authors need to provide two definitions. Moreover, the definitions make reference to dimensions of agency, which does not seem appropriate.

The lack of a commonly agreed definition is often highlighted as a cause for concern. For example, Connah and Wavish [20] state that the term agent has “almost as many meanings as there are instances of its use” and that this causes “considerable confusion”. Riecken understates the confusion by suggesting that the term, agent, “is a bit messy” [135]. Krogh [96], too, notes that there have been many attempts to find one central common denominator and, with a certain amount of pessimism, predicts that any such attempts will fail. However, he does comment that these definitions are technically useful even though they are usually flawed. As an example, he cites the definition of software agents by Genesereth and Ketchpel [66].
“An entity is a software agent if and only if it communicates correctly in an agent communication language such as ACL.”

Krogh argues that this definition is inappropriate for the following reasons:

- If it does not communicate ‘correctly’ then it is not an agent.
- If it does not communicate at all then it is not appropriate to ascribe agenthood to an entity.

Instead, Krogh argues that there are many situations in which we would wish to ascribe agenthood to entities that cannot communicate correctly or cannot communicate at all. Without fully elaborating, Krogh further suggests that in some cases it is appropriate to consider entities that are not computer programs as agents.

Recognising that there is no commonly accepted definition of what constitutes an agent, Krogh chooses to delineate a class of agents that have certain dimensions as described above. In particular, these are independent, selfish, interacting, heterogeneous and persistent. However, the terms are not defining in themselves and can introduce even more ambiguity since, as stated earlier, the meanings attributed to these dimensions are not themselves uniform.

To summarise, there is a distinct lack of precision and consensus in work dealing with agents and their dimensions. Consequently, there is no common currency for the notion of agenthood, or indeed for dimensions of agency. As previously noted in the discussion of agent dimensions, the terms reflexive and reactive are often confused. This is not an isolated case: terms are often used interchangeably without real regard for their significance and relevance. Another example, of this kind, is that agency is often taken to imply some degree of autonomy and the two terms are often used interchangeably [61].

In this thesis we will argue that agency and autonomy relate to very specific and distinct, though related, qualities. We will show that a precise understanding of these terms and the relationship between them is of fundamental importance in defining the nature of agent-based systems.

One issue in the design of agents we have not yet considered is the ability to function in a world containing other agents. A system in which there are several agents functioning in a coherent way is referred to as a multi-agent system [101]. However, before considering such systems we briefly review general distributed systems.

### 1.3 Multi-Agent Systems

#### 1.3.1 Distributed Systems

Many application areas demand the construction of computer systems that are distributed according to the spatial configuration of the application. These systems, by virtue of their distributed nature, can have many advantages over their centralised counterparts. Specifically, for example, they may be able to exploit parallelism, are more flexible since the organisation of individual components may be reconfigured for new situations, and can be more robust since the failure of an individual component may lead to a degradation of performance rather than complete system breakdown. In addition, software engineering concerns such as modularity are necessarily addressed because the construction of each component must be treated individually. The system may then be easier to design, implement and maintain.

Combining distributed systems with artificial intelligence techniques leads to a sub-field of AI known as distributed artificial intelligence (DAI). In many cases, a DAI method is concerned with the decomposition of a problem according to the capabilities of the intelligent components, how the activities of these components can be coordinated, and how the results of the separate activities can be integrated into a complete solution. Typically, the issue is how to construct a set of interacting intelligent components that can achieve a global system goal, by imposing some fixed architecture and pre-determined interaction mechanisms.
1.3.2 Multi-Agent Systems

A multi-agent system is a distributed system consisting of intelligent components that are agents. Generally, in a multi-agent system, there is no pre-established architecture incorporating the agents and their interactions are not pre-defined such as with processes in concurrent programs. In addition, there is no global system goal, the agents being heterogeneous with their own goals and capabilities. As a result, agents need to coordinate their activities in order to avoid duplication of effort, to avoid unwittingly hindering other agents in achieving goals, and to exploit other agents’ capabilities.

This motivates the consideration of several additional issues regarding agents below.

Agent Modelling – As discussed, an agent may need a model of its world. If this world contains agents then it may be beneficial to model these other agents.

Multi-Agent Planning – In some cases agents will share plans in order to coordinate their behaviour with each other, or plan to achieve a goal using others. If, for example, lifting tables requires two individuals then an agent will have to plan to involve at least one other agent to lift a particular table.

Social Relationships – Agents may have social relationships with other agents. For example, if one agent has performed a service for another agent, the second agent may be under an obligation to reciprocate in some way. If two agents are working together to achieve a task, the agents are typically said to be cooperating.

Interaction – Agents may interact. In a multi-agent world where interaction is not pre-defined, agents may need models of each other to decide how to interact, and to decide on their success or failure. This may impact on the social relationships between the agents.

Communication – Agents may communicate to exploit interaction and ensure coordination. An agent may persuade others to adopt its goals and alter their plans [10].

The same problems regarding the undefined nature of agents discussed earlier arise in multi-agent systems. It is difficult to consider these issues in any structured or principled way without agreement on the basic components that are involved. We argue that in order to understand fully the issues introduced above, it is first necessary to understand the nature of individual agents themselves.

1.4 Aims

Agent systems that can act independently in complex environments are very appealing and, judging by the recent effort invested in their development and application, are here to stay. However, single-agent systems are fundamentally limited by their own particular dimensions. Individual agents in multi-agent systems, by contrast, can exploit the capabilities of other agents, allowing for a much greater range of collective functionality than an isolated agent. Multi-agent systems, however, are more complex in design and construction since this increased functionality arises through the interaction of the individual agents involved. Furthermore, in many cases, these agents are autonomous and the interaction emerges in a fashion that is essentially unplanned. An understanding of the way in which such systems operate can only be achieved through an analysis of the relationships that arise between agents, and any pre-existing architectural structure.

This thesis provides a detailed examination and analysis of the general social organisation of multi-agent systems and develops models of dimensions (capabilities) of agents that need to function effectively and efficiently within them. A key argument of this thesis is that this analysis must be directed at the individual interacting agents themselves. We can itemise the aims of this thesis as follows.

- To provide principled definitions for agency and autonomy, and to explicate the nature of the relationship between them. The distinction between agency and autonomy is critical to understanding the nature of the relationships in multi-agent systems.
To provide a unifying framework that incorporates these definitions within which we can situate existing work on, and definitions of, agents. We aim to achieve this by constructing high-level models of agents and autonomous agents and their operation in a way that is not architecture-specific. This framework should serve as a foundation both for the development of agent systems and for analysing agent relationships.

To analyse and define key relationships that arise between agents. These relationships are universal and fundamental, arising as a natural consequence of the definitions of agency and autonomy.

To build models of the dimensions of deliberative agents that are required for them to recognise and exploit social relationships in order that they may interact effectively in multi-agent systems.

To demonstrate the practical applicability of the models constructed to existing theories and systems so that they may be readily analysed and evaluated.

1.4.1 Principles
In attempting to meet these aims, the work described in this thesis is guided by several underlying principles that we list below.

- This research is concerned with the development of a theory that subsumes, as far as possible, existing work, so that it is useful and widely applicable. It is not our intention to produce yet another set of definitions that do not relate to any previous attempts.

- A major criticism of many existing definitions and concepts is that they are vague and ambiguous. If this research is to avoid such problems, we must ensure that we are precise and unambiguous at all times. This work will therefore use formal methods to provide a rigorous and precise underpinning to the work in this thesis.

- Abstraction in analysis and design is an important tool because it enables an appropriate level of description to be chosen. Consequently, the work described in this thesis provides a means of moving between different levels of agent specification from the most abstract to the least abstract, from primitive definitional specifications through deliberative agent dimensions to instances of systems and theories.

- Agent theories should serve as specifications [173]. As a result of this work, we aim to provide an explicit design and specification environment for the development of real systems; our formal models will relate directly to computational systems.

1.5 Thesis Overview
The remaining chapters of this thesis describe our efforts at satisfying the aims listed above, while adhering to the principles. Chapter 2 provides a more detailed introduction to single-agent and multi-agent systems and architectures. In Chapter 3, we propose definitions for agency and autonomy and develop a unifying formal framework for reasoning about agent systems. This framework is constructed with the $Z$ specification language. Chapter 4 develops a model of how and why goals are initially generated and subsequently adopted through agent interaction. It then examines the different kinds of social relationship that arise between agents when goal adoption occurs. Chapter 5 provides an operational account of how these social relationships are created and destroyed. This chapter does not introduce new concepts but addresses issues relating to the development of real systems. Chapter 6 builds a model of the specific deliberative agent dimensions required for an agent to interact effectively with others in multi-agent environments. We then demonstrate the applicability of the work in the subsequent three chapters by relating it in a strong way to existing systems, architectures and theories. In Chapter 7, the contract net protocol, a classic example of a DAI technique, is formalised in terms
of our models. The work in Chapter 8 constructs a computational architecture for AgentSpeak(L), proposed as an agent language for capturing the operational semantics of an idealised agent system. The recent and ambitious *Social Dependence Networks* developed from *Social Power Theory*, which provides a taxonomy of dependence relationships between agents, are then assessed by applying the work developed in this thesis in Chapter 9. Finally, in Chapter 10, we evaluate the contributions of this work, discuss the conclusions that can be drawn, consider its limitations, and outline potential areas of further investigation.
Chapter 2

Agent Systems

2.1 Introduction

Recently, there has been an explosion of interest in agent-based systems and the related subfield of distributed artificial intelligence (DAI), so there is now a wealth of work in this area including a number of surveys. For example, Wooldridge and Jennings provide a comprehensive survey of theoretical and practical agent research [173], while Bond and Gasser [3] and, more recently, Moulin and Chaib-draa [115] consider systems and theories in DAI. Complementing this is a survey of software agents by Nwana [126] and Industrial Applications of DAI techniques by Parunak [128]. Detailed reviews of agent-based work can be found in these surveys. In this chapter our aim is more modest, providing a general background to agent systems by showing some of the variety of ways in which the agent metaphor has been interpreted and implemented. We identify five agent categories and survey two or three examples we consider to be representative of each. The categories are deliberative architectures, reactive architectures, hybrid architectures, distributed agent architectures, and agent-oriented programming.

In general, architectures provide information about essential data structures, relationships between these data structures, the processes or functions that operate on these data structures and the operation or execution cycle of an agent. Wooldridge and Jennings distinguish three categories of single-agent systems based on their architectures [173].

- **Deliberative Agent Systems** symbolically model their environment and manipulate these symbols in order to act.
- **Hybrid Agent Systems** can act both deliberatively and reactively.

Single-agent architectures result from what is referred to as the micro-level design perspective, where the focus is on the individual agent. The macro-level design perspective, on the other hand, describes the stance taken when dealing with the social or global dimensions of a distributed agent system. Such systems are designed from a holistic view as well as at the micro-level of individual agents. The study of the macro-level of agent design is used in the construction of distributed agent systems.

Rather than defining an architecture, Agent Oriented Programming is a paradigm for directly programming the behaviour of agents using computational languages whose semantics capture some theory of rational agency [145]. Typically agents have an initial set of beliefs, goals and plans and an interpreter that details how agents should achieve their goals given an environmental context.
2.1.1 Overview

In this chapter, the reviews are not detailed, nor do we have any specific goal of highlighting universal agent issues. Instead, we provide overviews of example systems and highlight the most relevant aspects or criticisms of them that have been made in the literature. Sections 2.2, 2.3 and 2.4 consider examples of deliberative, reactive and hybrid single-agent architectures respectively, all designed at the micro-level. Distributed agent architectures are described in Section 2.5. These are systems that include a macro-level design perspective; collections of agents which interact in some pre-specified way to fulfill the global goals of the system as well as their own local goals. Section 2.6 considers examples of Agent Oriented Programming. The final section discusses issues relating to the construction of agent systems in general, presents some conclusions about the current state of agent technology, and finally motivates the approach taken in that thesis.

2.2 Deliberative Agent Architectures

Any agent system able to maintain and manipulate structures of symbols that model the world is a deliberative agent [173]. In order to model rational or intentional agency, an abstraction level is chosen for the symbols such that they represent mental attitudes. Most agent systems include a deliberative architecture to support deliberative reasoning at the mental-attitude level.

Mental attitudes used to describe and characterise the behaviour of agents include beliefs, goals, assumptions, desires, knowledge, plans, motivations and intentions, and are commonly grouped into three categories, informative, motivational and deliberative [94]. The first refers to that which a system considers to be true about the world and includes knowledge, beliefs and assumptions, the second to the ‘wants’ of a system including goals, desires and motivations, and the third concerns how an agent’s behaviour is directed and includes plans and intentions. The distinction between the second and third categories is subtle since it is possible that a system may desire a certain state without planning for it, or intending it to happen.

There are several compelling reasons why agents defined using mental attitudes might be useful. First, if an agent can be described in terms of what it knows, what it wants and what it intends then, since it is modelled on familiar concepts, it becomes possible for users to understand and predict its behaviour. Second, understanding the relationship between these different attitudes and how they affect behaviour could provide the control mechanism for ‘intelligent action’ in general. Third, computational agents designed in this way may be able to interpret the behaviour of others independently of any implementation.

2.2.1 BDI Architecture

Although not a system, the belief-desire-intention (BDI) model provides a foundation for many systems and we therefore consider it separately as an abstract architecture in its own right. Arguably, the most successful agent architectures are based on the BDI model, in which agents continually monitor their environments and act to change them, based on the three mental attitudes of belief, desire and intention, representing informational, motivational and decision-making capabilities. Architectures based on the BDI model explicitly represent beliefs, desires and intentions as data structures, which determine the operation of the agent.

Many formal models have been devised in order to investigate and analyse the relationships between belief, desire and intention. Much of this work investigates how modal and temporal logics can be used to specify theories of agent behaviour. For example, Cohen and Levesque [17], Kinny et al. [93], Jennings [86], Haddadi [72], Wooldridge [171] and Rao [134], have all proposed such models. These models are often underpinned using the possible-worlds semantics, first proposed by Hintikka [79]. Using this as a basis, it is possible to develop BDI modal logics that determine how different relationships between the mental attitudes produce different agent behaviour.

As well as the formal models of BDI agents that specify desired behaviours of BDI agents, generic BDI architectures that encapsulate the BDI model have been constructed to support the design of practical agents [56], and programming languages have been proposed that can be used to implement BDI agents that satisfy some of the formal BDI properties [132]. In addition, BDI agents have been used
extensively in real-world applications [67]. Many of the architectures and systems that are discussed in this chapter have been influenced by models of BDI, including the Intelligent Resource-bounded Machine Architecture (IRMA) [7] and GRATE* [87]. Both these agent systems are categorised by Wooldridge and Jennings as deliberative [173], and are considered next.

2.2.2 IRMA

In considering the problem of enabling agents to perform means-ends reasoning, to weigh up competing alternative plans of action and to cope with interaction between these two forms of reasoning in a resource-bounded way, Bratman proposed a high-level architecture, subsequently implemented as the Intelligent Resource-bounded Machine Architecture or IRMA [7]. This architecture, based on the BDI model, recognises that agents will always be computationally limited or resource-bounded, and that sub-optimal decisions may have to be made.

As with other BDI architectures, it explicitly represents the beliefs, desires and intentions of the agent as data structures in the system. These are shown as ovals in Figure 2.1, which includes another data structure, the plan library, that maintains a repository of plans and their preconditions. In IRMA, once a plan has been selected for execution, it becomes an active plan, and therefore an intention.

IRMA’s reasoning cycle is based around five key processes, comprising the opportunity analyser, the means-end reasoner, the compatibility filter, the filter override mechanism and the deliberation process, described below.

- Initially, IRMA perceives its environment and updates its beliefs.
- Then, using the opportunity analyser it must check whether its current goals have been inadvertently achieved, whether existing intentions can no longer be realised, or whether new alternative plans can be proposed.
- Goals are achieved by presenting them to the means-end reasoner, which attempts to generate new sub-plans.
Plans from both the opportunity analyser and the means-end reasoner are passed to the filtering process made up of the compatibility filter and the filter override mechanism, which run in parallel.

The role of the compatibility filter is to ascertain whether new plans are consistent with the existing intentions of the agent. Typically, incompatible plans are discarded but, in some cases, if they satisfy certain properties, can be passed to the deliberation process by the filter override mechanism instead. When the override mechanism intervenes in this way, existing incompatible intentions are suspended.

Next the deliberation process determines how the plans it is passed after filtering affect the current intention structure. This intention structure simply comprises plans that have been selected for execution, and may be updated to accommodate new plans, when appropriate.

IRMA has been demonstrated and tested in the Tileworld simulation environment [130], which showed that given appropriate overriding mechanisms, committing to plans by means of intentions is a useful strategy for effective agent behaviour in a dynamic environment.

2.2.3 GRATE*

There are strong philosophical arguments to suggest that notions of collective or group intention cannot be analysed as the summation of individual intentional behaviour [142]. Adopting this view, Jennings proposes a model of joint responsibility [86], which aims to capture some intuitive notions fundamental to distributed problem-solving, and proposed the GRATE* architecture [89] as a means of implementing this model. The architecture contains beliefs, desires, intentions and joint-intentions as the data primitives, since in the joint-responsibility model, joint-intentions are irreducible to single intentions. The joint-intentions control group problem-solving behaviour specifying, for example, preconditions that agents must satisfy to be involved in some group activity.

GRATE* comprises both a functional architecture describing the computational processes necessary to support joint-responsibility as well as an implementation architecture to provide an environment for the construction of agents in general. The implementation architecture is displayed in Figure 2.2, taken from [89]. Agents essentially comprise two components, a cooperation and control
layer, and a domain level system. The former is a meta-level controller that operates directly on the domain level system coordinating its activity (which it views as a set of tasks), with other agents.

The cooperation layer comprises three main generic problem-solving modules. These are generic because they do not need to be tailored in any way for specific applications. First, the control module manages the interactions with the domain level system. Second, the situation assessment module decides whether to perform activities locally, whether to delegate activities or whether to cooperate with other agents. Lastly, the cooperation module controls the agent’s social activity to enact decisions taken by the situation assessment module such as how to respond to requests for cooperation. In addition, there are three data components: the information store contains both information communicated by other agents and from the original domain level system itself, the acquaintance models contain information about others, and the self-model contains information about the domain level system. Jennings reports that this system has been tested in several experiments against a collection of interacting agents not able to form groups with favourable results [89]. The GRATE* architecture is also used as a generic agent architecture for a development method for multi-agent systems known as ARCHON, which has been applied in industrial systems. This system is discussed in Section 2.5.

2.3 Reactive Agent Architectures

As discussed in the previous chapter, some authors have argued that intelligence does not necessarily require symbolic representation and manipulation. This view is strengthened by the problems that have emerged in mainstream artificial intelligence such as the complexity and, in some cases, intractability, of some symbolic manipulation problems such as planning. Arguing for systems situated and embodied in the real world making timely responses to events, architectures were proposed where agent behaviour is directly coupled with the world. These architectures typically incorporate stimulus-response rules; the environment provides a stimulus that causes a rule to fire and the agent responds with some behaviour.

Agents that do not maintain a symbolic representation of their environment are known as reactive agents, and corresponding architectures are referred to as reactive architectures. The main proponent of reactive systems was Brooks who conceived the subsumption architecture for controlling the behaviour of a robot. In addition, Maes, a former colleague of Brooks, developed the agent network architecture, built upon the principle that intelligent behaviour is an emergent phenomenon arising from the interaction of “societies of non-intelligent systems”. Both architectures are considered here.

2.3.1 Subsumption Architecture

Brooks proposed the subsumption architecture as a means of controlling the behaviour of a mobile robot in real time [8] without using a central control mechanism. The architecture comprises eight task-achieving behaviours, each of which is implemented separately and arranged as shown in Figure 2.3, which is based on a diagram originally presented in [8]. The hierarchy of layers reflects how specific the behaviour is; the more specific the task, the higher the level. In the case of the mobile robot, there are eight levels from 0 to 7 which relate to contact avoidance, wandering, exploring, building maps, noticing change, distinguishing objects, changing the world according to goals, and reasoning about the behaviour of others.

The first step in the agent’s construction is to build the 0th control level and, once this has been tested, to build the 1st control level on the 0th level. The 1st level has access to the data in level 0 and can also supply its own inputs to this layer to suppress the normal activity of the 0th layer. The 0th level continues to execute, unaware that there is a higher level intermittently influencing its behaviour. This process is then repeated for each successive layer. Subsequently, each layer competes to control the behaviour of the robot.

Brooks argues that using layers in this way enables the control system to function at a very early stage in system development, and that higher levels can be added as required without having to change the existing lower-level architecture. This design strategy has had a significant impact on agent architectures in general, forming the basis for many hybrid architectures such as INTERRAP [120] and TouringMachines [51], which also contain layers representing the world at different levels of
abstraction that compete for control over the agent’s actuators. However, by itself, the subsumption architecture does not support any explicit reasoning and it is not clear how it could be scaled up to domains requiring more sophisticated problem-solving behaviour.

2.3.2 Agent Network Architecture

Building on the ideas of the subsumption architecture and those of Minsky [111], who has argued that intelligent systems consist of societies of mindless interacting systems. Maes developed the agent network architecture (ANA) [106]. The ANA consists of a collection of competence modules, each of which competes to control behaviour. This competition is dependent on the activation of a module which is dependent on both external and internal considerations. The former includes whether a module is executable, the agent’s perception of the environment and the current goals of the agent. However, the novelty of this architecture is in the internal mechanism whereby modules affect the activation of other modules by links between them. Activated modules increase the activation of their successors, non-executable models activate their predecessors, and all modules decrease the activation of their conflictors.

One illustrative example of the ANA mechanism can be seen in Figure 2.4, taken from [106], which models the behaviour of an agent in drinking from a cup. This agent has two permanent goals: to relieve its thirst and to be polite, the latter requiring the agent to move the cup to its mouth rather than its mouth to the cup. If the agent is thirsty in a situation where the data observed includes the presence of a cup, the modules drink and recognise-cup are activated. Once the external activation is calculated the internal activation is considered and, since the recognise-cup module is currently executable, activation spreads from this module to its successors, from all other modules to their predecessors, and from every module to all their conflictors. Finally, the activation of each module is inspected and, if one surpasses a threshold value, it becomes activated and controls behaviour.

Maes argues that such an architecture is attractive because it is distributed, modular, and robust. However, whilst it represents a novel approach for constructing autonomous agents, and does provide some evidence that cognitive functions can be implemented in terms of simple behaviours, it is difficult to see, as with the subsumption architecture, how it could be extended to achieve more rational behaviour or to provide a paradigm for designing agent-based applications in general.
2.4 Hybrid Agent Architectures

It is now generally recognised that if agent systems are to survive in real and complex environments they need to be reactive in order to respond to environmental changes sufficiently quickly and be deliberative in order to achieve current complex goals without deleteriously affecting longer term options [52]. If environments change rapidly or unexpectedly, agents may need to act in a reactive manner, whereas more stable environments may allow agents time to deliberate on the best course of action. Architectures containing both deliberative and reactive components are called hybrid and include the Procedural Reasoning System (PRS) [69], TouringMachines [51] and INTERRAP [119], which we consider here.

2.4.1 Procedural Reasoning System

The PRS architecture is directly inspired by the BDI model of agents and can be seen in Figure 2.5, taken from [68]. Beliefs, desires and intentions of PRS are represented explicitly in the Database, Goalbase and Intention Structure, respectively. PRS agents continually respond to perceptions, called events which represent new beliefs and goals, by selecting plans to become intentions.

Plans, sometimes called Knowledge Areas for historical reasons, are designed by a programmer before run-time to capture the procedural knowledge of the agent as recipes detailing courses of action. Plans are the main feature of PRS systems and comprise several components which we describe here. The trigger or invocation specifies that which the agent must perceive in order for a plan to become a contender for execution selection. In addition, the plan’s context must be believed by the agent before the plan can be adopted for execution. The plan body captures procedural knowledge as an OR-tree with arcs labelled with formulas which may be primitive external actions such as C++ function calls, internal actions such as adding a fact to its database, query goals that are matched to the agent’s beliefs or, finally, achieve goals which generate new subgoals.

At the start of execution agents are initialised with a set of plans, goals, beliefs, an empty event buffer and no intentions. The operation of the agent is then as follows.

Figure 2.4: Agent Network Architecture
1. Perceive the world, and update the set of events.

2. For each event, generate the set of plans whose trigger condition matches the event. These are known as the relevant plans of an event.

3. For each event, select the subset of relevant plans whose context condition is satisfied by the agent’s current beliefs. These plans are known as active plans.

4. From the set of active plans, select one for execution so that it is now an intention.

5. Include this new intention in the current intention structure either by creating a new intention stack or placing it on the top of an existing stack.

6. Select an intention stack, take the topmost intention and execute the next formula in it.

The Procedural Reasoning System (PRS) was originally developed by Georgeff, Lansky and Ingrand as a prototype Lisp system [69] and has since been developed into a C++ implementation referred to as the distributed Multi-Agent Reasoning System (dMARS) [34]. Kinny and Georgeff [91] have used the Tileworld simulation [130] to show that for resource-bounded agents, this reactive meta-level control was critical in complex and dynamic environments. It is perhaps the most successful and widely applied architecture of all agent systems having been used in developing many applications such as air traffic control, spacecraft systems, telecommunications management and air-combat modelling [67].

2.4.2 TouringMachines

Ferguson proposes the TouringMachine as an architecture for controlling and coordinating the actions of autonomous agents situated in dynamic multi-agent worlds. The TouringMachine architecture is illustrated in Figure 2.6, which is taken from [52]. It is similar to the subsumption architecture in that it consists of a number of layers, the reactive layer, planning layer and modelling layer, which continually compete to control the agent’s behaviour. The reactive layer responds quickly to events not explicitly programmed in the other layers such as when a new agent or obstacle is perceived.
Generating, executing and modifying plans such as constructing a route in order to move to a target destination, is the responsibility of the planning layer. Finally, the modelling layer is used for building and maintaining models of entities in the environment (including the agent itself), which are used to understand the current behaviours of others and make predictions about their future behaviours.

Each layer models the agent’s world at a different level of abstraction but, unlike the subsumption architecture, each is directly connected to both the action and perception of the agent, and any two layers can communicate with each other. Since these layers are each modelled at different abstraction levels, they each have an incomplete view of the world encoding different strategies and tactics for responding to events. Conflicts over action-selection, therefore, inevitably arise and so the architecture includes a set of global context-dependent control rules to achieve coherence between these layers, which work by suppressing either the input to, or the output from, a layer.

Ferguson argues that this architecture can exhibit a wide variety of behaviours from reactive through to goal-directed, reflective and predictive. Using a testbed known as the TouringWorld, Ferguson has shown how this architecture can be used to build agents that can achieve multiple goals in dynamic multi-agent worlds.

### 2.4.3 INTERRRAP

Müller et al. [120] argue that layered hybrid architectures are beneficial since they support the modelling of an agent’s environment at different levels of abstraction, the different times taken to respond to an action (responsiveness), and the level of knowledge and reasoning sophistication required by an agent. They categorise Ferguson’s TouringMachine architecture described above as horizontally layered architecture — each layer can interact with every other layer as well as with the perceptions and actions. In general, many communication paths have to be considered and any centralised control will therefore need to be sophisticated.

In response, the authors propose vertically layered architectures, where layers are configured similarly, but where communication is only possible between adjacent layers. Clearly, fewer communication channels have to be designed. Consider a system with \( n \) layers, for example. If each layer can only communicate with adjacent layers in both directions, at most \( 2(n - 1) \) channels will be required. In a horizontal architecture, the first layer alone requires \( 2(n - 1) \) paths to communicate in both di-
rections with every other layer, the next layer needs \(2(n - 2)\) paths to ensure communication with all succeeding layers and so on. An \(n\)-layered horizontal architecture may therefore require anything up to \(2n!\) communication paths. The disadvantage of vertically layered architectures is that for a layer to control behaviour, the control must necessarily pass through all lower levels.

The INTERRRAP agent architecture [56] is an example of a vertically layered architecture and is shown in Figure 2.7, which is taken from [120]. This architecture comprises four layers that can communicate in both directions with adjacent layers. INTERRRAP was developed to model resource-bounded autonomous agents that interact with others in dynamic environments, whilst implementing a pragmatic BDI architecture. The mental state of the agent is contained in three different hierarchical layers known as the behaviour-based layer, the plan-based layer and the cooperation-based layer. Whilst the behaviour-based layer is always carefully modelled for specific domains, the others contain more generic information pertaining to goal-directed and social behaviour.

The agent’s knowledge-base is correspondingly split into three layers. At the lowest level, the agent’s world model comprises beliefs about the environment. At the next level, the mental model contains knowledge about the agent itself including its goals, plans and intentions. The social model is the highest level containing information about others and the current state of joint-plans, joint-goals and joint-intentions. These three knowledge bases are respectively accessed and updated by the behaviour-based component, plan-based component and cooperation component, respectively. The lowest level allows agents to react to unforeseen events and to quickly execute routine tasks without any explicit symbolic manipulation. The plan-based layer allows non-social goal-directed behaviour. Plans are either hierarchical templates that call other plans, or directly executable behaviours. Finally, the social level enables the agent to interact with others by coordinating actions and forming joint-plans.

The essential operation of the agent is simple; in response to events in the environment, control spreads upwards until the appropriate level is reached. The authors report that the benefits of decentralising knowledge and behaviour into layers must be weighed against the communication between and coordination of these layers, and argue that this is analogous to problems determining the number of agents required for building multi-agent applications. Extending this analogy — that the design
of layered architectures is analogous to designing multi-agent systems by treating each layer as an autonomous agent — the contract net protocol is proposed as a useful mechanism for coordinating the concurrent execution of symbolic reasoning at different layers. (We discuss the contract net protocol in Section 2.5.)

2.5 Distributed Agent Architectures

The architectures described above may enable agents to interact with other agents. However, the design of these systems is concerned with the individual agents and the dimensions required to interact effectively with others. The macro-level of modelling, on the other hand, considers a multi-agent system from an omniscient or holistic view where interaction, coordination and cooperation between the agents is designed in advance of run-time. The concern here is with the global system structure and mechanisms for interaction, including agent communication protocols, to enable effective coordination [128].

In this section we describe one of the precursors to mainstream multi-agent systems, which uses a blackboard architecture known as HEARSAY-II [50] to build a distributed problem-solving system. One of the most employed DAI mechanisms for controlling the problem-solving behaviour of a collection of distributed agents is the contract net protocol [152], which is concerned with how agents can dynamically configure in order to form contracts. ARCHON, on the other hand, provides a methodology for constructing industrial DAI systems, combining both micro and macro elements of multi-agent design. These three examples are described here.

2.5.1 HEARSAY-II

HEARSAY-II was developed in the seventies at Carnegie Mellon University [50] and is a problem-solving organisation that attempts to effectively exploit a multiprocessor system. An overall solution is constructed by the aggregation of partial solutions in situations where input data contains incomplete and incorrect knowledge.

HEARSAY-II architectures comprise three main data structures: the blackboard, focus of control and scheduling queue, as illustrated in Figure 2.8, taken from [100]. The blackboard is a shared database and has different layers each representing the problem space in different ways and at different levels of abstraction. Packets of information are called hypotheses, which can be represented at different abstraction levels and are connected using a graph structure. The top level represents the highest-level of abstraction so that the information at any level is an abstraction of elements at the next lower level.

Each different area of knowledge is represented by a knowledge source (KS), which can access certain levels in the blackboard to read, write, modify and remove hypotheses. Every KS has a set of patterns and associated actions; if a pattern matches some current hypothesis, a KS activation is created and placed in a scheduling queue and may be subsequently selected by the scheduler. The scheduling queue contains those KS activations that have yet to be performed, ordered according to priority recorded by the focus-of-control, which contains meta-domain information about the system’s current problem-solving activity. This information is updated by the blackboard monitor, which also informs individual KSs of changes to any hypotheses that they have declared to be relevant.

Subsequently, this knowledge-based system formed the basis for a distributed problem-solving system without requiring extensive modification. The most important requirement for the distributed system was to minimise the ratio of inter-node communication to individual node processing. To facilitate each node being able to function without inter-node communication, each comprised a complete HEARSAY-II architecture. Further, in order to achieve a distributed set of HEARSAY-II systems, each node was augmented with a Receive-KS and Transmit-KS. The former receives information from other nodes and determines when to write it to the blackboard and the latter evaluates which information to communicate to others.

Figure 2.9 provides a schematic diagram of three centralised HEARSAY-II systems combined into a distributed system, taken from [100]. The resulting system was one of the first to deal with the issues of distributed problem solvers in general, where each node has a local, incomplete and inconsistent
view of the world. At the time, it represented a new method for designing distributed systems since existing methods typically involved taking a centralised approach and augmenting it to cope with the uncertainty inherent in distribution. This distributed system is one of the first that could be called a multi-agent system since each node had a significant degree of problem-solving autonomy.

### 2.5.2 Contract Net Protocol

In the example of a distributed problem solver described above, the relationships between nodes are static, representing an unalterable structure. The contract net protocol, proposed by Smith and Davis [152], provides a mechanism where nodes can *dynamically* create relationships in response to the current processing requirements of the system as a whole thereby enabling *opportunistic task allocation*.

In the contract net framework, a node with a task to be achieved forms *contracts* with others who proceed to accomplish this task. The steps to forming and maintaining a contract are defined by the *contract net protocol*, described as follows.

1. A node with a task decomposes it into a number of subtasks. For each subtask the node makes a *task announcement* describing what needs to be performed along with *eligibility requirements* detailing the necessary processing requirements to be able to complete it.

2. Once a node receives a task announcement, if it fulfils the eligibility specification, it may *bid* for the task. A node will not bid unless it is free to perform the task. The potential manager receives a set of bids and ranks them according to criteria associated with the task.

3. Once all bids have been received by the potential manager node it *awards* a contract to the bidder with the highest ranked bid.

4. There is now a *contract* between the node (called the *manager*) who made the task announcement and the bidder (called the *contractor*) with the highest ranked bid. The manager monitors the problem-solving of the contractor, requesting progress reports or cancelling the contract if necessary. The manager then integrates partial results from completed contracts to provide a complete solution to the original task.
Figure 2.9: Network of HEARSAY-II Systems
Bond and Gasser [3] list a number of innovations of the contract net protocol including the design of a new style of communication protocol called negotiation, workers and managers integrating their efforts by mutual selection, system control arising through local contracts between nodes, and a distinction between task and result sharing. However, the contract net protocol does not provide any strategies for global coherence and is only suited to situations where tasks are easily decomposable into independent sub-tasks. In addition, it is only suitable if agents are not in conflict, since it does not support any mechanism for bargaining or other forms of what Müller [117] refers to as ‘negotiation’. Consequently, he prefers to call the contract net protocol a “standardised coordination method” rather than a negotiation principle.

### 2.5.3 ARCHON

ARCHON (Architecture for Cooperative Heterogeneous ON-line systems) is a development environment for building cooperating heterogeneous systems [28] incorporating a generic agent architecture based on GRATE* discussed in Section 2.2. It is built using C++ and an object oriented variant of Common Lisp and has been widely applied in many industrial contexts such as electricity management and particle accelerator control [22, 85, 129]. Arguably, one of the reasons for ARCHON’s industrial uptake is that its application enables legacy systems to be incorporated into the ensuing DAI system. This is achieved by ‘agentifying’ existing components so that they can interact in a multi-agent context, communicating with other agents to enhance their individual problem-solving potential. A further reason is that ARCHON provides not only a generic agent architecture but also a methodology for constructing DAI systems.

Agents in ARCHON comprise three layers known as the intelligent system (IS), the ARCHON layer (AL), and an interface between them called the AL-IS interface, shown in Figure 2.10. This diagram and the others describing ARCHON are taken from [16]. The IS incorporates the domain-level problem-solving capabilities of the agent, which may be a component of the legacy system. The AL, which incorporates the generic agent architecture, represents the social capabilities of the agent such as its ability to plan, communicate and model others. It controls the tasks of the IS, and decides when to interact with other agents. Since the ISs of the distributed system may be heterogeneous, the AL-IS interface is used so that each IS can be treated homogeneously by an ARCHON layer. Specifically, each AL views its IS in a functional way as a collection of tasks that can be invoked to return results. These agents are then configured according to an overall design strategy and may communicate with each other by messages along a communication link as shown in Figure 2.10.

The basic functional architecture of an ARCHON agent can be seen in Figure 2.11. It has local control over its IS, a decision making component, models of itself and others, and inter-agent communication capabilities. This functional view is modelled as a four module GRATE*-like implementation structure, each module incorporating one of the components of the functional representation. They are called the Monitor, Planning and Communication Module (PCM), High-level Communication Module (HLCM) and Agent Management Module (AIM), each considered below.
The monitor represents each individual IS task as a *monitoring unit* (MU). Monitoring units are represented in *OR-plans* where nodes are MUs and arcs are conditions. These plans are subsequently used to build *behaviours*, which are similar to PRS plans described earlier in Section 2.4. Each behaviour contains a plan body, trigger condition and related ‘children behaviours’ thus enabling behaviours to be incorporated into other behaviours facilitating a modular, hierarchical design. The AIM contains a *self-model* and a set of *acquaintance models*. The self-model contains the definitions of the MUs, plans and behaviours, and the current interests and workload of the (local) agent. The acquaintance models of a local agent provide descriptions of those agents who could either perform tasks for the local agent or who might be interested in the results of tasks completed by the local agent. The PCM determines when to exploit others’ capabilities and when to make contributions to the agent community based on the AIM module. The HLCM enables agents to communicate.

The ARCHON methodology incorporates both a *top-down* and a *bottom-up* approach. The former, influenced by classical software-engineering approaches, concerns describing overall system goals, modularising these into sub-goals and, subsequently, identifying the required sub-system components and data-flows between them. The latter approach determines how legacy components constrain the top-down approach by identifying redundancy and isolating additional functionality required at the domain level.

One example of a system developed using the ARCHON approach is CIDIM (*cooperating systems for distribution management systems*) which aids human control engineers responsible for ensuring the continuous supply of electricity over a grid. In this application, the bottom-up approach forced examination of the three pre-existing stand-alone systems known as the *High-Voltage Expert System* (HVES), the *Switching Schedule Program* (SSPA) and a weather watch system that supplied information about lightning strikes to the controller. The top-down approach identified several high-level goals (such as restoring power after breakdowns) and associated tasks (such as fault diagnosis), which resulted in developing additional ISs. The existing stand-alone systems were agentified into the *Low-voltage expert system* (LVES), the *switch checking system* (SCS) and a *security analysis system*. Additional ISs identified by the method became the *Telemetry* and *Information* agents, all shown in Figure 2.12. Cockburn and Jennings [16] report that ARCHON and components of the CIDIM system are now being offered to regional electricity companies.
2.6 Agent Oriented Programming

In the examples so far, the focus has been on agent architectures to support the design of agent-based systems. Another approach to building agents is to design a programming language whose denotational and operational semantics are based on some theory of rational or intentional agency, such as the BDI model, and to program the desired behaviour of individual agents directly using mental attitudes. Such a technique is referred to as Agent Oriented Programming. An agent oriented program is a set of transition rules that specify how an agent in a given mental state will respond to an input, which may be a set of messages from other agents, by defining its new mental state and any outputs. The languages considered in this section are PLACA, which extends the expressive power of AGENT0 [145] developed by Shoham who first introduced the concept of agent oriented programs, Concurrent METATEM, which consists of directly executable temporal logic formulae [58], and AgentSpeak(L), which formalises the operational semantics of a simplified PRS [132] system.

2.6.1 PLACA

A program in PLACA is defined by a consistent, initial mental state and a set of mental-state rules specifying how the mental state changes in various scenarios. An agent's state consists of capabilities, beliefs, intentions and plans (which are initially empty). At every step, an agent collects messages from the input buffer that have been received from others, clears the buffer and updates its mental state according to its defining program.

An overview of the PLACA interpreter can be seen in Figure 2.13, taken from [161]. At the beginning of each execution step, the Mental-Change Rule Checker identifies those transition rules that are satisfied at the current state and the Mental-Change Rule Applier applies those rules to the current mental state and messages collected from the Input Buffer. Once the mental state is updated, messages that need to be sent are placed in the Output Buffer, and actions that need to be performed are recorded and executed in the next step. If there is sufficient time before the next tick of the clock, the planner may construct and refine current plans for satisfying intentions.

PLACA is based on a formal specification of the relationship between the mental state components using a modal logic. It is asserted that PLACA agents must satisfy all the axioms that this provides. However, as Wooldridge and Jennings [173], and Rao [132] note, the relationship between the logic and the programming language is not well-defined. The authors do not justify how the data structures defining the agent’s mental state capture the model-theoretic semantics of beliefs, commitments and
2.6.2 Concurrent MetaTEm

Concurrent MetaTEm [58] is an agent programming language where each agent is defined as a concurrently executing process, communicating with other such processes via message-passing. In this language, agents are specified using a temporal logic that is directly executable. To directly execute a specification written in a temporal logic formula, the interpreter attempts to construct a model structure where the agent specification is satisfied. However, since the environment is dynamic, the environment may constantly be altering that model, so responses have to be made to what has just become true or not true. The way in which the model is constructed in a dynamic environment is referred to as the execution strategy. The execution of a Concurrent MetaTEm program produces a sequence of temporally ordered states, each labelled with a model structure stating those propositions that are true. This trace provides Concurrent MetaTEm specifications with a concrete computational interpretation.

The basic form of a specification in Concurrent MetaTEm is “on the basis of what has happened in the past do something in the future”. Each agent is specified by a set of rules of the form past; ⇒ future; , and at each cycle, the precedent of each rule is matched against an internal history, firing if a match can be found. Once a rule fires, the agent is committed to the antecedent which typically involves trying to make some predicate true.

As an example, consider the following Concurrent MetaTEm program taken from [172], specifying the behaviour of a controller agent solely responsible for supplying an infinitely renewable resource. Whilst the resource cannot be used by two agents at the same time, it is possible that the controller may be asked by two different agents for the resource simultaneously. The predicates ask(\(x\)) and give(\(x\)) mean that agent \(x\) has been asked for the resource, and that agent \(x\) has been given the resource, respectively. In temporal logic, the formula, \(\mathcal{G}Form\), is satisfied at present if \(Form\) is satisfied at the next time moment, \(\mathcal{F}Form\) is satisfied at present if \(Form\) is true at some time in the future, and \(F \mathcal{Z} G\) is satisfied if \(F\) is true since \(G\) has been true.

The specification of the controller agent is defined in Concurrent MetaTEm as follows.

---

Capabilities. The programming language is therefore only loosely based on the formal description and not underpinned by a well-defined semantics.
1. \( \Diamond \text{ask}(x) \Rightarrow \Diamond \text{give}(x) \)

2. \( (\neg \text{ask}(x) \land \exists (\text{give}(x) \land \neg \text{ask}(x))) \Rightarrow \neg \text{give}(x) \)

3. \( \text{give}(x) \land \text{give}(y) \Rightarrow (x = y) \)

These three formula can be interpreted in English as follows.

1. If an agent asks then eventually give the resource to that agent.
2. Do not give to anyone unless they have asked since you last gave to them.
3. If you give to two people then they must be the same person.

Concurrent METATEM is attractive because it is directly executable and so no time-consuming and error-prone refinement is required from specification to implementation. In addition, specifications have a concrete computational semantics defined by the sequence of models that arise through the constant interplay of the dynamic environment and the action of the agent. However, since the execution of Concurrent METATEM is based on theorem proving, some standard problems arise such as the undecidability of first order logic and the complexity involved in simple propositional logic. In addition, it is not clear how other modal operators such as belief, desire and intention could be incorporated into the language whilst maintaining its directly executable property.

Whilst further work is clearly useful in establishing when such techniques are applicable it is doubtful whether there will be a massive improvement in the sophistication of such languages in anything less than the long term.

2.6.3 AgentSpeak(L)

Rao’s AgentSpeak(L) [132] is an attempt to provide an operational and proof-theoretic semantics for a language that can be viewed as an abstraction of an implemented BDI system such as, for example, PRS, GRATE* and INTERRRAP considered earlier. It is a language based on a simplified PRS system. However, it loses none of the expressive power of PRS, since the PRS constructs not included in AgentSpeak(L) are simply designed to make programming tasks more efficient.

Agents have beliefs (about themselves, others and the environment) and intentions, which are sequences of plans (called intended means). Each agent has a library of plans, and each plan comprises a trigger, context and body, which is a sequence of actions and goals (collectively called formulas).

The basic operation an AgentSpeak(L) Agent is similar to PRS and has two aspects. The first concerns agents responding to internal events (new subgoals) and external events (new beliefs) by selecting appropriate plans. If the event queue is non-empty, an event is selected and those plans whose trigger matches the event are identified as relevant plans. Those relevant plans whose context is satisfied by the current beliefs become the active plans. From this set, one plan, called the intended means, is selected non-deterministically. Now, if the event that generated the intended means is external, the intended means creates a new intention. If the event is internal then the intended means is pushed onto the intention whose top plan’s current formula is the subgoal that generated the event.

The second aspect of the agent’s operation is the execution of intentions. First, an intention is selected, and the next formula in the top plan of that intention is evaluated. If the formula is an action it is placed in a buffer. If the formula is a query goal, and if it is satisfiable with respect to the beliefs using a substitution, the substitution is applied to the rest of the executing plan. Alternatively, if the formula is an achieve goal then a new goal event is added to the set of events to be processed.

The AgentSpeak(L) language is a significant attempt to unite the theory and practice of BDI agents. However, there are a number of problems. First, at the time of writing, a one-to-one correspondence between the operational semantics, proof-theoretic semantics and abstract interpreter has not been shown. Second, no investigation has been undertaken to show that agents specified in this abstract programming language can actually be executed. In particular, many cases, such as what
happens when actions fail, or query goals are not satisfiable, are not considered. Third, the link between mental attitudes as data structures in implemented systems (as modelled by AgentSpeak(L)), and as modal operators in the theoretical model is weak. Rao simply suggests that we, as designers, should ascribe these theoretical notions to implemented agents, rather than suggesting that there is an approach to defining a formal relationship between them.

2.7 Discussion

Agent designers recognise that agents require both deliberative and reactive architectural components in order to act effectively and efficiently in dynamic and uncertain environments with multiple conflicting goals. Indeed, as Jennings and others have argued [88], from a practical viewpoint, the difference between these architectures lies simply in the stage of the development process at which the system must reason; the design of the behaviour of reactive agents occurs during their construction whereas with deliberative agents it occurs at run-time.

Similarly, multi-agent systems may be developed incorporating both a micro-level and macro-level design perspective. ARCHON, for example, incorporates both views, providing a methodology for top-level design relating to macro-level aspects such as communication configurations as well as micro-level aspects such as generic agent architectures. The distinction between the micro-level and macro-level view is analogous in some respects to the distinction between deliberative and reactive architectures. The more a macro-level stance is taken, the more the relationships and interactions are developed at design-time. Alternatively, it is implicit in a micro-level stance that agents will form their own group relationships dynamically and participate in interactions in order to achieve their individual goals at run-time.

We have described systems that have been successful in producing desirable aspects of intelligent agent behaviour from different architectural paradigms. The most significant problem with most of these systems is their weak relation to theoretical principles; without knowing in what way a system embodies a theoretical model, no general principles can exist to guide agent-practitioners in general. For example, Rao argues [132], implemented BDI systems incorporate so many assumptions and simplifications that they do not have a strong theoretical underpinning; theoretical models of mental attitudes use modal logics but implemented systems use data structures.

Conceptually, models expressed in temporal and modal logic are difficult to understand. Moreover, since different behaviours arise according to different axiomatisations between modalities, any agent specification defines very precisely the behaviour of an agent at an early stage. This is a problem in the software engineering view where specifications are the platforms for design, since it is often required that specifications do not force design decisions to be made during the early development stages. In addition, modal logics are computationally ungrounded [172] and it is not clear how either implementations could be verified with respect to a specification, or how specifications could be refined to implementations.

The divide between the theoretical, idealised models of agency and the engineering practice of building them has been widely recognised for a number of years. Uniting theory and practice, or at least bringing them significantly closer, is now seen as critical to the uptake of agent research by the mainstream software engineering community. Agent systems do not embody any fundamental or principled theory because current theories are unnecessarily complex, unrealistic and unrelated to the development of software. Therefore, future theoretical research must construct intuitive, formal and computational models that can be directly and usefully applied to agent systems to provide a strong correspondence between theory and practice. In this thesis we build a model inspired by these needs which is intended to span and relate definitions, dimensions, architectures, theories and systems.
Chapter 3

The Agency Framework

3.1 Introduction

Though agents are becoming increasingly popular across a wide range of applications, the rapid growth of the field has led to much confusion regarding agents and their functionality. One part of this confusion is that it is now generally recognised that there is no agreement on what it is that makes something an agent. For example, Franklin and Graesser [61] provide evidence of the degree of disparity which exists, citing ten different definitions of leading agent practitioners. This lack of consensus sets up barriers to the development of an accepted foundation on which to build a rigorous scientific discipline. It can also be argued that it has already hindered research since integration and comparison of different approaches and results is made very difficult because of the plethora of different terms and notions.

To address this confusion, it is sensible that a well-defined and precise vocabulary for the fundamental elements of agents and multi-agent systems be developed. If such a vocabulary is also situated in a structured framework, it can provide the right kind of platform on which to base further research. Formal specification techniques are appropriate for this task. It has been claimed elsewhere that formal specification can be used to construct formal frameworks within which common properties of a family of systems can be identified [33, 42, 62, 63, 64]. As a result of such specifications, it becomes possible to consider different systems as instances of one design, and how new designs can be constructed from an existing design framework.

More precisely we argue that a formal framework must satisfy three distinct requirements, as follows.

- It must provide meanings for common concepts and terms precisely and unambiguously, and do so in a readable and understandable manner. The availability of readable explicit notations allows a movement from a vague and conflicting understanding of a class of models towards a common conceptual framework. A common conceptual framework exists if there is a generally held understanding of the salient features and issues involved in the relevant class of models.

- It must be sufficiently well-structured to provide a foundation for subsequent development of new and increasingly more refined concepts. In particular, it is important that a practitioner is in a position to choose the level of abstraction suitable for their current purpose.

- It must enable alternative designs of particular models and systems to be presented explicitly, compared and evaluated. It must provide a description of the common abstractions found within that class of models as well as a means of further refining these descriptions to detail particular models and systems.

In this chapter, we lay the foundations for a principled theory of agency by describing just such a framework, which we call the agency framework. This framework is essentially a four-tiered hierarchy comprising entities, objects, agents and autonomous agents where agents are viewed as objects with
goals, and autonomous agents are agents with motivations. In the next section we argue that the Z specification language is an appropriate formal method and provide a brief overview of the notation. The next sections describe a framework for agency and autonomy beginning with the base initial concepts and continuing with descriptions of entities which provide a template that can be used to define objects, agents and autonomous agents in turn. We then show how our framework can be applied to specify an example architecture for systems called tropistic agents. Finally, we provide a summary, consider related work in detail and present some conclusions. The basic strategy for introducing new aspects of the formal framework and later models in this thesis is as follows. First, we provide an intuitive description of what is required and why, then we provide, where appropriate, a textual definition and lastly we introduce the specification that formalises the definition.

3.2 Z Specification Language

In this work we view our enterprise as that of building programs. Z is particularly suitable in squaring the demands of formal modelling with the need for implementation by allowing transition between specification and program. There are many well-developed strategies and tools to aid this transformation. Programs can also be verified with respect to a specification; it is possible to prove that a program behaves precisely as set out in the Z specification. As we discussed in the previous chapter, this is not possible when specifications are written in modal logics since they have the computationally ungrounded possible-worlds model as their semantics. Thus our approach to formal specification is pragmatic; we need to be formal to be precise about the concepts we discuss, yet we want to remain directly connected to issues of implementation.

The Z language is sufficiently expressive to allow a consistent, unified and structured account of a computer system and its associated operations. Structured specifications, which are made possible by Z’s use of schemas and schema inclusion, enable the description of systems at different levels of abstraction, with system complexity being added at successively lower levels. Ascribing various agent qualities to systems can be viewed as abstraction mechanisms [173] and thus the ability to describe systems at different levels of abstraction is a useful tool.

Z schema boxes have proved appropriate in the construction of formal frameworks in other areas such as hypertext systems [42] and high-performance systems [33]. In particular, schemas are ideal for manipulation in the design process since by viewing the design process as a constraint of possible states, design strategies can be presented as further predicates in an abstract state schema. The use of abstraction renders prejudice about design unnecessary and enables a well-structured specification of a general system to be written. We say that a specification is well-structured if it describes a system at its most abstract level and then, through using schema inclusion and refinement, specifies the system at each subsequent lower level. Through this use of schema inclusion, the relationships between different levels of abstraction are well-defined, and easy transition between them is facilitated.

The Z specification language is increasingly being used both in industry and academia, as a strong and elegant means of formal specification, and is supported by a large array of books (e.g. [4, 76, 154, 168]), articles (e.g. [5, 6, 78]), industrial case studies (e.g. [19, 25, 166]) and development tools (e.g. [77, 155, 140]). Furthermore, Z (along with the related language VDM [75]), is gaining increasing acceptance as a tool within the artificial intelligence community (e.g. [24, 70, 110, 171]) and is therefore appropriate for the current work in terms of standards and dissemination capabilities. In particular, Z is more widely accessible than many other formalisms since it is based on existing elementary components such as set theory and first order predicate calculus.

3.2.1 State Schemas

Z is based on set theory and first order predicate calculus but extends these languages by allowing an additional mathematical type known as the schema. These schemas are either state schemas or operation schemas. Z state schemas have two parts: the upper, declarative, part which declares variables and their types, and the lower, predicate, part which relates and constrains these variables. The type of any schema can be considered as the Cartesian product of the types of each of its variables, without any notion of order, but constrained by the predicates.
State schemas describe the possible states of a system. Consider, for example, the following schema, $\text{Schema}_1$, which contains three state variables and one constraining predicate. It describes a system that contains three variables, $\text{variable}_1$ and $\text{variable}_2$ of type $\text{Type}_A$, and $\text{variable}_3$ of $\text{Type}_B$, such that $\text{variable}_1$ and $\text{variable}_2$ are not equal.

$\text{Schema}_1$

$\text{variable}_1, \text{variable}_2 : \text{Type}_A$
$\text{variable}_3 : \text{Type}_B$

$\text{variable}_1 \neq \text{variable}_2$

It is possible to select a specific state variable of a schema. If, for example, we wish to select $\text{variable}_2$ from $\text{Schema}_1$, we would write $\text{Schema}_1.\text{variable}_2$. Modularity is facilitated in Z by allowing schemas to be included within other schemas. This is illustrated by virtue of the equivalence of the following two schemas.

$\text{Schema}_2$

Schema

$\text{variable}_4 : \text{Type}_B$

$\text{variable}_3 = \text{variable}_4$

$\text{Schema}_2$

$\text{variable}_1, \text{variable}_2 : \text{Type}_A$
$\text{variable}_3, \text{variable}_4 : \text{Type}_B$

$\text{variable}_1 \neq \text{variable}_2$
$\text{variable}_3 = \text{variable}_4$

### 3.2.2 Operation Schemas

Operations are characterised by their effect on state. An operation schema relates the variables of a state schema before and after an operation. In general, an operation schema has a before state denoted by undashed state variables, an after state denoted by dashed state variables, inputs denoted by question-marks and outputs denoted by exclamation-marks. For example, consider an operation, $O_p$, on the state schema, $\text{Schema}_2$, where $\theta \text{Schema}_2$ represents the state variables of $\text{Schema}_2$ before the operation, and $\theta \text{Schema}_2'$ the state variables after the operation.

$O_p$

$\text{Schema}_2$

$\text{Schema}_2'$

$\text{input}_1? : \text{Type}_{1_i}$
$\text{input}_2? : \text{Type}_{1_2}$
$\text{output}_1!: \text{Type}_{0_i}$
$\text{output}_2!: \text{Type}_{0_2}$

$P_1 \wedge P_2 \ldots \wedge P_n$

$\theta \text{Schema}_2' = R\text{Exps}(\theta \text{Schema}_2, \text{inputs}, \text{global variables})$
$\text{output}_1! = \text{Exp}_1$
$\text{output}_2! = \text{Exp}_2$
This operation schema can be described as follows. \( Op \) changes the state of \( Schema_2 \) and has two inputs, \( input_1 \) of type \( Type_1 \), and \( input_2 \) of type \( Type_2 \). In addition, there are two outputs, \( output_1 \) of type \( Type_{O1} \), and \( output_2 \) of type \( Type_{O2} \). There are \( n \) pre-conditions \( P_1, P_2, \ldots, P_n \), a set of relationships defining the values of the after-state variables, \( Schema_2' \), in terms of expressions, \( RExp_s \), dependent on the before-state variables \( Schema_2 \), the inputs, and any constant global variables which may have been defined previously. Finally, the outputs, \( output_1 \) and \( output_2 \), are given the values \( Exp_1 \) and \( Exp_2 \). This notion of change is made explicit by the ‘delta’ (\( \Delta \)) convention, which is defined as follows.

\[
\begin{align*}
\Delta Schema_2 \\
Schema_2 & \\
Schema_2' & \\
\end{align*}
\]

If the operation, \( Op \), does not affect any of the variables from \( Schema_1 \), then the ‘xi’ (\( \equiv \)) convention can be used as shown below.

\[
\begin{align*}
Op & \\
\Delta Schema_2 & \\
\equiv & Schema_1 \\
\end{align*}
\]

This states that the variables of \( Schema_1 \) are unchanged as a result of operation \( Op \). Specifically, the operation can only change the value of \( variable_4 \).

### 3.2.3 Other Types

To introduce a type in \( Z \), where nothing is stated about the elements of the type, a given set is used. Notationally this is achieved by enclosing a type name in square brackets. For example, \([NODE] \) is written to declare the set \( NODE \) as a new type. Declaring a variable of this type is achieved by writing \( node_1 : NODE \). In order to define a variable to be some set of values of a type, rather than an individual value, we use the power set of a type denoted by \( \mathcal{P} \). The expression, \( nodes : \mathcal{P} NODE \), declares the variable \( nodes \) to be of type power set of nodes, or equivalently, a set of nodes. The set of non-empty sets of nodes is written \( \mathcal{P} \_1 NODE \). In order to introduce a variable which is an ordered pair of values from existing types the Cartesian product, denoted by the symbol, \( \times \), is used. For example, writing \( pair : NODE \times NODE \), declares the variable \( pair \) to be an ordered pair such as \( (node_1, node_2) \). These can be split into their component parts using the functions \( first \) and \( second \).

In this case we have the following predicates.

\[
first(node_1, node_2) = node_1 \land second(node_1, node_2) = node_2
\]

A relation expresses some relationship between two types, known as the source and target. The type of a relation between source set \( X \) and target set \( Y \), written \( X \leftrightarrow Y \), is equivalent to \( \mathcal{P}(X \times Y) \). In this way, a relation is defined as a set of ordered pairs of \( X \times Y \). If the pair \( (x, y) \) exists in the relation \( R \), then \( y \) is related to \( x \) according to \( R \). Consider the following function between nodes.

\[
Rel = \{ (node_1, node_2), (node_2, node_3), (node_3, node_2), (node_4, node_1) \}
\]

The domain of a relation or function, written \( \text{dom} \), identifies those elements in the source set that are related, and the range, written \( \text{ran} \), identifies those elements in the target set that are related. Using the example relation described above we can make the following assertions.

\[
\text{dom} Rel = \{ node_1, node_2, node_3, node_4 \} \land \text{ran} Rel = \{ node_2, node_3, node_4 \}
\]

The inverse of a relation is written \( R^{-1} \) and can be found by swapping each pair in the relation.
Table 3.1: Summary of Z Notation

<table>
<thead>
<tr>
<th>Definitions and declarations</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a, b) Identifiers</td>
<td>(A \rightarrow B) Partial function</td>
</tr>
<tr>
<td>(p, q) Predicates</td>
<td>(A \rightarrow B) Total function</td>
</tr>
<tr>
<td>(s, t) Sequences</td>
<td>(A \rightarrow B) Partial Injection</td>
</tr>
<tr>
<td>(x, y) Expressions</td>
<td>(A \rightarrow B) Bijection</td>
</tr>
<tr>
<td>(A, B) Sets</td>
<td>(\text{Sequences})</td>
</tr>
<tr>
<td>(a == x) Abbreviated definition</td>
<td>(\text{Seq} A) Finite sequences</td>
</tr>
<tr>
<td>([a]) Introduction of given set</td>
<td>(\text{Seq}_1 A) Non-empty finite sequences</td>
</tr>
<tr>
<td>(a ::= b{B})</td>
<td>(\text{Iseq} A) Injective sequences</td>
</tr>
<tr>
<td>(</td>
<td>c{C}|) Free type declaration</td>
</tr>
<tr>
<td>(\mu d \mid P) Definite description</td>
<td>(\langle x, y, \ldots \rangle) Sequence ({(1, x), (2, y), \ldots })</td>
</tr>
<tr>
<td>Logic</td>
<td>(s \sim t) Sequence concatenation</td>
</tr>
<tr>
<td>(\neg p) Logical negation</td>
<td>(\text{Head} s) First element of sequence</td>
</tr>
<tr>
<td>(p \wedge q) Logical conjunction</td>
<td>(\text{Last} s) Last element of sequence</td>
</tr>
<tr>
<td>(p \vee q) Logical disjunction</td>
<td>(s \in t) Subsequence Relation</td>
</tr>
<tr>
<td>(p \Rightarrow q) Logical implication</td>
<td>(\text{Sequence})</td>
</tr>
<tr>
<td>(p \Leftrightarrow q) Logical equivalence</td>
<td>(\text{Vertical schema})</td>
</tr>
<tr>
<td>(\forall X \bullet q) Universal quantification</td>
<td>(S) Vertical schema</td>
</tr>
<tr>
<td>(\exists X \bullet q) Existential quantification</td>
<td>(d) Axiomatic definition</td>
</tr>
<tr>
<td>Sets</td>
<td>(\text{Operation schema})</td>
</tr>
<tr>
<td>(x \in y) Set membership</td>
<td>(\triangle S) Operation schema</td>
</tr>
<tr>
<td>(x \notin y) Non-membership</td>
<td>(\emptyset S) Binding formation</td>
</tr>
<tr>
<td>{}\ Empty set</td>
<td>(z.a) Component inclusion</td>
</tr>
<tr>
<td>(A \subseteq B) Set inclusion</td>
<td>(\text{Conventions})</td>
</tr>
<tr>
<td>(A \subset B) Strict set inclusion</td>
<td>(a?) Input to an operation</td>
</tr>
<tr>
<td>{(x, y, \ldots)} Set of elements</td>
<td>(a!) Output from an operation</td>
</tr>
<tr>
<td>(\langle x, y, \ldots \rangle) Ordered tuple</td>
<td>(a) State component before operation</td>
</tr>
<tr>
<td>(A \times B \times \ldots) Cartesian product</td>
<td>(a!) State component after operation</td>
</tr>
<tr>
<td>(\mathbb{P} A) Power set</td>
<td>(S) State schema before operation</td>
</tr>
<tr>
<td>(\mathbb{P}_1 A) Non-empty power set</td>
<td>(S') State schema after operation</td>
</tr>
<tr>
<td>(A \cup B) Set union</td>
<td>(\triangle S) Change of state</td>
</tr>
<tr>
<td>(A \setminus B) Set difference</td>
<td>(\equiv S) No change of state</td>
</tr>
<tr>
<td>(\cap A) Generalised intersection</td>
<td>(\cup A) Generalised union</td>
</tr>
<tr>
<td>#A Size of finite set</td>
<td>(\text{Relational Overriding})</td>
</tr>
<tr>
<td>Relations</td>
<td></td>
</tr>
</tbody>
</table>
\[ \text{Rel}^{-1} = \{(\text{node}_2, \text{node}_1), (\text{node}_3, \text{node}_2), (\text{node}_2, \text{node}_3), (\text{node}_4, \text{node}_4)\} \]

The relational image of some subset of the source set, \( S \), written \( \text{Rel}(S) \), is the set of elements in the target set which are related to some element in \( S \).

\[ \text{Rel}(\{\text{node}_1, \text{node}_2\}) = \{\text{node}_2, \text{node}_3\} \]

When all elements from the source are not related to more than one element in the target type, the relation is a function. A total function denoted by \( \rightarrow \), relates every element in the source set, while a partial function, denoted by \( 
leftrightarrow \), does not necessarily relate every element. A partial injective function \( \nleftrightarrow \) is one where no two elements in the domain relate to the same element in the range and a total surjective function is one where every element in the target set is related \( \nleftrightarrow \). Finally, a bijection \( \nleftrightarrow \) is total, injective and surjective.

Sequences are enclosed in angled brackets \( \langle \rangle \). A sequence of elements of type \( \text{Node} \) has type \( \text{seq} \text{Node} \). Sequences can additionally be defined to be non-empty \( \text{seq}_1 \), injective \( \text{isseq} \) or both \( \text{isseq}_1 \). A sequence is defined as a function from the natural numbers, \( \mathbb{N} \), to \( X \) such that the domain is equal to the contiguous set of numbers from 1 up to the number of elements in the sequence. For example, we have the following equality.

\[ \langle a, c, b \rangle = \{(1, a), (2, b), (3, c)\} \]

The \textit{head} and \textit{last} of a sequence extract the first and last element of a sequence. Two sequences can also be concatenated using the ‘\( \cdot \)’ operator.

\[ \text{head} \langle a, c, b \rangle = a \land \text{last} \langle a, c, b \rangle = b \]
\[ \langle a, c, b \rangle \cdot \langle b, c, a \rangle = \langle a, c, b, c, a \rangle \]

This completes a description of most of the Z notation used in this thesis. When other new notation is introduced we will also provide explanations in the text. A table of the complete set of Z notation used in this thesis, along with associated textual descriptions, is shown in Table 3.1. In addition, complete Z definitions are provided in Appendix A.

### 3.2.4 Specification Structure Diagrams

Diagrams are used to detail the way in which schemas are used to produce a specification structure, which provides a graphical overview of the way in which the formal models in this thesis are constructed. The key to these diagrams is presented in Figure 3.1 and is explained below.

**State Schema** – State schemas are represented by a box enclosing the schema name.

**Operation Schema** – Operation schemas are represented by a hexagon enclosing the schema name.

**Schema Inclusion** – A solid arrow between boxes represents state schema inclusion. In the case shown in Figure 3.1, \( S1 \) is included in \( S2 \).

```
S2
S1
...
```

**Variable Inclusion** – A dashed arrow between boxes represents a schema being included in the declarative part of the second schema as a type. In the case shown in Figure 3.1 a variable included in the state schema \( S2 \) is declared in terms of the type defined by state schema \( S1 \). For example, the schema below includes a variable that is defined as a set of elements of the schema type \( S1 \).
Operation on State  – A solid arrow between a hexagon, $O$, and a box, $S$, indicates that the operation schema, $O$, is defined in terms of a state change to the state schema, $S$.

```
\[ S2 \]
\[ \text{variable}_1 : \mathcal{P} S1 \]
\[ \ldots \]
```

Operation Inclusion  – A solid arrow between two hexagons represents operation inclusion. In the case shown in Figure 3.1 the operation schema $O2$ includes the operation schema $O1$.

```
\[ O \]
\[ \triangle S \]
\[ \ldots \]
```

State Operation Inclusion  – A dashed arrow between a hexagon and a box indicates that the state schema has been included in the definition of an operation schema. In the case shown in Figure 3.1 the state schema $S$ is included but its state is unaffected.

Schema Disjunction  – A set of converging dashed arrows between a set of hexagons $O1$, $O2$ and another hexagon $O$ indicates that the operation schema $O$ is defined as the disjunction of the operation schemas $O1$ and $O2$. The pre-condition of $O$ is the logical disjunction of the pre-conditions of $O1$ and $O2$. 

44
3.3 Initial Concepts

As discussed in Chapter 1 and at the beginning of this chapter we have seen that there is no consensus on basic agent terminology or concepts. It is therefore necessary to start from first principles in constructing our agent framework. However, there is such a wealth of existing work that we must ensure it can be related and evaluated with respect to the framework. We consider the appropriate details in Section 3.10.

Before it is possible to construct agent models it is necessary to define the building blocks or primitives from which these models are created. We start by defining three primitives: attributes, actions and motivations, which are used as the basis for development of the agent framework described in this chapter, and all subsequent models. Formally, these primitives are specified as given sets. In addition, we specify two secondary concepts: goals and environments in terms of attributes.

Attributes are simply features of the world, and are the only characteristics that are manifest. They need not be perceived by any particular entity, but must be potentially perceivable in an omniscient sense. This notion of a feature allows anything to be included such as, for example, the fact that a tree is green, or is in a park, or is twenty feet tall.

**Definition:** An attribute is a perceivable feature.

In Z, this is defined as follows.

\[
\text{Attribute} \equiv \mathbb{P}_1 \text{Attribute}
\]

An environment is then simply a set of attributes that describe all the features within that environment. Thus a new type, Environment, is defined to be a (non-empty) set of attributes.

\[
\text{Environment} \equiv \mathbb{P}_1 \text{Attribute}
\]

The second primitive that needs defining is an action. Actions can change environments by adding or removing attributes. For example the action of a robot, responsible for attaching tyres to cars in a factory, moving from one wheel to the next, will delete the attribute that the robot is at the first wheel and add the attribute that the agent is at the second.

**Definition:** An action is a discrete event that can change the state of the environment when performed.

\[
\text{Action} \equiv \mathbb{P}_1 \text{Attribute}
\]

A goal defines a state of affairs that is desirable in some way. For example, a robot may have the goal of attaching a tyre to a car.

**Definition:** A goal is a state of affairs to be achieved in the environment.

We can define goals to be just (non-empty) sets of attributes that describe a state of affairs in the world.

\[
\text{Goal} \equiv \mathbb{P}_1 \text{Attribute}
\]

We also introduce motivations but delay their discussion until Section 3.7 at which time they will be more relevant.

**Definition:** A motivation is any desire or preference that can lead to the generation and adoption of goals and that affects the outcome of the reasoning or behavioural task intended to satisfy those goals.

As with actions and attributes, the type of all motivations is defined as a given set.

\[
\text{Motivation} \equiv \mathbb{P}_1 \text{Attribute}
\]

The formal specification that follows is constructed solely from these three primitive types to provide models and definitions of entities, objects, agents and autonomous agents, which are related as shown in the Venn diagram of Figure 3.2. In this view, all autonomous agents are agents, all agents are objects and all objects are entities. Further, there is a critical distinction between autonomous agents and non-autonomous agents that will provide the basis for the analysis described in subsequent chapters. We refer to the model described in this chapter as the *agency framework.*
3.4 Entities

The entity, as defined in this section, serves as an abstraction mechanism. It provides a template from which objects, agents and autonomous agents can be defined. In addition, we require that the specification initially describes components in a multi-agent system at the highest possible level of abstraction. In response, we define anything that is considered to be a single component, an entity. For example, a tea-cup is an entity as is a robot. These entities may have complex descriptions, but at the very highest level they are just collections of attributes. We do not care which attributes are grouped together to describe a given entity, or how that is achieved. We are only concerned with being able to describe a collection of attributes as a single component. An entity is defined using the existing primitives.

**Definition:** An entity is something that comprises a non-empty set of attributes, a set of actions, a set of goals and a set of motivations.

The schema below formalises the definition of an entity. It has a declarative part containing four variables. What is critical about the schema is that the set of attributes of the entity, attributes, is non-empty. The other components of the schema will be discussed in due course in the subsequent three sections, respectively.

<table>
<thead>
<tr>
<th>Entity</th>
</tr>
</thead>
<tbody>
<tr>
<td>attributes : ( \mathbb{P} ) Attribute</td>
</tr>
<tr>
<td>capabilities : ( \mathbb{P} ) Action</td>
</tr>
<tr>
<td>goals : ( \mathbb{P} ) Goal</td>
</tr>
<tr>
<td>motivations : ( \mathbb{P} ) Motivation</td>
</tr>
<tr>
<td>attributes ( \neq {} )</td>
</tr>
</tbody>
</table>

The attributes of an entity are limited to those features that are permanent. That is to say that attributes are features that can be ascribed to the entity itself rather than aspects of its particular state. Attributes relating to such transient things as the entity’s orientation, configuration (which refers to elements such as the angle of a robot’s arm), location, and so on, which are not solely dependent on the entity, but are also a factor of its environment, are not represented. By contrast, colour, age, mass, density and size, for example, which are fixed and which are a function only of the entity itself, are represented.\(^1\) Returning to the example of the tea-cup, the attributes of the cup may state that

\(^1\) Of course some of these attributes may also change over time. For example, an agent on a diet may lose mass and gain
it is stable, blue, hard, and so on. They would not specify its temperature, position, whether it was standing on its base or whether it was full. A robot’s attributes may specify that it is red, large, heavy, and has three arms but would not include the current position of those arms.

Unless an entity refers to everything, such as the universe, it must be situated in an environment. Conversely, an environment will include all the entities within it. The Env schema formalises an environment whose state is represented by the environment variable of type Environment, which must consist of a non-empty set of attributes. The entities variable refers to the set of entities in the environment. The last predicate formalises the necessity that the sum of the attributes of the entities in an environment is a subset of all the attributes of that environment.

\[
\begin{align*}
\text{Env} & \\
\text{environment} : \text{Environment} \\
\text{entities} : \mathbb{P} \text{Entity} \\
\text{environment} \neq \{ \} \\
\cup \{ e : \text{entities} \bullet e.\text{attributes} \} \subseteq \text{environment}
\end{align*}
\]

### 3.4.1 Entity State

Once an entity is placed in an environment, aspects such as its orientation and location, which are part of its state, can be specified.

Those features of an entity that are not fixed (as represented by attributes), but include aspects of its current state, we collectively refer to as the entity’s situation. For example, the situation of a cup might now include that it is upright and situated on a table, while the situation of a robot may specify that it is located in the first floor of a car factory, holding a tyre. The attributes and situation are both part of the environment however, an attribute cannot be part of both the entity’s attributes and its situation.

The next schema formalises the state of an entity in an environment and includes the schema representing the entity itself, Entity, and the schema specifying the current environment, Env. This is equivalent to incorporating all of the declarations and predicates from both schemas. In addition, an extra variable, situation, is specified to denote the entity’s current situation. The union of attributes and situation is a proper subset of the environment, since equality would entail that the entity is isolated, and the environment is effectively nonexistent.

\[
\begin{align*}
\text{EntityState} & \\
\text{Entity} & \\
\text{Env} & \\
\text{situation} : \mathbb{P} \text{Attribute} \\
\text{situation} \neq \{ \} \\
\text{attributes} \cap \text{situation} = \{ \} \\
\text{attributes} \cup \text{situation} \subseteq \text{environment}
\end{align*}
\]

### 3.4.2 Entity Operations

We now move to specify the constraints on the way in which an entity changes its state in an environment. Changes to the state of entities do not affect their attributes, only their situation. For example, a cricket ball may change its orientation and location during the course of a cricket match since these...
are aspects of its situation. However, if the ball is hit hard, it may lose its shape, which is one of the ball’s attributes, so that it can no longer be used for cricket. In general, if the attributes of an entity change, then a new entity is instantiated.

To describe this, we need to introduce an operation schema that describes the relationship between two states; the state before the operation, and the state afterwards. The \( \triangle \) EntityState schema specifies that a change to the EntityState schema will leave the Entity schema unchanged, indicated by ‘\( \equiv \) Entity’. In addition, the resulting situation of an entity will always be contained in the new environment as shown by the only schema predicate.

\[
\begin{align*}
\triangle & \text{EntityState} \\
\text{EntityState} & \equiv \text{Entity} \\
\text{situation'} & \subseteq \text{environment'}
\end{align*}
\]

### 3.5 Objects

In this section we define an object, in terms of our previous definition of an entity, in order to build a model of entities and objects where different levels of abstraction are related. If we consider entities at a lower level of abstraction (in more detail) it may be possible to identify ways in which they interact with their environment. Entities able to affect their environment through action leads to the concept of objects, which we define in terms of entities as follows.

**Definition:** An object is an entity with a non-empty set of actions.

Thus, an object in our model is an entity with capabilities. The Object schema below has a declarative part that simply includes the previously defined schema, Entity. The predicate part of the schema specifies that an object must have a non-empty set of actions as well as attributes. Objects are therefore defined by their ability in terms of their actions, and their characteristics in terms of their attributes. Whilst the attributes of an entity are manifest and are always potentially observable, the capabilities of an object are not.

\[
\begin{align*}
\text{Object} & \\
\text{Entity} & \\
capabilities & \neq \{ \}
\end{align*}
\]

As an example of an object, consider again the robot, and assume that it does not have a power supply. Since the robot has no power, its capabilities are severely limited and include just those which rely on its physical presence, such as supporting things, weighing things down, and so on. Similarly, the capabilities of the tea-cup include that it can support things and that it can contain liquid. Once the robot is connected to a power supply it becomes a new object with increased capabilities, including being able to fix tyres to a car and reporting defective ones.

#### 3.5.1 Object Behaviour

Since an object has actions, these may be performed in certain environments that will be determined by the state of that environment. The behaviour of an object can therefore be modelled as a mapping from the environment to a set of actions that are a subset of its capabilities. This mapping is known as the action-selection function.

The ObjectAction schema below formalises this view and refines the Object schema, which is included. The action-selection function, objectactions, determines which set of actions are performed next in a given environment and is defined as a total function. That is, given an environment, it returns
a (possibly empty) set of actions.\(^2\) The assertion in the predicate part of the schema constrains the next actions to be taken (determined by applying \textit{objectactions}) by the object to be within the object’s capabilities.

\[
\text{ObjectAction} \\
\text{Object} \\
\text{objectactions} : \text{Environment} \rightarrow \mathcal{P} \text{ Action} \\
\forall \text{ environment} : \text{Environment} \bullet (\text{objectactions environment}) \subseteq \text{capabilities}
\]

3.5.2 Object State

As with an entity, an object must be situated in an environment and will have a \textit{situation} as specified previously. The environment, which includes the object’s situation, can be used to determine those actions that it is to perform next.

We define the state of an object in its environment in the \textit{ObjectState} schema by refining the schema, \textit{EntityState}, which contains the attributes and situation of the object, and the environment in which the object is situated, and also includes the \textit{ObjectAction} schema. The variable, \textit{willdo}, specifies the next actions that the object will perform. It is redundant since it is recoverable in exactly the way in which it is specified by applying the \textit{objectactions} function from the \textit{ObjectAction} schema to the current \textit{environment}, and is a subset of the capabilities of the object.

\[
\text{ObjectState} \\
\text{EntityState} \\
\text{ObjectAction} \\
\text{willdo} : \mathcal{P} \text{ Action} \\
\text{willdo} = \text{objectactions environment} \\
\text{willdo} \subseteq \text{capabilities}
\]

For example, the tyre-attaching robot, in a situation which includes holding a tyre, may now have \textit{willdo} as a set of actions to attach the tyre to the car.

3.5.3 Object Operations

So far, we have described an object and the way in which its actions are selected. Next, we describe how the performance of these actions affects the environment in which the object is situated. Those variables that relate to the state of the object (its situation and next actions) can change, while the other variables that are not concerned with state but with the \textit{nature} of the object itself (namely, its attributes, capabilities and action-selection function) remain unchanged. If these later variables ever did change, then a \textit{new} object would be instantiated. In this view a robot without a power supply is a different object from a robot with a power supply.

The \(\triangle \text{ObjectState}\) schema shows how these constraints are formalised. It refines the \(\triangle \text{Entity}\) schema, which asserts that \textit{situation} rather than \textit{attributes} change, and further states that none of the variables in \textit{ObjectAction} (which includes \textit{Object}) are affected by a change of state.

\[
\triangle \text{ObjectState} \\
\text{ObjectState} \\
\text{ObjectState}' \\
\triangle \text{EntityState} \\
\equiv \text{ObjectAction}
\]

\(^2\)These actions refer to those that are \textit{performed} by the acting agent rather than those \textit{selected}, which are are not considered in the agency framework. However, the framework can be extended to develop theories of how agents recover from situations where the actions performed are not the actions selected [37].
Now, when actions are performed in an environment, we say that an interaction takes place. An interaction changes the state of the environment by adding and removing attributes. In our model, all actions result in the same change to an environment whether taken by an object, agent or autonomous agent. The function that formalises how the environment is affected by actions performed within it can therefore be defined axiomatically. This function maps the current environment and the performed actions to the resulting environment.

\[
\text{effectinteraction} : \text{Environment} \rightarrow \mathbb{P} \text{Action} \rightarrow \text{Environment}
\]

This allows us to model an object interacting with its environment. Both the state of the object and the environment change as specified by the schema ObjectInteracts. The resulting environment is determined by applying effectinteraction to the current state of the environment and the current set of actions. In turn, this environment then determines the next set of actions to be performed by applying objectactions again.

\[
\begin{align*}
\Delta \text{ObjectState} \\
\text{environment}' = \text{effectinteraction environment willdo} \\
\text{willdo}' = \text{objectactions environment}'
\end{align*}
\]

3.6 Agency

3.6.1 Introduction

There are many dictionary definitions for an agent. Wooldridge and Jennings [173] quote the definition of an agent as “one who, or that which, exerts power or produces an effect.” However, they omit the second sense of agent, which is given as “one who acts for another . . . .” This is important, for it is not the acting alone that defines agency, but the acting for someone or something that is defining. Indeed, Wooldridge and Jennings acknowledge the difficulties in a purely action-based analysis of agency.

In our view agents are just objects with certain dispositions. Specifically, we regard an object as an agent if it is serving some purpose. They may always be agents, or they may revert to being objects in certain circumstances. This is explored further in the next chapter. For the moment, we concentrate on the nature of the disposition that characterises an agent. An object is an agent if it serves a useful purpose either to a different agent, or to itself, in which latter case the agent is autonomous (see Section 3.7). Specifically, an agent is something that satisfies a goal or set of goals (often of another). Thus if I want to use some object for my purpose, then that object becomes my agent. It has been ascribed or, if we anthropomorphise, has adopted, my goal. An agent is thus defined in relation to its goals.

3.6.2 Agent Specification

As stated previously, a goal is defined as a state of affairs to be achieved in the environment. An agent is defined in terms of an object as follows.

Definition: An agent is an object with a non-empty set of goals.

The formal description of an agent is specified by the Agent schema. This refines the object schema and constrains the set of goals to be non-empty.

\[
\begin{align*}
\text{Agent} \\
\text{Object} \\
\text{goals} \neq \{ \}
\end{align*}
\]

Thus an agent has, or is ascribed, a set of goals that it retains over any instantiation (or lifetime). One object may give rise to different instantiations of agents. An agent is instantiated from an object in response to another agent. Thus agency is transient, and an object that becomes an agent at some time may subsequently revert to being an object.

Note, that this definition means that in the limiting case, very simple non-computational entities without perception can be agents. For example, a cup is an object. We can regard it as an agent and ascribe to it mental state, but it serves no useful purpose to do so without considering the circumstances. A cup is an agent if it is containing liquid and it is doing so to some end. In other words, if I fill a cup with tea, then the cup is my agent; it serves my purpose. Alternatively, the cup would also be an agent if it were placed upside down on a stack of papers and used as a paperweight. It would not be an agent if it were just sitting on a table without serving any purpose to any one. In this case it would be an object. As this example shows, we do not require an entity to be intelligent for it to be an agent. Clearly, the example of the cup is counter-intuitive and it is much more intuitive to talk about robots, but it is important to realise that any object, computational or otherwise, can be an agent once it is serving a purpose.

Consider the robot example, and suppose now that the robot has a power supply. If the robot has no goal, then it cannot use its actuators in any sensible way but only, perhaps, in a random way, and must be considered an object. Alternatively, if the robot has some goal or set of goals that allow it to employ its actuators in some directed way, such as picking up a cup, or fixing a tyre onto a car, then it is an agent. The goal need not be explicitly represented, but can instead be implicit in the hardware or software design of the robot. It is merely necessary for there to be a goal of some kind.

Returning to the example of the cup as my agent, it is clear that not everyone will know about this agency. If, for example, I am in a cafe and there is a half-full cup of tea on my table, there are several views that can be taken. It can be regarded by the waiter as an agent for me, storing my tea, or it can be regarded as an object serving no purpose if the waiter thinks it is not mine or that I have finished. The view of the cup as an object or agent is relevant to whether the waiter will remove the cup or leave it at the table. Note that we are not suggesting that the cup actually possesses a goal, just that there is a goal that it is satisfying.

These examples highlight the range of behaviour that is available from agents. The tea-cup is passive and has goals imposed upon and ascribed to it, while the robot is capable of actively manipulating the environment by performing actions designed to satisfy its goals.

### 3.6.3 Agent Perception

We now introduce perception. An agent in an environment may have a set of percepts available, which are the possible attributes that an agent could perceive, subject to its capabilities and current state. We refer to these as the possible percepts of an agent. However, due to limited resources, an agent will not normally be able to perceive all those attributes possible, and will base its actions on a subset, which we call the actual percepts of an agent. Indeed, some agents will not be able to perceive at all.

In the case of a cup, for example, the set of possible percepts will be empty and consequently the set of actual percepts will also be empty. The robot, however, may have several sensors that allow it to perceive. Thus it is not a requirement of an agent that it is able to perceive.

To distinguish between representations of mental models and representations of the actual environment, we define a type, View, to be the perception of an environment by an agent. This has an equivalent type to that of Environment, but now we can distinguish between physical and mental components of the same type.

\[ \text{View} \equiv \mathbb{P}_1 \text{Attribute} \]

It is also important to note that it is only meaningful for us to consider perceptual abilities in the context of goals. Thus when considering objects without goals, perceptual abilities are not relevant. Objects respond directly to their environments and make no use of percepts even if they are available. We say that perceptual capabilities are inert in the context of objects.

An agent has a (possibly empty) set of actions that enable it to perceive its world, which we call its perceiving actions. The set of percepts that an agent is potentially capable of perceiving is a function of the current environment, which includes the agent’s situation and its perceiving actions.
Since the agent is resource-bounded, it may not be able to perceive the entire set of attributes and selects a subset based on its current goals. For example, the distributed Multi-Agent Reasoning System (dMARS) \[34\], may have a set of events that it has to process, where events correspond to environmental change. Each of these percepts is available to the agent but because of its limited resources it may only be able to process one event, and must make a selection based on its goals.

The perception capabilities of an agent are defined in the AgentPerception schema, which includes the Agent schema and refines it by introducing three variables. The set of perceiving actions is denoted by perceivingactions, a subset of the capabilities of an agent. The canperceive function determines the attributes that are potentially available to an agent through its perception capabilities. Notice that this function is applied to a physical environment (in which it is situated) and returns a mental environment. The second argument of this schema is constrained to be equal to perceivingactions. Finally, the function, willperceive, describes those attributes actually perceived by an agent. This function is always applied to the goals of the agent and in contrast to the previous function, takes a mental environment and returns another mental environment.

\[
\begin{align*}
\text{AgentPerception} & \\
\text{Agent} & \\
\text{perceivingactions} : \mathcal{P} \text{Action} & \\
\text{canperceive} : \text{Environment} \to \mathcal{P} \text{Action} \to \text{View} & \\
\text{willperceive} : \mathcal{P} \text{Goal} \to \text{View} \to \text{View} & \\
\text{perceivingactions} \subseteq \text{capabilities} & \\
\forall \ env : \text{Environment}; \ as : \mathcal{P} \text{Action} \quad & \\
\quad as \in \text{dom}(\text{canperceive} \ env) \Rightarrow as = \text{perceivingactions} & \\
\text{dom} \text{willperceive} = \{\text{goals}\} & \\
\end{align*}
\]

### 3.6.4 Agent Action

Any agent is still an object and can be viewed as such, so that the selection of actions is dependent solely on the environment. However, at the agent level of abstraction, goals and perceptions as well as the environment can be viewed as directing behaviour. This is specified by the agentactions function in the AgentAction schema below, which is dependent on the goals, the actual perceptions of the agent and the current environment itself. Since the objectactions function is still applicable for modelling the agent solely at the object level, the ObjectAction schema is included. The first predicate requires that agentactions returns a set of actions within the agent’s capabilities, while the last predicate constrains its application to the agent’s goals. If there are no perceptions, then the action-selection function is dependent only on the environment, as it is with objectactions.

\[
\begin{align*}
\text{AgentAction} & \\
\text{Agent} & \\
\text{ObjectAction} & \\
\text{agentactions} : \mathcal{P} \text{Goal} \to \text{View} \to \text{Environment} \to \mathcal{P} \text{Action} & \\
\forall \ gs : \mathcal{P} \text{Goal}; \ v : \text{View}; \ env : \text{Environment} & \\
\quad (\text{agentactions} \ gs \ v \ env) \subseteq \text{capabilities} & \\
\text{dom} \text{agentactions} = \{\text{goals}\} & \\
\end{align*}
\]

### 3.6.5 Agent State

Now, to describe an agent with capabilities and behaviours for perception and action situated in an environment, we include the two schemas previously defined for action and perception as well as the schema defining the agent as a situated object. The AgentState schema, which formalises an
agent situated in an environment, therefore includes the schemas $AgentAction$, $AgentPerception$ and $ObjectState$.

In addition, since the attributes of the environment are now accessible, it is possible to specify the possible percepts and actual percepts of the agent. These are denoted by the variables, possiblepercepts and actualpercepts, which are calculated using the canperceive and willperceive functions respectively.

Consider again the robot agent and the cup agent which are attaching tyres and storing tea respectively. Now, suppose that the robot also has perceptual capabilities that allow it to perceive attributes in its environment. Potentially, as a consequence of its current environment the robot may be able to perceive a multitude of attributes including that the car is red, a tyre is flat, the car door is open, and so on. Again, however, due to limited perceptual and processing abilities, and to the goal of attaching tyres, the actual percepts of the robot may only include that the tyre is flat and not the relatively insignificant attribute of the car being red. The cup agent, on the other hand, has no perceiving capabilities and consequently no possible or actual percepts.

Since goals are fixed for any agent, it is changes to the actual percepts of an agent that affect its selection of actions. An agent without perceptions does not therefore have any increased functionality as a result of having goals, but the behaviour of an agent without perceptions can still be viewed and modelled in terms of goals affecting its action-selection. In addition, as will be shown in the next chapter, it is necessary to model such as entity as an agent in order to analyse key inter-agent relationships.

### 3.6.6 Agent Operations

Operations characterising agent behaviour are constrained to affect only certain aspects. The attributes, capabilities, goals, perceptual capabilities, and action and perception selection functions are unchanged by any operation. If any of these variables change, a new agent is instantiated. The only variables that may change are necessarily associated with the state of the agent such as its situation and possible and actual percepts. These constraints are formalised in the $\Delta Agent$ schema, which defines a change in agent state. It includes $\Delta ObjectState$ to ensure that only the state properties of objects change and, in addition, that variables included in the $AgentAction$, and $AgentPerception$ schemas are unaltered.

When an agent acts in an environment, the environment changes according to the specific actions performed. This is not dependent on whether the entity is an object or an agent. Thus the schema describing object interaction is still directly applicable. Formally, the $AgentInteracts$ schema includes
Object Interacts and affects the state of an agent as specified by \( \Delta AgentState \). The three predicates of this schema show explicitly how the schema variables are updated.

\[
\begin{align*}
AgentInteracts & \\
\Delta AgentState & \\
ObjectInteracts & \\
posspercepts' &= can_perceive \text{ environment}' \text{ perceiving actions} \\
actualpercepts' &= wil_perceive \text{ goals } posspercepts' \\
willdo' &= agentactions \text{ goals } actualpercepts' \text{ environment}'
\end{align*}
\]

3.7 Autonomy

3.7.1 Introduction

The definition of agency developed so far relies upon the existence of other agents to provide the goals that are adopted when an agent is instantiated. In order to ground this chain of goal adoption, to escape what could be an infinite regress, and also to bring out the notion of autonomy, we introduce motivation.

Grounding the hierarchies of goal adoption demands that we have some agents that can generate their own goals. These agents are autonomous since they are not dependent on the goals of others, and possess goals that are generated from within rather than adopted from other agents. Such goals are generated from motivations, higher-level non-derivative components characterising the nature of the agent, but which are related to goals. Motivations are, however, qualitatively different from goals in that they are not describable states of affairs in the environment. For example, consider the motivation greed. This does not specify a state of affairs to be achieved, nor is it describable in terms of the environment, but it may (if other motivations permit) give rise to the generation of a goal to rob a bank. The distinction between the motivation of greed and the goal of robbing a bank is clear, with the former providing a reason to do the latter, and the latter specifying what must be done. We now re-state our definition of motivation.

**Definition:** A motivation is any desire or preference that can lead to the generation and adoption of goals and that affects the outcome of the reasoning or behavioural task intended to satisfy those goals.

A motivated agent is thus an agent that pursues its own agenda for reasoning and behaviour in accordance with its internal motivation. Since motivations ground the goal-generation regress, we claim that motivation is critical in achieving autonomy. An autonomous agent must be a motivated agent.

Although it draws on Kunda’s work on motivation in psychology [97], the definition used for motivation above expresses its role but does not tie us to any particular implementation. Indeed, there are several views as to exactly how the role of motivation as defined here can be fulfilled. Simon, for example, takes motivation to be “that which controls attention at any given time,” and explores the relation of motivation to information-processing behaviour, but from a cognitive perspective [148]. More recently, Sloman has elaborated on Simon’s work, showing how motivations are relevant to emotions and the development of a computational theory of mind [149, 150]. Some have used motivation and related notions such as motives [124], and concerns [113], in developing computational architectures for autonomous agents while others have argued for an approach based on rationality that relies on utility theory [138].

In this framework, we take a neutral stance on such detail by specifying motivation as a given set, omitting any further information. This allows us to use the concept of distinct and possibly conflicting

---

4Note the distinction between our definition and many existing ones such as that of Wooldridge and Jennings [173], which tend to be more general and not so precise. In what follows, when we use the word autonomous we will take it to mean the definition provided in this section.
motivations influencing the behaviour of the agent, but also defers the choice of the actual mechanism to a subsequent point of refinement or implementation. Moreover, whilst others have been concerned with modelling motivation [113, 124], our work is concerned with its use in defining autonomy.

### 3.7.2 Autonomous Agent Specification

An autonomous agent may now be defined.  

**Definition:** An autonomous agent is an agent with a non-empty set of motivations.  

It is specified simply as an agent with motivations.

\[
\text{AutonomousAgent}
\]

\[
\text{Agent}
\]

\[
\text{motivations} \neq \{ \}
\]

In illustration of these ideas, note that the cup cannot be considered autonomous because, while it can have goals ascribed to it, it cannot *generate* its own goals. In this respect it relies on other entities for purposeful existence. The robot, however, is potentially autonomous in the sense that it may have a mechanism for internal goal generation. Suppose the robot has motivations of achievement, hunger and self-preservation, where achievement is related to attaching tyres onto a car on a production line, hunger is related to maintaining power levels, and self-preservation is related to avoiding system breakdowns. In normal operation, the robot will generate goals to attach tyres to cars through a series of subgoals. If its power levels are low, however, it may replace the goal of attaching tyres with a newly-generated goal of recharging its batteries. A third possibility is that in satisfying its achievement motivation, it works for too long and is in danger of overheating. In this case, the robot can generate a goal of pausing for an appropriate period in order to avoid any damage to its components. Such a robot is autonomous because its goals are not imposed, but are generated in response to its environment. The views of the cup and robot in terms of the agent hierarchy are shown in Table 3.2 that provides example instantiations of the different requirements for each level.\(^5\)

### 3.7.3 Autonomous Agent Perception

With autonomous agents, therefore, it is both goals and motivations that are relevant to determining what is perceived in an environment. The schema below thus specifies a modified version of the non-autonomous agent’s *will perceive* function as *auto will perceive*. That which an autonomous agent is potentially capable of perceiving at any time is independent of its motivations. Indeed, it will always be independent of goals and motivations, and there is consequently no equivalent increase in functionality to *can perceive*.

---

\(^5\) According to our definition an autonomous agent is an object which has both motivations and goals. However, it may be possible to design motivated autonomous agents which, during moments of inactivity, do not have a current goal. For such entities to be described consistently within our framework it would be necessary to introduce a “null goal” so that they maintain their agency.
3.7.4 Autonomous Agent Action

An autonomous agent will have some potential means of evaluating behaviour in terms of the environment and its motivations. In other words, the behaviour of the agent is determined by both external and internal factors. This is qualitatively different from an agent that merely has goals because motivations are non-derivative and governed by internal inaccessible rules, while goals are derivative but relate to motivations. Specifically, the action-selection function for an autonomous agent is produced at every instance by the motivations of the agent. The next schema defines the action-selection function, autoactions and includes the AgentAction and AutonomousAgent schemas. The domain of the autoactions function is equal to the motivations of the agent.

\[
\text{autoactions} : \text{P Motivation} \rightarrow \text{P Goal} \rightarrow \text{View} \rightarrow \text{Environment} \rightarrow \text{P Action}
\]

3.7.5 Autonomous Agent State

In exactly the same way that the state of an agent is defined by refining the definition of the state of an object, the state of an autonomous agent is defined using the state of an agent. The actions performed by an autonomous agent are a function of its motivations, goals, percepts and environment.

\[
\text{willdo} = \text{autoactions motivations goals actualpercepts environment}
\]

3.7.6 Autonomous Agent Operations

In considering the definition of a change in state for an autonomous agent, there are some subtle but important differences with previous schemas. Whereas previously goals were fixed for agents as capabilities were for objects, we do not explicitly state whether motivations change when actions are performed. If they do change, then the agent functions, willperceive and agentactions, will also change. If they do not change, motivations may generate new and different goals for the agent to pursue. In any of these cases, the characterising features of an agent are in flux so that an autonomous agent can be regarded as a continually re-instantiated non-autonomous agent. In this sense, autonomous agents are permanently agents as opposed to transient non-autonomous agents, which may revert to being objects.
Finally, we specify the operation of an autonomous agent performing its next set of actions, a refinement of the AgentInteracts schema.

For the rest of this thesis we will refer to the four-tiered agent hierarchy we have introduced here as the agency framework.

3.8 Applying the Framework: Tropistic Agents

3.8.1 Tropistic Agents

The agency framework specifies a set of generic architectures. The types, functions and schemas it contains can be applied to other systems and concepts. In order to illustrate its use in this way, we reformulate tropistic agents [65], an example of an agent architecture. It is one of a set of core agent architectures used by Genesereth and Nilsson to demonstrate some key issues of intelligent agent design. The activity of tropistic agents, as with reflexive agents, is determined entirely by the state of the environment in which they are situated. First, we summarise the original description of tropistic agents and then reformulate it using the agency framework.

The set of environmental states is denoted by $S$. Since agent perceptions are limited in general, it cannot be assumed that an arbitrary state is distinguishable from every other state. Perceptions thus partition $S$ in such a way that environments from different partitions can be be distinguished whilst environments from the same partition cannot. The partitions are defined by the sensory function, $see$, which maps environments contained in $S$ to environments contained in $T$, the set of all observed environments. The effectory function, $do$, which determines how environments change when an agent performs an action, taken from the set of the agents’ actions, $A$, maps the agent’s action and the current environment to a new environment. Finally, action-selection for a tropistic agent, $action$, is determined by perceptions and maps elements of $T$ to elements of $A$. Tropistic agents are defined by the following tuple:

$$(S, T, A, see : S \to T, do : A \times S \to S, action : T \to A)$$

3.8.2 Reformulating Perception

The agency framework is now applied to reformulate tropistic agents by first defining types: equating the set $S$ to the agency framework type, $Environment$; the set $T$, as it refers to agents’ perceptions, to the type $View$; and the set, $A$, to the type $Action$. We can then write the following type definitions.

$$S == Environment \land T == View \land A == Action$$
According to the agency framework tropistic agents are not autonomous. Thus the agent-level of conceptualisation is the most suitable level. The functions defining architecture at this level are `canperceive`, `willperceive` and `agentactions`, defining the possible percepts, actual percepts and performed actions, respectively. The effect of actions on environments is independent of the level chosen in the agent hierarchy and defined by `effectinteraction`. Recall that these functions have the following type signatures.

- `canperceive : Environment \to P\ Action \to View`
- `willperceive : P\ Goal \to View \to View`
- `agentactions : P\ Goal \to View \to Environment \to P\ Action`
- `effectinteraction : Environment \to P\ Action \to Environment`

These functions include explicit reference to the agents’ goals, which are not represented in the model of tropistic agents since they are implicitly fixed in the hard-coded functions. In what follows, we take the value of these goals to be `gs` and accordingly set all goal parameters of the agency framework functions to this value.

The goals of a tropistic agent do not constrain the selection of its perceptions from those that are available, and `willperceive` is defined as the identity function on observed environments. In the agency framework, the perceiving actions are used at every perceiving step so that the second argument of `canperceive` is always applied to the perceiving actions (`perceivingactions`) of the agents as specified in the `AgentPerception` schema in Section 3.6. Accordingly, tropistic perception is reformulated in the second predicate below. There is an implicit assumption that tropistic agents are capable perceivers; perceptions are always a subset of the actual environment. This assumption is formalised in the last of the three predicates below that together define tropistic perception.

- `willperceive gs = {v : View \cdot (v, v)}`
- `\forall e : S \cdot \text{see } e = \text{willperceive } gs \text{ (canperceive e perceivingactions)}`
- `\forall e, v : S \cdot \text{willperceive } gs \text{ (canperceive e perceivingactions) } \subseteq e`

The set of partitions in `S` can be calculated using set comprehension.

- `partitions == \{ e, v : Environment \mid v = \text{see } e \cdot \text{see } \triangledown \{ v \}\}`

### 3.8.3 Reformulating Action

The difference between the agency framework and tropistic agent effectory functions is simply that the former version allows for a set of actions to be performed rather than a single action.

- `\forall e : Environment; a : Action \cdot (a, e) = \text{effectinteraction } e \{ a \}`

The action selected by a tropistic agent is dependent solely on its perceptions. In the agency framework, the actions performed are additionally dependent on goals and the environment. The environment can affect the performance of selected actions if, for example, an agent has incorrect or incomplete perceptions of it. By contrast it is assumed that a tropistic agent correctly perceives its static environment and performs actions that are equivalent to those selected. These assumptions mean that the environment does not affect the performance of actions once they have been selected. In order to specify this in the agency framework, we fix the `Environment` parameter of `agentactions` to the empty set, and so define `action` using `agentactions` as follows.

- `\forall v : T \bullet \text{action } v = \text{agentactions } gs \{ v \}`
3.8.4 Discussion

Reformulating tropistic agents using the agency framework highlights several issues of note. First, we argue that the agency framework provides a more intuitive conceptualisation of an agent as an object with a purpose. Goals are hard-coded into tropistic agents’ actions and perception functions; they are neither ascribed to the agents nor are there any explicit mechanisms where agents’ goals direct behaviour. Second, explicitly incorporating agents’ goals into the agency framework architectures provides a more sophisticated design environment. It incorporates the premise that agents’ goals change over time and that the selection of actions and perceptions must be adapted accordingly. It is not efficient to have to re-write the functions defining action and perception selection every time new goals are adopted. Third, as the agency framework is more general than described for tropistic agents, we can explicitly formalise any assumptions (implicit or otherwise) regarding the agent, its environment, or the interaction between them.

3.9 Summary

In this chapter we have described a formal model of a four-tiered framework that defines, in decreasing levels of abstraction, entities, objects, agents and autonomous agents, and which we call the agency framework. This has been constructed in such a way that the set of entities in an environment includes the set of objects, the set of objects includes the set of agents, and the set of agents includes the set of autonomous agents. As well as definitions, the framework has provided functional and operational models of the architecture and behaviour of entities at each level in the hierarchy.

An entity is the most abstract description of a component and is modelled as a collection of attributes that are grouped together in some way. An entity with capabilities is an object, which can affect its environment by interacting with it. Agents are objects whose capabilities affect the environment in a way that is purposeful, and can therefore be ascribed goals. Autonomous agents are agents that generate their own goals through non-derivative internal motivations and are consequently capable of independent purposeful behaviour.

The specification structure used to formalise this four-stage hierarchy is shown in Figure 3.3. Components are specified at the highest level of abstraction in the Entity schema and then, through schema inclusion, this is refined to define objects, agents and autonomous agents in the schemas Object, Agent and AutonomousAgent, respectively. For objects, agents and autonomous agents, behaviour is described by the ObjectAction, AgentAction and AutonomousAgentAction schemas. For agents and autonomous agents, we detail their perception in an environment in AgentPerception and AutonomousAgentPerception. Similarly, for entities, objects, agents and autonomous agents we define their state when situated in an environment in EntityState, ObjectState, AgentState and AutonomousAgentState. Since the relationships between the different levels are well-defined, easy movement between them is facilitated. This enables the appropriate level of abstraction to be chosen to describe a given system component. It is then possible, for example, to describe a robot at either the object level or the agent level as appropriate.

Finally, in order to describe interaction, operations are defined that affect the state of components situated in an environment rather than the components themselves, as changes of state to EntityState, ObjectState, and so on, as shown in Figure 3.4.

3.10 Related Work

There exists a small body of work that provides a similar view to that presented here. For example, Covrigaru and Lindsay describe a set of properties that characterise autonomous systems, relating to such factors as type and number of goals, complexity, interaction, robustness, and so on [23]. In contrast, we define what is necessary for a system to be autonomous in very precise terms, and we distinguish clearly between objectness, agency and autonomy. One particular consequence of the difference in views is that we allow a rock, for example, to be considered an agent if it is being used for some purpose, such as a hammer for tent-pegs. Covrigaru and Lindsay deny the rock the quality
Figure 3.3: Schema Structure for Specifying the Agency Framework

Figure 3.4: Schema Structure for Specifying Agency Framework Operation
of autonomy because it is not goal-directed, but ignore the possibility of agency, skipping over an important part of the agency framework.

Castelfranchi shares some of our views in arguing that autonomy is characterised in terms of agents having their own goals, making decisions about these goals and adopting the goals of others only when they choose to [11]. However, these notions are often vague and rely on an existing analysis of the possible social relationships and dependencies between agents. He also considers which aspects of an agent architecture determine whether it is autonomous, and characterises types of autonomy such as social, executive and motivational autonomy [13]. Though Castelfranchi recognises that motivations are significant in descriptions of autonomy, he does not consider them to be sufficient qualities. In addition, these notions of autonomy are relative to his proposed social theory in contrast to our own by which we take autonomy to be an absolute concept that is constant regardless of the context in which it occurs. Furthermore it is not dependent on specific agent architectures. In the agency framework, autonomy either exists or it does not, depending solely on whether an agent has motivations.

Demazeau and Müller [29] discuss the minimal features of a generic agent and also make explicit a conceivable distinction between agents and autonomous agents. In their view, a minimal agent possesses knowledge, goals, a set of possible plans or solutions to achieve these goals, reasoning capabilities to produce these plans and, lastly, decision capabilities to choose which plan to pursue. The goals of an agent can either be implicitly encoded in the algorithms of that agent or explicitly acquired. An autonomous agent is taken to mean an agent whose “existence is not justified by the existence of others.” Some of these ideas are subsumed by the agency framework but also conflict as discussed below.

First, the authors’ definition of a minimal agent is too strong. For example, whilst in many cases agents may be able to plan, it is surely not a necessary characteristic. In addition, no model of how these different aspects of agency are related is provided and, further, the definitions exclude non-computational entities being agents. Our definition of agency on the other hand is more clearly defined and can be more generally applied. For example, it does not rule out non-computational entities as agents. This is important since there are examples of existing agent systems in which non-computational entities are modelled as agents [128]. In fact, we do not distinguish between computational and non-computational agents at all. Programs, robots and cups can all be agents depending on whether they serve a purpose.

Demazeau and Müller’s view of a goal is, however, accommodated within ours, since according to the agency framework, goals can be either explicitly represented and acquired by an agent or implicitly ascribed. Similarly, their definition of autonomy is subsumed by our own, since a motivated agent creates its own goals and does not rely on others for purposeful existence. In addition, the agency framework specifies the precise relationship between agency and autonomy, a question not investigated by Demazeau and Müller, since it can only be settled by definition as has been done in this chapter.

Research by Maruichi et al. [108], with a bias towards agent-oriented programming, takes what at first appears to be a similar view to our own, defining agents in terms of objects and then distinguishing those agents that are autonomous. They model agents as concurrent objects and argue that since these objects automatically execute methods in response to messages without evaluating their meaning or considering their own internal state, they are not autonomous. Autonomy, it is argued, is achieved when an object or agent can control method execution after evaluation of both the meaning of the message and the internal state. However, in that work the authors do not distinguish between objects and agents. Subsequent work by Tokoro offers a related view in which he distinguishes objects, concurrent objects, autonomous agents and volitional agents [162], again similar in spirit to our view, though not so precisely defined or so widely applicable.

More recently, Franklin and Graesser [61] attempt to distinguish agents from programs. In their view, any software agent is a program but a program may not be a software agent. They argue that there is an important distinction between a program and an agent and provide the following definition.

“An autonomous agent is a system situated within and part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to affect what it senses in the future.”

61
This definition does not appear to be at odds with our own but the last remark concerning acting so as to affect what it senses is vague and open to several interpretations and may simply be a consequence of any goal-directed behaviour. Crucially however, the authors have not fulfilled their promise of a definition of agency but have instead moved straight to a definition of an autonomous agent without first considering what it is to be an agent. This is a key aspect of the agency framework.

3.11 Conclusions

As a result of the work in this chapter, we have formal definitions for agents and autonomous agents that are clear, precise and unambiguous. The work is not biased towards any existing classifications or notions because there is no consensus. Recent papers define agents in wildly different ways, if at all, and this makes it extremely difficult to be explicit about their nature and functionality. The agency framework explicates those factors that are necessary for agency and autonomy, and is sufficiently abstract to cover the gamut of agents, both hardware and software, intelligent and unintelligent,

The definitions have been constructed so that they relate to existing work but in such a way as not to specify a prescribed internal architecture. This makes good sense, since it allows a variety of different architectural and design views to be accommodated within a single unifying structure. All that is required by our specification is a minimal adherence to features of, and relationships between, the entities described therein. Thus we allow a cup to be viewed as an object or an agent depending on the manner in which it functions. Similarly, we allow a robot to be viewed as an object, an agent or an autonomous agent depending on the nature of its control structures. We do not specify here how those control structures should function, but instead how the control is directed.

The agency framework provides an important basis for reference. We can classify both human and artificial agents equally well. As an example, consider the relationship of a programmer to a program. Programs are always designed to satisfy goals, but these goals are rarely explicit or able to be modified independently of the programmer. The programs lack goal-generating motivations, but can be ascribed goals. In this respect, they are agents and not autonomous agents. Programmers typically develop programs according to several motivations which determine how the program is constructed. Time and effort must be balanced against cost, ease of use, simplicity, functionality and other factors. Programmers consider these factors in determining the design of programs and in the goal or goals of programs. Programmers can change the goals of programs by modifying code if desired, and can modify their own goals to suit circumstances. In this respect, programmers are autonomous agents.

The difference between these kinds of programs as agents and much recent use of the term is that the relationship between the user (or programmer) and the program has become explicit. Software agents assist users. They adopt the goals of the users in the tasks that they perform. Whether or not they are autonomous depends on the ability of the agents to function independently of those users, and to modify their goals in relation to circumstances.
Chapter 4

Agency Relationships

4.1 Introduction

Whilst agent systems have become increasingly prevalent, being proposed for a number of diverse application areas, the services that they can provide will always be limited by their specific functionality. For example, consider again the example of the household robot and suppose that during the preparation of the evening meal the cooker fails. If fixing the cooker is not in the robot’s capabilities then it will need help from another agent to fix the cooker and, in turn, prepare a meal. Equally, if the cooker is heavy, moving the cooker to diagnose the problem may be outside the capabilities of the repair robot. Its goal of diagnosing the problem can then only be achieved if the domestic robot helps.

More generally, if single-agent systems can cooperate, agents may be able to exploit the capabilities and functionalities of others to achieve individual goals. In addition to the standard advantages of general distributed systems, multi-agent systems can have a much broader application than single-agent systems through the combined functionality of their constituent agents. This is the standard view of multi-agent systems. Durfee, for example, defines a multi-agent system as a collection of problem solvers that “work together” to achieve goals that are beyond the scope of their individual abilities [46]. This notion of agents helping each other, working together or cooperating in some way is common. In a similar way, Huberman and Clearwater characterise multi-agent systems by the interaction of many agents trying to solve problems in a cooperative fashion [83], and Lesser describes a multi-agent system as a computational system in which several semi-autonomous agents interact or work together to perform some tasks or satisfy some goals [99]. There are, however, much broader definitions such as that of Lizotte and Moulin, who describe a multi-agent world as a system in which several agents interact [101].

It is this interaction between individual agents by which goals are typically achieved in these systems [160]. The form of such interaction can range over interleaved actions, combined actions, message-passing or high-level linguistic utterances such as speech acts. The particular form of interaction always depends on the nature of the agents themselves. Thus an account of interaction is only possible through an understanding of the agents involved.

In the previous chapter, we defined agency and autonomy in terms of goals. Agents satisfy goals, while autonomous agents may, additionally, generate them. Goals may be adopted by either autonomous agents, non-autonomous agents or objects without goals, each situation requiring a separate analysis. Since non-autonomous agents satisfy goals for others they rely on other agents for purposeful existence, which indicates that goal adoption creates critical inter-agent relationships. The combined total of these agent relationships defines a social organisation that is not artificially or externally imposed but arises as a natural and elegant consequence of our definitions of agency and autonomy. Moreover, the agency framework allows an explicit and precise analysis of multi-agent systems with no more conceptual primitives than were introduced in the previous chapter for individual agents.

As before, the discussion of multi-agent systems in this chapter is not limited to computational entities, since it is the relationships in which they participate that are significant. For example, if I use a cup to store tea, or a robot to attach a tyre to my car, then the relationships I have with each are im-
important, and I would not wish either of these relationships to be broken or constrained. However, while the components of a multi-agent system need not be computational, it is only computational entities that can recognise and exploit the relationships therein. There are thus two parts to this work. The first is to understand the key inter-agent relationships that arise in multi-agent systems, and the second is to consider how these relationships can be recognised and exploited by computational components. This chapter and the next consider the former, while Chapter 6 considers the latter.

Section 4.2 refines the agency framework and describes other kinds of entity that will be useful for the subsequent discussion. It provides an initial high-level definition of a multi-agent system as a collection of different types of entity. Before goals can be adopted by agents in response to others, goals must be generated and, in the next section, we develop a high-level model of how autonomous agents can generate concrete goals from their abstract motivations. Section 4.4 considers how goals can be adopted by objects, agents and autonomous agents. In the case of an object, this is a case of instantiating the object with some set of goals. However, if the entity is a non-autonomous agent, then it is necessary to consider other agents for which this purpose exists. If the agent is autonomous, a direct appeal can be made to it. The subsequent two sections then consider the obligatory and voluntary relationships that result from goal adoption, which are called engagements and cooperations. Section 4.7 defines a society of agents as the set of agents and the engagements and cooperations in which they are involved, collectively called the agency relationships. Once the society of agents is defined, it is possible to determine a taxonomy of the different ways in which two agents can be related. We refer to relations in this taxonomy as the agency relations taxonomy. Finally, the summary and conclusions are presented in Section 4.9.

In this and subsequent chapters, we use an example based on two autonomous agents called Anne and Bill, who are both librarians and who share an office.

### 4.2 Multi-Agent Systems

Definitions of what constitutes a multi-agent system do not suffer the same diversity as that for agency and autonomy, and providing an encompassing definition is not such an issue. Most authors take a multi-agent system to be a system containing multiple agents (which may be autonomous) interacting with each other to achieve their goals (which may be either global or local to an agent). A definition of what constitutes a multi-agent system is therefore not as critical as definitions for agency and autonomy but we require one, nonetheless, to ground the ideas developed in the rest of this thesis. In particular, we provide a definition that subsumes existing notions and provides sufficient conditions for a system to be categorised as a multi-agent system. This definition is based on previous notions and arranged to apply at the highest level where only system components are identified. Once the inter-agent relationships have been investigated, we refine our definition.

#### 4.2.1 Multi-Agent System Definition

A multi-agent system is one that contains a collection of two or more agents. Since goals cannot exist without having been generated by autonomous agents, it is impossible for agents to exist without autonomy. In addition, therefore, a multi-agent system must contain at least one autonomous agent. (Some have taken multi-agent systems to contain a collection of autonomous agents [115], but this view arises because of the confusion surrounding the concepts of agency and autonomy, as we have seen.) A multi-agent system may contain multiple autonomous agents but this is not necessary. One further requirement for a set of agents to be a multi-agent system is that there must be some interaction between the agents, since otherwise each would be acting independently of the others and it would make little sense to consider such a collection of components as one system. Moreover, this interaction must result when one agent satisfies the goals of another.

**Definition** A multi-agent system is any system that contains

1. two or more agents;
2. at least one autonomous agent; and
3. at least one relationship between two agents where one satisfies the goal of the other.

This definition contains elements of all of the definitions considered in the introduction, but is based on the notions of the previous chapter, providing a precise description of a multi-agent system.

4.2.2 Multi-Agent System Specification

At the most basic level therefore, a multi-agent system contains agents and autonomous agents. However, since agents can be instantiated from objects, and objects from other entities, we include all four hierarchical categories in the formal specification below, which captures how different classes of entity are related. The first predicate of the $\text{MASystem}$ schema states that $\text{autonomousagents}$ is a subset of $\text{agents}$, $\text{agents}$ is a subset of $\text{objects}$, and $\text{objects}$ is a subset of $\text{entities}$. The rest of the schema requires there to be at least two agents of which one is autonomous, and that at least two of the agents in the system share a minimum of one goal. It provides a formal realisation of the multi-agent system definition above.

$$\begin{align*}
\text{MASystem} & \\
entities : & P Entity \\
objects : & P Object \\
agents : & P Agent \\
autonomousagents : & P AutonomousAgent
\end{align*}$$

$$\begin{align*}
\text{autonomousagents} & \subseteq \text{agents} \subseteq \text{objects} \subseteq \text{entities} \\
\#\text{agents} & \geq 2 \\
\#\text{autonomousagents} & \geq 1 \\
\exists \text{aa1, aa2 : agents} & \land \text{aa1.goals} \cap \text{aa2.goals} \neq \{\}
\end{align*}$$

4.2.3 Server-Agents and Neutral-Objects

In what follows, it will be useful to identify further sub-categories of entity. Before proceeding, therefore, we distinguish those objects that are not agents, and those agents that are not autonomous, and refer to them as $\text{neutral-objects}$ and $\text{server-agents}$ respectively. They are defined in the following schemas.

$$\begin{align*}
\text{NeutralObject} & \\
\text{Object} & \\
goals & = \{\} \\
motivations & = \{\}
\end{align*}$$

$$\begin{align*}
\text{ServerAgent} & \\
\text{Agent} & \\
motivations & = \{\}
\end{align*}$$

It is now possible to elaborate the previous description. A multi-agent system contains a collection of entities of which some are objects; of these objects some are neutral-objects and some are agents; and of these agents some are server-agents and some are autonomous agents. This is shown in a modified Venn diagram in Figure 4.1. A term written across a boundary signifies the set of elements contained within it.
We can correspondingly refine the $\textit{MASystem}$ schema as follows. The variable, $\textit{neutralobjects}$, is the set of all objects that are not serving any purpose, and the $\textit{serveragents}$ variable is the set of all agents that are not autonomous. The predicates state that the set of all agents is the union of all autonomous agents and all server-agents, and that the set of all objects is the union of all agents and all neutral-objects.

\[
\begin{align*}
\textit{MultiAgentSystem} & \quad \textit{MASystem} \\
& \quad \textit{neutralobjects} : \mathcal{P} \textit{NeutralObject} \\
& \quad \textit{serveragents} : \mathcal{P} \textit{ServerAgent} \\
& \quad \textit{agents} = \textit{autonomousagents} \cup \textit{serveragents} \\
& \quad \textit{objects} = \textit{agents} \cup \textit{neutralobjects}
\end{align*}
\]

In order to construct a multi-agent system, however, one agent must adopt the goal of another. Before this can happen, the goal must first be created. In the next section, we discuss how goals are generated by autonomous agents from their motivations.

### 4.3 Goal Generation

An account of how autonomous agents generate goals from their motivations is an important topic of investigation in the design of autonomous agents and others have looked at specific computational architectures for achieving this aim [113, 124]. Though we do not address particular architectures or mechanisms involved in the generation of goals, we do consider a high-level functional description that can be refined to describe particular architectures. The model that follows is therefore not architecture-specific but simply illustrates how the control of goal generation is directed. It is important to understand the original generation of goals since it is their creation and subsequent dispersal that defines the set of inter-agent relationships that are considered subsequently in this chapter.

For an agent to assess the relative benefit of generating one goal over others, it must have a repository of known goals that we refer to as a \textit{goal library} which may be fixed, but is more likely to be updated through learning. These goals capture knowledge of limited and well-defined aspects of the world by describing particular \textit{sub-states} of environments. An autonomous agent thus tries to find a way to mitigate motivations, either by selecting an action to achieve an existing goal, or by retrieving
a goal from the goal library. The first of these alternatives was addressed by the autoactions function in the AutonomousAgentAction schema seen earlier, while the second is considered here.

In order to retrieve goals to mitigate motivations, an autonomous agent must have some way of assessing the effects of competing or alternative goals. Clearly, the goals that make the greatest positive contribution to the motivations of the agent should be selected unless a greater motivational effect can be achieved by destroying some subset of its goals. The motivational effect of generating or destroying goals is not only dependent on the current motivations but also on the current goals of the agent and its current perceptions. For example, it is sensible to require that an autonomous agent should not generate a goal that is not compatible with the achievement or satisfaction of its existing goals, and neither should it generate a goal that is already satisfied in the current environment.

Once generated, it is possible that goals may be changed or modified if it seems appropriate to do so in the light of any changes to the environment or unexpected problems in the goal-achievement process. Goals can thus be dynamically re-evaluated as a result of considering the implications of the original and alternative goals in more detail, in terms of the actions required to achieve them.

Formally, the ability of autonomous agents to generate their own goals is specified in the schema, AssessGoals, which describes how autonomous agents monitor their motivations for goal generation. First, the AutonomousAgentState schema is included. The new variable, goallibrary, is declared to represent the repository of available known goals. The motiveeffectgenerate function returns a numeric value representing the motivational effect of satisfying a new set of additional goals with a set of motivations, current goals and current perceptions. Similarly, the motiveeffectdestroy function returns a numeric value representing the motivational effect of removing some subset of existing goals with the same set of motivations, goals and perceptions. The predicate part specifies that the current goals must be in the goal library. For ease of expression, we define a function related to motiveeffectadd called satisfyadd, which returns the motivational effect of an autonomous agent satisfying an additional set of goals. The function satisfydestroy, is analogously related to motiveeffectdestroy.

\[
\begin{align*}
\text{AssessGoals} & \\
\text{AutonomousAgentState} & \\
goallibrary & : P \text{Goal} \\
motiveeffectgenerate & : P \text{Motivation} \rightarrow P \text{Goal} \rightarrow \text{View} \rightarrow P \text{Goal} \rightarrow \mathbb{Z} \\
motiveeffectdestroy & : P \text{Motivation} \rightarrow P \text{Goal} \rightarrow \text{View} \rightarrow P \text{Goal} \rightarrow \mathbb{Z} \\
satisfygenerate, satisfydestroy & : P \text{Goal} \rightarrow \mathbb{Z} \\
\end{align*}
\]

\[
\begin{align*}
goals \subseteq \text{goallibrary} & \\
\forall gs : P \text{goallibrary} & \bullet \\
satisfygenerate gs = \text{motiveeffectgenerate motivations goals actualpercepts gs} & \wedge \\
satisfydestroy gs = \text{motiveeffectdestroy motivations goals actualpercepts gs} & \\
\end{align*}
\]

The GenerateGoals operation schema formally describes the generation of a new set of goals, which changes the state of the agent, indicated by \( \Delta \text{AutonomousAgentState} \). The remaining part of the schema states that there is a set of goals in the goal library that has a greater motivational effect than any other set of goals, and the current goals of the agent are updated to include the new goals. After this operation, the variables, goallibrary, motiveeffectgenerate, motiveeffectdestroy and satisfygenerate are unchanged but this is not specified for reasons of brevity.

\[
\begin{align*}
\text{GenerateGoals} & \\
\Delta \text{AutonomousAgentState} & \\
\text{AssessGoals} & \\
\exists gs : P \text{Goal} \mid gs \subseteq \text{goallibrary} & \bullet \\
(\forall os : P \text{Goal} \mid os \in (P \text{goallibrary}) & \bullet \\
(satisfygenerate gs \geq satisfygenerate os) & \wedge goals' = goals \cup gs) & \\
\end{align*}
\]
Once generated by an autonomous agent, goals persist in the relevant multi-agent system until, for whatever reason, they are explicitly destroyed by that autonomous agent. The destruction of goals is defined analogously to the generation of goals in the $\text{DestroyGoals}$ schema, which states that an agent destroys the subset of its goals that provide the greatest motivational advantage for doing so.

$$
\begin{align*}
\text{DestroyGoals} \\
\vartriangle \text{AutonomousAgentState} \\
\text{AssessGoals} \\
\exists gs : \mathcal{P} \text{Goal} \mid gs \subseteq \text{goallibrary} \cdot \\
(\forall os : \mathcal{P} \text{Goal} \mid os \in (\mathcal{P} \text{goallibrary}) \cdot \\
(satisfy\text{destroy} gs \geq satisfy\text{destroy} os) \land \text{goals}^d = \text{goals} \setminus gs)
\end{align*}
$$

This section has provided a high-level description of how goals are generated from motivations. In particular, we have considered how the current perceptions, goals and motivations of an autonomous agent affects the generation of new goals, or the destruction of existing ones. Since goals can only be generated by autonomous agents a multi-agent system must contain at least one autonomous agent. However, in attempting to achieve their goals, autonomous agents may attempt to involve others, which is considered next.

4.4 Goal Adoption

Agents can make use of the capabilities of others if these others adopt their goals. For example, if Anne needs to move a table which cannot be lifted alone, she must get someone else to adopt her goal before it can be moved. Similarly, if she wants tea, then she must make use of a kettle to boil water, a teapot to make the tea and subsequently a cup from which to drink it. Each of these objects can be ascribed, or viewed, as adopting Anne’s goals in order that her thirst can be relieved.

In general, entities may serve the purposes of others by adopting their goals. Since they must have capabilities they must be objects. In the discussion that follows, therefore, we consider only objects. However, the ways in which objects adopt the goals of others depends on the kind of object. An object is either a neutral-object, a server-agent or an autonomous agent, and each category requires a separate analysis by the agent with a goal to be adopted, which we call the viewing agent.

- A neutral-object serves no purpose and can be instantiated as an agent by the viewing agent without regard to others. Here, agents are created from objects with the addition of relevant associated goals.

- A server-agent is not isolated since it must already be serving a purpose for one or more other agents. An additional goal may affect the achievement of existing goals so for goal adoption to take place, the viewing agent must analyse both the server-agent and the agents that are engaging it in order to avoid conflict.

- An autonomous agent generates its own goals, adopting the goals of a viewing agent either by recognising that goal or by negotiation with the viewing agent or some intermediary agent. The viewing agent must therefore consider the motivations of the autonomous agent to determine whether it will adopt the goal of the viewing agent.

These three scenarios are illustrated in Figure 4.2. A target agent or object is one that is intended to adopt goals, an engaging agent is one whose goals are currently (already) adopted by the target agent, and a viewing agent is an agent that seeks to engage a target agent or object by having it adopt goals. (The engagement relationship between agents is examined in more detail in the next section.) It is a viewing agent because the way in which goal adoption is attempted is determined by its view of the situation. (We consider the dimensions required by viewing agents in Chapter 6.) In (a), the viewing agent must analyse the target neutral-object in order for the neutral-object to adopt the goal of the
viewing agent. Second, in (b) the viewing agent must analyse the target server-agent and any agents which are engaging the target agent. Third, in (c) the viewing agent analyses the target autonomous agent only. Each of these is considered below.

4.4.1 Goal Adoption by Neutral-Objects

The simplest case involves instantiating an agent from a neutral-object with the goals to be adopted. The act of goal transfer causes an agent to be created from a neutral-object. Thus, for example, a cup in Anne and Bill’s office, which is just an neutral-object, becomes an agent when it is used for storing Anne’s tea. In this case it adopts or is ascribed her goal of storing liquid. It is possible to create the agent from the object because the cup is not being used by anyone else; it is not engaged by another agent. Neutral-objects are not engaged but disengaged.

Below, we specify a function that creates a new entity by ascribing a set of goals to an existing entity. It takes an entity and a set of goals, gs, and creates a new entity with the same set of attributes, actions and motivations but with the additional goals, gs. The function is total and is thus valid for any entity and any set of goals.

EntityA doptGo als : (Entity × P Goal) → Entity

∀ gs : P Goal; old, new : Entity ⊨
EntityA doptGo als(old, gs) = new ⇔ new.goals = old.goals ∪ gs
∧ new.capabilities = old.capabilities ∧ new.attributes = old.attributes

When a neutral-object adopts a set of goals it becomes a server-agent, as specified below. As described previously, a variable with a ‘?’ in the Z notation indicates an input. Thus, a neutral-object and a set of goals are inputs, the entities in the multi-agent system change, indicated by the term \( \triangle MultiAgentSystem \), and the sets of objects and agents are updated accordingly. In addition, the variables, entities, objects and agents, are updated by removing the neutral-object and adding the newly instantiated server-agent. The final three predicates are redundant since they follow from the previous predicates, but are included to detail how all the state variables are affected. In subsequent schemas, such redundancy is not included.
For completeness, we specify the related operation where an entity is released from all of its agency obligations. Here, a server-agent reverts from a server-agent to a neutral-object. It is possible that a cup can be engaged for two separate goals. For example, it may be engaged as a vase for flowers and as a paper-weight for loose papers if there is a breeze coming from a nearby open window. If the window is closed and the flowers are removed, the cup is released from all its agency obligations and reverts to being a neutral-object.

Formally, this operation is defined by the RevertToNeutralObject schema. It uses the axiomatic function, EntityRemoveGoals, defined similarly to EntityAdoptGoals, which removes a set of goals from an entity.

\[
\text{EntityRemoveGoals} : (\text{Entity} \times \mathbb{P} \text{Goal}) \rightarrow \text{Entity}
\]

\[
\forall gs : \mathbb{P} \text{Goal}; \ old, new : \text{Entity} \quad \text{EntityRemoveGoals}(old, gs) = new \Leftrightarrow new.goals = old.goals \setminus gs \quad \wedge \ new.capabilities = old.capabilities \wedge new.attributes = old.attributes
\]

The predicates in the following schema check that the input goals are the same as the set of current goals of the server-agent. This ensures that the server-agent is released from all of its agency obligations. The variables, neutralobjects, serveragents, and autonomousagents, are updated accordingly.

\[
\text{RevertToNeutralObject}
\]

4.4.2 Goal Adoption by Server-Agents

If the target object is engaged by other agents then it is itself an agent, so the protocol for goal adoption changes. In this case, there are alternative ways to engage the target object.

The first involves supplying the target object with more goals that does not affect the existing agency obligations. In this case the agent is shared between the viewing agent and the existing engaging agents. The second involves trying to persuade any engaging agents to release the engaged object
so that it becomes a neutral-object and can therefore subsequently be engaged by the viewing agent as required. (This may relate to the issue of goal adoption for autonomous agents, which is considered later). The third possibility involves displacing the engaging agent so that the engaged object becomes a neutral-object and can then subsequently be ascribed other goals. This possibility is dangerous since it may cause conflict with the previous engaging agents.

As an example, suppose that a cup is currently in use as a paper-weight for Anne, so that the cup is Anne’s agent with her goal of securing loose papers. Suppose also, that Bill wishes to use the cup to have some tea. The first way for Bill to engage the cup is for him to attempt to use the cup without destroying the existing agency relationship between Anne and the cup. Since this would involve an awkward attempt at making tea in, and subsequently drinking from, a stationary cup, he may decide instead to try other alternatives. The second alternative is to negotiate with Anne to release the cup so that it can be used for storing tea while the third alternative is for Bill to displace the goal ascribed to the cup by removing the cup from the desk and pouring tea into it. The cup is no longer an agent for Anne and is now ascribed the goal of storing tea for Bill. It has switched from being engaged by Anne to being engaged by Bill, and this is equivalent to the agent reverting to an object and then being re-instantiated as a new agent. This method may not be an appropriate strategy, however, because in destroying the agency obligation of the cup as a paper-weight, there is a risk of conflict between Anne and Bill.

The adoption of goals by server-agents is formalised in the next schema, in which a server-agent is ascribed an additional set of goals. This describes the alternative where the cup is serving as a paperweight and is then subsequently given the goal of storing flowers. The initial target agent is removed from the set of server-agents which is further updated to include the newly-instantiated target agent, while the preconditions ensure that the target agent is a current system server-agent in the system and that the new goals are distinct from the existing goals.

---

```plaintext
ServerAgentAdoptGoals

\[sa? : ServerAgent\]
\[gs? : \wp \text{Goal}\]
\[\triangle \text{MultiAgentSystem}\]

\[sa? \in \text{serveragents}\]
\[gs? \cap sa?.goals = \{\}\]
\[\text{neutralobjects}' = \text{neutralobjects}\]
\[\text{serveragents}' = \text{serveragents} \setminus \{sa?\} \cup \{\text{EntityAdoptGoals}\ (sa?, gs?)\}\]
\[\text{autonomousagents}' = \text{autonomousagents}\]
```

In some situations, a server-agent is released from some but not all of its agency obligations. Suppose, for example, that a window is open in the librarians’ office and that a cup is being used as a paperweight by Anne and a vase by Bill. If the window is subsequently closed, then the cup may be released from its agency obligations as a paperweight but still remain an agent because it is holding flowers. Formally, the operation schema representing this change of agency obligation is specified in the next schema. Notice that the goals that are removed from an agent in this operation must be a proper subset of its goals. The server-agent, sa?, is removed from the set of server-agents and replaced with the agent that results from removing the goals, gs?, from a?.
4.4.3 Autonomous Goal Adoption

In the example above, the second possibility for goal adoption by server-agents involves Bill persuading Anne to first release the cup from its existing agency. The cup would then become a neutral-object and could be instantiated as required by Bill. In general, such persuasion or negotiation may be more difficult than the direct physical action required for goal adoption in non-autonomous entities. Autonomous agents are motivated and as such, only participate in an activity and assist others if it is to their motivational advantage to do so. They create their own agendas and for them, goal adoption is a voluntary process as opposed to an obligatory one for non-autonomous agents. In a similar example, Anne might ask Bill to assist in moving a table, but Bill may refuse.

Formally, the operation of an autonomous agent adopting the goals of another is specified in the following schema where the set of autonomous agents is updated to include the newly instantiated target autonomous agent. Note that this does not detail the persuasion involved, but simply the state change resulting from the goal adoption.

In general, goals must be adopted through explicit autonomous agent initiative, as opposed to an ascription of goals for non-autonomous agents. However, in some contexts the ascription of goals to autonomous agents may be meaningful. Suppose, as a dramatic yet unlikely example, that Anne incapacitates Bill in some way and places him by the door to function as a draft excluder. In this situation, the autonomous agent, Bill could be ascribed the goal of keeping out the draft even though he has not explicitly adopted this goal. Such cases can be described by considering the autonomous agent as an agent in an obligatory relationship. In this thesis, however, we restrict autonomous goal adoption to the explicit and voluntary generation of goals that have been recognised in others.

Several points arise as a consequence of autonomous agents having to adopt goals.

- An autonomous agent can only explicitly adopt the goals of another if these can be represented in its goal library. For example, consider an agent whose goal base consists entirely of formulae in predicate calculus. If the goal base can represent predicates of the form $On(x, y)$ but not of the form $NextTo(x, y)$ then it can adopt the goal $On(\text{block}_A, \text{block}_B)$ but not the goal $NextTo(\text{block}_A, \text{block}_B)$.
In order for an autonomous agent to adopt a goal, it must generate this goal through its motivations so that autonomous goal adoption is, in fact, a special case of goal generation. The generated goal is first recognised and then considered in terms of its motivational effect.

Autonomous agents only adopt the goals of another if, at that time, this adoption provides the greatest motivational effect compared with the generation of any other possible set of goals.

4.4.4 Autonomous Goal Destruction

For a number of reasons an autonomous agent may destroy adopted goals. For example, suppose Anne wishes to move a table and has persuaded Bill to help. If Anne subsequently destroys some important agency relationship of Bill’s, it is possible that Bill may then destroy the goal he has adopted from Anne of moving the table. As with goal adoption, for an autonomous agent to destroy goals, this must be considered the most motivationally beneficial course of action. This scenario is formalised below and is similar to the previous schema.

AutonomousAgentDestroysGoals

\[
\begin{align*}
aa &: AutonomousAgent \\
gs &: P \text{ Goal} \\
\Delta &: MultiAgentSystem \\
AssessGoals
\end{align*}
\]

\[
\begin{align*}
\text{aa} &\in \text{autonomousagents} \\
\text{gs} &\subseteq \text{aa}.goals \\
\text{autonomousagents}' &= (\text{autonomousagents} \setminus \{\text{aa}\}) \cup \\
&\quad \{\text{EntityRemoveGoals}(\text{aa}, \text{gs})\} \\
\text{agents}' &= \text{agents} \land \text{objects}' = \text{objects}
\end{align*}
\]

Figure 4.3 gives an overview of the schemas used to define a multi-agent system as a collection of different categories of entity. The MASystem schema is defined using the four entity schema types of the agency framework, while the MultiAgentSystem schema includes this, but also introduces neutral-objects and server-agents. There are then six operation schemas describing goal adoption that change the state of this schema.

4.5 Engagement

Now, whenever an agent uses another non-autonomous entity, there is a relationship between the agent and the entity. In keeping with existing work concerning the nature of interdependencies between agents we consider agency relationships as specific kinds of social relationship [11], which must be recognised by computational components. For example, if Bill is collecting dirty cups for washing, it is important how he views the relationship of Anne to each cup. If she is using a cup, then there is a relationship and the cup should remain, otherwise it can be removed.

4.5.1 Direct Engagement

The social relationship between a non-autonomous entity that has adopted the goals of another agent and that agent is termed an engagement, and the agent is said to be engaging the entity. If Anne is using the cup to drink tea then she is said to be engaging the cup. In addition, Anne is engaging the cup through direct intervention. We refer to such a relationship as a direct engagement. This distinguishes direct engagements from other engagements in which there are intermediary agents.

A direct engagement takes place whenever a neutral-object or a server-agent adopts the goals of another. Thus, an agent with some goals, called the client, uses another agent, called the server,
to assist them in the achievement of those goals. Note that according to our previous definition, a server-agent is non-autonomous, and either exists already as a result of some other engagement, or is instantiated from a neutral-object for the current engagement. There is no restriction placed on a client-agent, and it may be autonomous or non-autonomous.

**Definition** A direct engagement between an agent and a server-agent exists when, through the direct intervention of the first agent, the server-agent has adopted a goal of the agent.

The following schema provides a formal definition of a direct engagement, in which there is a client agent, \( \text{client} \), a server-agent, \( \text{server} \), and the goal that \( \text{server} \) satisfies for \( \text{client} \). An agent cannot engage itself, and both agents must have the goal of the engagement.

\[
\text{DirectEngagement} \\
\text{client : Agent} \\
\text{server : ServerAgent} \\
\text{goal : Goal} \\
\text{client} \neq \text{server} \\
\text{goal} \in (\text{client.goals} \cap \text{server.goals})
\]

### 4.5.2 Direct Engagements in a Multi-Agent System

A multi-agent system may contain many engagements between individual agents. Formally, the set of all direct engagements in a multi-agent system is represented by the \( \text{dir engagements} \) variable in the \text{SystemEngagements} schema. The first predicate states that for any direct engagement in \( \text{dir engagements} \), there can be no intermediate direct engagements of the same goal. In addition, the set of all server-agents, \( \text{serveragents} \), is equal to the set comprising all server-agents involved anywhere in \( \text{dir engagements} \). The final predicate states that those agents that are part of some engagement are a subset of the set of all agents.
4.5.3 Engagement Chains

Once autonomous agents have generated goals and engaged other server-agents, these server-agents may, in turn, engage other non-autonomous entities with the purpose of achieving or pursuing the original goal. This process can then, in principle, continue indefinitely. For example, the librarian Anne (A) may generate the goal of finding the location of a library book, and if A engages a workstation (W) to run a program (P) to search a database (D) for this information, there is a direct engagement between Anne and the workstation, between the workstation and the program, and between the program and the database, all with the goal of locating the book as illustrated in Figure 4.4. These chains of engagement provide more information with which to analyse multi-agent systems than using engagements alone, since the flow of goal adoption is explicitly represented. We call these features engagement chains. An engagement chain thus represents the goal and all the agents involved in the sequence of direct engagements. More specifically, it represents a sequence of direct engagements. Since goals are grounded by motivations, the agent at the head of the chain must be autonomous.

**Definition** An engagement chain consists of a goal, the autonomous agent that generated that goal, and a non-empty sequence of server-agents such that (i) the autonomous agent engages the agent at the head of the chain with respect to the goal and, (ii) each agent in the sequence directly engages the next with respect to the goal.

Formally, an engagement chain comprises a goal, goal, the autonomous client-agent that generated the goal, autoagent, and a sequence of server-agents, agentchain, where each agent in the sequence is directly engaging the next. For any engagement chain, there must be at least one server-agent and the same agent cannot be involved more than once, so that agentchain is represented as a non-empty
injective sequence. The predicate states that all the agents involved must share the goal of the chain. Note that the autonomous agent may have generated the goal by adopting it from another autonomous agent.

```
EngagementChain
  goal : Goal
  autoagent : AutonomousAgent
  agentchain : iseq1 ServerAgent

  goal ∈ autoagent.goals
  goal ∈ ∩{s : ServerAgent | s ∈ ran agentchain • s.goals}
```

4.5.4 Engagement Chains in a Multi-Agent System

In general, a multi-agent system contains many engagement chains. For any two consecutive agents in an engagement chain there must be a corresponding direct engagement between those two agents with the goals of the chain.

Formally, we represent the set of all such engagement chains by the engchains variable in the SystemEngagementChains schema. This set can be related to direngagements, the set of direct engagements in SystemEngagements as follows. For every engagement chain, ec, there must be a direct engagement between the autonomous agent, ec.autoagent, and the first client, head ec.agentchain, of ec with respect to the goal of ec. There must also be a direct engagement between any two agents that are adjacent to each other in ec.agentchain with respect to ec.goal. Given two sequences, s and t, of the same type, the relation, s in t, holds when s is a subsequence of t. The definition of the subsequence relation can be found in Appendix A.4. In addition, autonomousagents can be related to the set, engchains, since the autonomous agents involved in an engagement chain are a subset of the set of all autonomous agents of the system.

```
SystemEngagementChains
  SystemEngagements
  engchains : ℙ EngagementChain

  ∀ ec : engchains; s1, s2 : Agent •
  (∃ d : direngagements • d.goal = ec.goal ∧ d.client = ec.autoagent ∧
  d.server = head (ec.agentchain)) ∧
  ⟨s1, s2⟩ in ec.agentchain ⇒ (∃ d : direngagements •
  d.client = s1 ∧ d.server = s2 ∧
  d.goal = ec.goal)

  {ec : engchains • ec.autoagent} ⊆ autonomousagents
```

Engagement chains provide a means of categorising the relationships between agents. An understanding of engagements alone rather than engagement chains would not, in general, be sufficient to understand why the relationship exists. Engagement chains capture the flow of control from agent to agent and the understanding that every direct engagement must eventually be traced back to an autonomous agent. Understanding a program engaged by a workstation with the goal of locating a book is important, but may not be sufficient. What is crucial is the sequence of direct engagements that led to the creation of this direct engagement, the autonomous agent who generated the original goal, and the motivations that led to the goal’s original generation.
4.6 Cooperation

The server-agents in the engagement relationships of the previous section have no means of resisting these relationships, so that they are obligatory. Autonomous agents, by contrast, generate their own goals and decide when to adopt the goals of others. Any relationship entered into by an autonomous agent is therefore voluntary. We refer to voluntary relationships as cooperations. An autonomous agent is said to be cooperating with another autonomous agent, if that agent has adopted the goal or goals of the other. The notion of autonomous goal acquisition applies both to the origination of goals by an autonomous agent for its own purposes, and the adoption of goals from others. For autonomous agents, the goal of another can only be adopted if it has a positive motivational effect, and this is also exactly why and how cooperations originate. Thus, as we have stated previously, goal adoption is just a special case of goal generation, where goals are generated in response to recognising them in others.

Thus we reserve the term, cooperation, for use only when the parties involved are autonomous and potentially capable of resisting. The difference between engagement and cooperation is in the autonomy or non-autonomy of the entities involved. It is senseless, for example, to consider a workstation cooperating with its user, but meaningful to consider the user engaging the workstation. Similarly, while it is not inconceivable for a user to engage a secretary, it makes better sense to say that the secretary is cooperating with the user, since the secretary can withdraw assistance at any point.

For example, if Anne and Bill both independently discover a broken library shelf and in response, both independently generate the goal of reporting it, then they are not cooperating. Certainly, both agents have the same goal but neither has adopted the goal of the other and neither agent can be said to be cooperating with the other. Cooperation cannot occur unwittingly if, by chance, two autonomous agents both generate the same goal. An agent can only cooperate with another if it has first recognised a goal in that other and — partly as a result of that recognition — has generated that goal for itself.

**Definition** An agent, $A$, is said to cooperate with another agent, $B$, if they are both autonomous, and $A$ has autonomously adopted the goal of $B$.

Note, that this definition entails that cooperation is not symmetric. That is, if $A$ is cooperating with $B$ then it does not follow that $B$ is cooperating with $A$.\(^1\)

**Definition** A cooperation comprises a goal, the autonomous agent who generated the goal, and the non-empty set of autonomous agents who adopted the goal from the original autonomous agent.

The schema below defines a cooperation to be a goal, $goal$, the autonomous agent that generated the goal, $generatingagent$, and the non-empty set of autonomous agents who adopted the goal, $cooperatingagents$. The predicates assert that all involved agents have the cooperation goal and that agents cannot cooperate with themselves.

<table>
<thead>
<tr>
<th>Cooperation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$goal: Goal$</td>
</tr>
<tr>
<td>$generatingagent : AutonomousAgent$</td>
</tr>
<tr>
<td>$cooperatingagents : P_1 AutonomousAgent$</td>
</tr>
</tbody>
</table>

$goal \in generatingagent.goals$

$\forall aa : cooperatingagents \bullet goal \in aa.goals$

$generatingagent \notin cooperatingagents$

Cooperation therefore arises through autonomous agents acting in their self-interest. Some authors have proposed explicit mechanisms for this to occur including Ito and Yano [84] who show how cooperation can arise from collections of agents trying to maximise benefit to themselves.

\(^1\)This definition of cooperation is quite different from the standard, symmetrical one, which typically includes references to “working together”. However, we use this word deliberately to make the point more forcibly that whilst the concept of “cooperation” can be a symmetrical process it is, in our view, not necessarily so. For example, suppose that Anne is expecting an academic to visit later in the day and starts to tidy the office in preparation. If Bill recognises Anne’s goal to tidy the office he may decide to adopt her goal and help. In some cases Anne may not even be aware that Bill has adopted her goal and in this case, whilst Bill is cooperating with Anne, Anne cannot be said to be cooperating with Bill.
4.6.1 Cooperations in a Multi-Agent System

The set of cooperations in a multi-agent system is defined by the `cooperations` variable in the schema, `SystemCooperations`. The predicate part of the schema states that for any cooperation, the union of the cooperating agents and the generating agent of that cooperation are a subset of the set of all autonomous agents that have that goal. The subset relation is used rather than the equality relation because two agents sharing a goal are not necessarily cooperating. In addition, the set of all agents involved as cooperating agents in a cooperation is a subset of (rather than is equal to) the set of all autonomous agents of the system since not all autonomous agents are necessarily involved in cooperations.

\[
\begin{align*}
\text{SystemCooperations} \\
\text{MultiAgentSystem} \\
\text{cooperations} : \mathbb{P} \text{ Cooperation} \\
\forall c : \text{cooperations} \bullet (c.\text{cooperatingagents} \cup \{c.\text{generatingagent}\}) \subseteq \\
\{a : \text{autonomousagents} \mid c.\text{goal} \in a.\text{goals} \bullet a\} \\
\cup \{c : \text{cooperations} \bullet (c.\text{cooperatingagents} \cup \{c.\text{generatingagent}\}) \subseteq \\
\text{autonomousagents}
\end{align*}
\]

4.6.2 Discussion and Example

Suppose that Bill informs Anne that he needs to borrow a car, and further suppose that Anne wishes to help Bill achieve his goal but is not in a position to lend her car because it is in a garage awaiting repair. If she adopts the goal to provide a car that Bill can borrow, she may ask another friend, Charlie, to borrow his car. Now, if Anne does not tell Charlie that the car is intended for Bill, then Anne is cooperating with Bill, and Charlie is cooperating with Anne, but Charlie is not cooperating with Bill. This can be represented by two cooperations, `coop1` and `coop2`, as follows.

\[
\begin{align*}
\text{coop1.\text{generatingagent}} &= \text{Bill} & \text{coop2.\text{generatingagent}} &= \text{Anne} \\
\text{coop1.\text{cooperatingagents}} &= \{\text{Anne}\} & \text{coop2.\text{cooperatingagents}} &= \{\text{Charlie}\} \\
\text{coop1.\text{goal}} &= \text{Bill\_use\_car} & \text{coop2.\text{goal}} &= \text{Anne\_use\_car}
\end{align*}
\]

Alternatively, if Anne informs Charlie that the car is intended for Bill, then Charlie can be seen to be cooperating with Bill. This is represented as follows using the one cooperation, `coop`.

\[
\begin{align*}
\text{coop.\text{generatingagent}} &= \text{Bill} \\
\text{coop.\text{cooperatingagents}} &= \{\text{Anne, Charlie}\} \\
\text{coop.\text{goal}} &= \text{Bill\_use\_car}
\end{align*}
\]

The definitions provided here state necessary conditions for cooperation. An agent must be autonomous and adopt the goal of another for cooperation to ensue. Further categories of goal adoption can be seen as refinements to this basic notion. For example, according to Conte et al. [21], cooperation includes a notion of goal adoption but also stipulates that agents should depend on each other for the shared goal. The authors also define three types of cooperation known as accidental, unilaterally intended and mutual cooperation. However, their definition requires a notion of dependence on each other for the goal to be achieved. In our view, Anne can cooperate with Bill even if Bill could achieve his goal independently of Anne.

Our definition of cooperation also provides necessary conditions for a group of entities becoming what Castelfranchi et al. call a collective entity [12, 14]. In addition to sharing a common goal, each agent in such a collective is required ‘to do its share’ to achieve the common goal of the group. In this way, a collective entity can be viewed as a refinement of a set of cooperative autonomous agents as described above. Similarly, Hirayama and Toyada [80] consider coalitions amongst agents who work together to benefit members of the coalition only.
The distinction between autonomous and non-autonomous agents enables us to detail precisely the nature of the relationships between agents. Autonomy implies independence and freedom to make decisions according to an agent’s own priorities and also to decide with whom to cooperate. By contrast, non-autonomous agents are naturally disposed to adopt the goals of others whenever possible, and typically have no choice in the matter. They are therefore benevolent agents.

The distinction between engagement and cooperation is similar to the distinction made between vertical cooperation and horizontal cooperation proposed by Fischer et al. [55, 57], which are two particular cooperation settings arising from different possible configurations of agents. They use a transportation domain as an example, where the former setting is a standard protocol for task allocation between a shipping company and its trucks, whilst the latter is a protocol for negotiation between companies that takes into account the fact that other companies are autonomous and therefore self-interested entities. Clearly, the trucks of a shipping company can be considered as server-agents for that company and are thus naturally disposed to adopt the goals of the company so that they can simply be engaged. In the latter case, negotiation must take place since autonomous companies need to cooperate with each other. The model proposed in this chapter provides a means of analysing these relations so that the form of the interaction or negotiation can be reasoned about in advance.

4.7 The Agency Society

We refer to engagements, engagement chains and cooperations collectively as the agency relationships, and use the term agency society to refer to the set of all entities and agency relationships in a multi-agent system. Formally, using the definitions of engagement, engagement chains and cooperations, we can now refine our existing multi-agent system definition, \( \text{MultiAgentSystem} \), from Section 4.2 to define the agency society. This defined a multi-agent system as a collection of entities, objects, agents and autonomous agents with the proviso that at least one entity was an autonomous agent and there were at least two agents in the system. To this we can add the set of engagements, engagement chains and cooperations between entities, which define the entire set of agent relationships that arise from the agency framework.

As previously stated, a multi-agent system must contain at least one relationship in which one agent satisfies the goal of another. For there to be at least one such relationship between agents, it is necessary that at least two agents satisfy the same goal. It is now possible to state sufficient conditions in terms of agency relationships, that there must be at least one engagement or one cooperation.

Formally, the the set of agency relationships that arise in multi-agent systems is defined in the AgencySociety schema. The way this schema has been defined using the schemas defining cooperations, engagements and engagement chains is shown in Figure 4.5. These schemas, in turn, require the set of entities, objects, agents and autonomous agents defined in the MultiAgentSchema schema, as well as schemas defining the individual relationships. The predicate in the schema states that there is at least one agency relationship.

\[
\begin{align*}
\text{AgencySociety} \rightarrow \\
\text{MultiAgentSystem} \\
\text{SystemEngagementChains} \\
\text{SystemCooperations} \\
\text{#cooperations} + \text{#direengagements} \geq 1
\end{align*}
\]

4.8 Agency Relations Taxonomy

By considering the entire set of engagements, engagement chains and cooperations, we can construct a map of the relationships between individual agents for a better understanding of their current social interdependence. Different situations each suggest different possibilities for interaction. For example, if Anne alone is engaging an entity, then she can interact with it without regard to others.

79
The present section provides a taxonomy of the relationships (or dependencies) between two agents that result from the agency relations that underly multi-agent systems, and are therefore derived from them. In what follows, we define eight types of agent relation: \textit{dengages}, \textit{engages}, \textit{indengages}, \textit{owns}, \textit{downs}, \textit{uowns}, \textit{sowns} and \textit{cooperates} which may hold between two agents. For each, we provide an initial description followed by a definition and formal specification. The definitions of these relations are interdependent, with some relying on others which imposes an order on them.

4.8.1 Direct Engagement Relation

The first relation we define specifies agents directly engaging others. One agent is related to another by the direct engagement relation if and only if there is a direct engagement for which the first agent is the client and the second agent is the server. For example, users directly engage their workstations and tea-shop customers directly engage their cups.

\textbf{Definition} An agent, \(c\), \textit{directly engages} another server-agent, \(s\), if, and only if, there is a direct engagement between \(c\) and \(s\).

\begin{align*}
\text{dengages} &= \{e : \text{direngagements} \cdot (e.\text{client}, e.\text{server})\}
\end{align*}

4.8.2 Generic Engagement Relation

The notion of direct engagement implies a tight coupling between the behaviours of the agents involved. Certainly, there can be no intermediate entity in a direct engagement. Suppose, however, that a user is engaging a workstation to access a printer. The user is not \textit{directly} engaging the printer but is \textit{indirectly} engaging the printer since the printer is still serving as an agent for the user. The entities an agent engages are those that, either directly or indirectly, serve some purpose for that agent. In general, any agent involved in an engagement chain engages all those agents that appear subsequently in the chain.
**Definition** An agent engages another (server) agent if there is some engagement chain that includes the server such that either the engaging agent precedes the server-agent or the engaging agent is the autonomous agent of the engagement chain.

In order to formally specify the engages relation, we must first specify the generic before relation, which holds between a pair of elements and a sequence of elements if the first element of the pair precedes the second element in the sequence.

\[
\text{before} : (X \times X) \leftrightarrow \text{seq } X
\]

\[
\forall a, b : X; \, s : \text{seq } X \cdot ((a, b), s) \in \text{before} \iff \\
(\exists t, u, v : \text{seq } X \cdot s = t \sqcap (a) \sqcap u \sqcap (b) \sqcap v)
\]

The engages relation comprises a set of pairs \((c, s)\), where there is an engagement chain, \(ec\), such that \(s\) is in \(ec.\text{agentchain}\) and either \(c\) is before \(s\) in the chain or \(c\) is the autonomous agent of \(ec\).

\[
\text{Engages} \quad \text{AgencySociety}
\]

\[
\text{engages} : \text{Agent} \leftrightarrow \text{ServerAgent}
\]

\[
\text{engages} = \{ ec : \text{engchains}; \, c : \text{Agent}; \, s : \text{ServerAgent} \mid \\
\quad s \in (\text{ran } ec.\text{agentchain}) \land \\
\quad ((c = ec.\text{autoagent}) \lor ((c, s), ec.\text{agentchain}) \in \text{before}) \cdot (c, s)\}
\]

### 4.8.3 Indirect Engagement Relation

To distinguish those engagements involving an intermediate agent we introduce the indirect engagement relation \(\text{indengages}\).

**Definition** An agent indirectly engages another if it engages it, but does not directly engage it.

The relation is formalised below in the \(\text{Indengages}\) schema, which includes the definition of generic engagements and direct engagement. Two agents are related by \(\text{indengages}\) if and only if they are related by \(\text{engages}\) but not by \(\text{dengages}\).

\[
\text{Indengages} \quad \text{Engages} \quad \text{Dengages} \\
\text{indengages} : \text{Agent} \leftrightarrow \text{ServerAgent} \\
\text{indengages} = \text{engages} \setminus \text{dengages}
\]

### 4.8.4 Generic Ownership Relation

If many agents directly engage the same entity, then no single agent has complete control over it. Any actions that an agent takes affecting the entity may destroy or hinder the engagements of the other engaging agents. This in turn, may have a deleterious effect on the engaging agents themselves. It is therefore important to understand when the behaviour of an engaged entity can be modified without any deleterious effect. This can certainly occur between an agent and an entity when there is no other agent using the entity for a different purpose. In this case we say that the agent owns the entity.

For example, suppose Anne is running several applications on a workstation, and each satisfies a different goal. If the machine is not being used by any others, then all the goals that can be ascribed to the workstation belong solely to Anne. In this way, Anne owns the workstation, and can decide
whether, for example, it is currently appropriate to re-boot the machine. In scenarios where the workstation is being used by other agents to run applications remotely, no agent owns the workstation and more care is required by all users when taking action, since this may affect other agency relationships. For example, if the workstation is being used to run remote applications and Anne re-boots the machine, it may have a deleterious effect on Anne’s relationship with the agents running remote applications.

**Definition** 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-agent that initiated the chain.

The pair \((c, s)\) is in the relation owns, if and only if, for every engagement chain, \( ec \), in which \( s \) appears, \( c \) is the autonomous agent of \( ec \) or \( c \) appears before \( s \) in the chain.

\[
\begin{align*}
\text{Owns} & : \text{AgencySociety} \\
\owns : \text{Agent} \leftrightarrow \text{ServerAgent} \\
\forall c : \text{Agent}; s : \text{ServerAgent} \bullet \\
(c, s) \in \owns & \iff (\forall ec : \text{engchains} \mid s \in \text{ran} ec.\text{agentchain} \bullet \\
& \quad \text{ec.autoagent} = c \lor ((c, s), ec.\text{agentchain}) \in \text{before})
\end{align*}
\]

4.8.5 Direct Ownership Relation

If an agent owns another agent and directly engages it then we say that the first agent directly owns the second.

**Definition** An agent, \( c \), directly owns another agent, \( s \), if it owns it, and directly engages it.

Formally, this relation is the intersection of the direct engagement relation, \( \downs \), and the generic ownership relation, \( \owns \).

\[
\begin{align*}
\text{Downs} & : \text{Dengages} \\
\owns & : \text{Agent} \leftrightarrow \text{ServerAgent} \\
\downs & : \text{Agent} \leftrightarrow \text{ServerAgent} \\
\downs = \owns \cap \dengages
\end{align*}
\]

4.8.6 Unique Ownership Relation

A further distinction of direct ownership can be made. Either no other agent directly owns the entity, or there is another agent who is also directly engaging that entity for the same purpose. The first case occurs normally but the second situation can occur if the entity is engaged by two agents each for the same purpose as generated by a single autonomous agent. This situation arises, for example, when a software agent and a program initiated by the agent are both searching a database for the same entry. To distinguish these situations we define the relation, uniquely owns, which holds when an agent directly and solely owns another.

**Definition** An agent \( c \) uniquely owns another agent \( s \), if it directly owns it, and no other agent is engaging it.

Formally, the agent pair \((c, s)\) is in this relation if \( c \) directly owns \( s \), and there is no other distinct agent, \( a \), that engages \( c \).
4.8.7 Specific Ownership Relation

There is also one further category of generic ownership since an agent may own another with respect to either multiple distinct goals or a single goal. Since multiple goals may conflict, this is an important distinction. For example, if Anne owns a workstation to find the location of two books for two different users then it is likely that achieving one goal may affect the achievement of the other. An agent specifically owns a server-agent if the server-agent has only a single goal.

**Definition** An agent, \( c \), specifically owns another agent, \( s \), if it owns it, and \( c \) has only one goal.

The formal definition states that the agent pair \((c, s)\) is an element of \( \text{sowns} \) if \( c \) owns \( s \) and the number of goals of \( S \) is equal to 1.

\[
\forall c : \text{Agent}; s : \text{ServerAgent} \cdot (c, s) \in \text{sowns} \iff (c, s) \in \text{owns} \land \#(s.\text{goals}) = 1
\]

4.8.8 Generic Cooperation Relation

The last relation we define holds between two agents when one is cooperating with another. If Anne generates the goal to move a table and persuades Bill to help then he has autonomously adopted her goal. In this situation, Bill cooperates with Anne.

**Definition** An agent, \( A_2 \), cooperates with agent, \( A_1 \), if and only if both agents are autonomous, and there is some cooperation in which \( A_1 \) is the generating agent, and \( A_2 \) is in the set of cooperating agents.

\[
\text{cooperates} = \bigcup \{ A_1, A_2 : \text{AutonomousAgent} \mid \exists c : \text{cooperations} \cdot A_1 = c.\text{generatingagent} \land A_2 \in c.\text{cooperatingagents} \land \{(A_1, A_2)\}\}
\]

Notice that the relationship is not symmetric: if agent \( aa1 \) is cooperating with \( aa2 \), this does not entail that \( aa2 \) is cooperating with \( aa1 \). Formally this can be stated as follows.

\[
\neg (\forall \text{system : Cooperates}; aa1, aa2 : \text{AutonomousAgent} \mid \{aa1, aa2\} \subseteq \text{system.autonomousagents} \land aa1 \neq aa2 \cdot (aa1, aa2) \in \text{system.cooperates} \Rightarrow (aa2, aa1) \in \text{system.cooperates})
\]

The set of all these relations are defined in the schema \textit{AgencyRelationsTaxonomy}. Figure 4.6 shows how schema inclusion is used to define the agency relations taxonomy.
Agents are limited entities and, as such, may need the help of other agents to achieve their goals. We categorise any system where agents are helping others to achieve goals as a multi-agent system.

The agency framework relies on the existence of autonomous agents that can generate their goals from their motivations, and the adoption of goals by, and in order to create, other agents so that a multi-agent system must contain at least one autonomous agent. Components in the agency framework, may be either neutral-objects, server-agents or autonomous agents, and these three specific and distinct categories give rise to three distinct cases of goal adoption. Neutral-objects can adopt goals without regard to any existing agency relationships. Since autonomous agents make their own decisions about the adoption of goals they must be persuaded to do so. Goal adoption by existing server-agents is the most complex situation, since it may affect the pursuit or maintenance of existing goals.

The generation and subsequent transfer of goals between entities through interaction produces relationships between agents, and the distinction between agency and autonomy enables us to distinguish between voluntary and compulsory relationships. The distinction arises naturally and elegantly from the agency framework. Server-agents do not have motivations and therefore rely on the existence of others to become agents and are naturally disposed to adopt the goals of others. These compulsory relations are called engagements. Autonomous agents, however, only enter into relationships if it is to their advantage, and must therefore volunteer their help. These voluntary relationships are called cooperations.

The distinction between engagement and cooperation is important since much existing work has defined cooperation only in terms of helpful agents that are predisposed to adopt the goals of others (e.g. [18, 152]). This assumes that agents are already designed with common or non-conflicting goals that facilitate the possibility of helping each other satisfy additional goals. The collection of engagements, engagement chains and cooperations are called the agency relationships, which together with the set of system entities describe the agency society. This definition can subsequently be used
as the platform from which to analyse the possible relationships or dependencies between two agents. A taxonomy of these relationships, known as the agency relations taxonomy, has been described here.

4.10 Conclusions

The inter-agent relationships identified in the chapter are not imposed on multi-agent systems, but arise naturally from agents interacting. They therefore underlie all multi-agent systems. Furthermore, the precise interdependence of agents in terms of the goals agents are achieving for others can then be analysed once the agency society is defined.

Not only are the agency relationships important in allowing users to analyse multi-agent systems, they are a crucial feature that need to be considered by computational entities designed to act within them. Any agent not recognising these relationships may (inadvertently) destroy or hinder these relationships in its interactions with agents. Engaging agents without regard to their current social context may bring about short-term gains, but may be detrimental to the agent’s achievement of its goals in the longer term. After a while, other autonomous agents may neither cooperate with this agent nor allow other non-autonomous entities to be engaged by it.

Agents not only need access to the information that identifies existing relationships so as not to destroy them, they may also need, or wish, to take advantage of the current set of agency relationships. If a software agent is engaged by an autonomous agent to find users who are engaging a particular application on a network, and another autonomous agent also wishes to establish this information, then it may either engage the software agent directly by requesting the results, or persuade the original autonomous agent to cooperate and pass on relevant information.

The analysis of the relationships between agents provides computational entities with a means of determining how they should approach interactions with those agents. For example, if I own an entity, I can do as I please, and other agents would be ill-advised to attempt to use this entity for another purpose. If I only engage it, then I may be more constrained in my interaction with it and may anticipate other agents engaging it.

Whenever an agent uses objects or resources, exploits others, communicates or interacts in any way, the relationships we have described here are present and have to be recognised. In this chapter we have shown how our notions of agency and autonomy can be used to provide an analysis of universal relationships that facilitates a better understanding of multi-agent systems, so that agent interaction can be effective and efficient.

For the remainder of this thesis we shall refer to the work in this chapter collectively as the agency relations model.
Chapter 5

An Operational Analysis of Agency Relationships

5.1 Introduction

As elaborated in Chapter 4, fundamental to the operation of multi-agent systems is the concept of cooperation and engagement between individual agents. If single-agent systems can both cooperate and engage others, they can exploit the capabilities and functionality of others to achieve their own individual goals. Once this is achieved, then such systems can potentially move beyond the advantages of robustness in traditional distributed systems in the face of individual component failure since components can be replaced and cooperation configurations realigned. In principle then, the multi-agent system paradigm allows the specific expertise and competence of different agents to complement each other so that in addition to general resilience, the overall system exhibits significantly greater functionality than individual components.

Since the set of agency relationships is critical to defining and understanding multi-agent systems, it is important to be able to model and analyse them in detail in the context of a well-founded framework. Moreover, the ways in which cooperative relationships come about are also important for a complete understanding of the nature of cooperation. This is especially true if such an analysis is to be relevant to real systems for which the invocation and destruction of such relationships is critical.

We have previously provided an analysis of key agent relationships that can be found in multi-agent systems. In this chapter, we provide an operational analysis of how these relationships are created and destroyed, and how this affects the agency society.

The four principal operations that arise from the investigation of goal adoption in Section 4.4 and which are considered in this chapter are as follows.

- A server-agent adopting the goals of an agent gives rise to a new engagement. The formal operational description is called $SystemEngage$.

- A server-agent being released from some or all of its agency obligations by an engaging agent destroys a direct engagement, called $SystemDisengage$.

- An autonomous agent adopting the goal of another to generate a new cooperation or extend an existing one. This is called $SystemCooperate$.

- An autonomous agent destroying the goals of an existing cooperation either destroys a cooperation or reduces it. This operation schema is $SystemLeaveCooperation$.

Not only is it important to understand multi-agent systems by analysing the agency relationships they contain, it is also important to understand the ways in which these relationships arise. This is particularly true if such an analysis is to be relevant to, and useful for, the construction of real systems for which the invocation and destruction of such relationships is critical. This chapter provides
an operational analysis of the invocation and destruction of engagement and cooperation. After a
description of some initial concepts in Section 5.2, the subsequent four sections describe each of the four
operations summarised above. Section 5.7 provides a worked example that shows how multi-agent
systems can be modelled in our theory. Note, that no new concepts are added in this chapter since
the focus is on providing an operational specification of agency relations developed in the previous
chapter.

5.2 Initial Concepts

To provide an operational account of these relationships, we must specify how they are affected when
new cooperations and engagements are invoked and destroyed. Before considering the operations
in detail we first specify some general functions to create schema types from individual compo-
nents. Thus, MakeEng, MakeEngChain and MakeCoop below simply construct the schema types,
Engagement, EngagementChain and Cooperation respectively. All three functions are injective
since in general the same schema element cannot be created from two different sets of component
elements. They are also surjective since every element of the schema type can be formed by choos-
ing the appropriate components. In addition, the functions, MakeEng and MakeCoop, are partial;
MakeEng is not defined if the two agents involved in the cooperation are the same, and MakeCoop is
not defined if the single autonomous agent initiating the cooperation is in the set of other cooperating
autonomous agents.

The functions make use of the unique identifier expression from the standard Z language [156].
For example, consider the mu-expression \( \mu a : A \mid p \). In this case, the function \( \mu \) assigns to the
variable \( a \) the unique value of the type \( A \) that satisfies the predicate \( p \). (For example, the expression
\( \mu n : \mathbb{N} \mid (n \times n) \mod 2 = 0 \land (n \times n) \leq 10 \) binds the variable \( n \) to the value 2.)

\[
\begin{align*}
\text{MakeEng} &: (\text{Goal} \times \text{Agent} \times \text{ServerAgent}) \rightarrow \text{DirectEngagement} \\
\text{MakeCh} &: (\text{Goal} \times \text{AutonomousAgent} \times \text{isseq}_1 \text{ServerAgent}) \rightarrow \text{EngagementChain} \\
\text{MakeCoop} &: (\text{Goal} \times \text{AutonomousAgent} \times \mathbb{P}_1 \text{AutonomousAgent}) \rightarrow \text{Cooperation}
\end{align*}
\]

\[
\forall g : \text{Goal}; a : \text{Agent}; aa : \text{AutonomousAgent}; s : \text{ServerAgent}; \\
\quad \text{ch} : \text{isseq}_1 \text{ServerAgent}; aas : \mathbb{P}_1 \text{AutonomousAgent} \bullet \\
\text{MakeEng}(g, a, s) = (\mu d : \text{DirectEngagement} | \\
\quad d,\text{goal} = g \land d,\text{client} = a \land d,\text{server} = s) \land \\
\text{MakeCh}(g, a, ch) = (\mu ec : \text{EngagementChain} | \\
\quad ec,\text{goal} = g \land ec,\text{autoagent} = a \land ec,\text{agentchain} = \text{ch}) \land \\
\text{MakeCoop}(g, aa, aas) = (\mu c : \text{Cooperation} | \\
\quad c,\text{goal} = g \land c,\text{generatingagent} = aa \land c,\text{cooperatingagents} = aas)
\]

Next, we define the generic function, \( \text{cut} \), which takes an injective sequence (where no element
appears more than once) and an element, and removes all the elements of the sequence that appear
after this element. If the element does not exist in the sequence, the sequence is left unaltered.

\[
\begin{align*}
\text{cut} &: (\text{isseq}_1 X \times X) \rightarrow \text{isseq}_1 X \\
\forall x : X; s, t : \text{seq}_1 X \bullet \\
\quad x \in \text{ran} s \Rightarrow (\text{cut}(s, x) = t \Leftrightarrow \text{last } t = x \land (\exists u : \text{seq}_1 X \bullet s = t \cup u)) \land \\
\quad x \not\in \text{ran} s \Rightarrow (\text{cut}(s, x) = s)
\end{align*}
\]

Lastly, we define two further functions, \( \text{extendchain} \) and \( \text{cutchain} \). The first function takes an
engagement chain and an object, and extends the engagement chain to include the object. The second
function cuts an engagement chain after the occurrence of an object.
Finally, we introduce some generic definitions that enable us to assert that an element is *optional*. The following definitions provide for a new type, *optional* $T$, for any existing type, $T$, along with the predicates *defined* and *undefined*, which test whether an element of *optional* $T$ is defined or not. The function, *the*, extracts the element from a defined member of *optional* $T$.

\[
\text{optional } [X] = \{ zs : \mathbb{P} X \mid \# zs \leq 1 \}
\]

\[
\begin{align*}
\text{defined } \_ & , \text{ undefined } \_ : \mathbb{P}(\text{optional } [X]) \\
\text{the:} & \text{ optional } [X] \rightarrow X
\end{align*}
\]

\[
\forall zs : \text{optional } [X] \bullet \\
\begin{align*}
\text{defined } zs & \iff \# zs = 1 \land \\
\text{undefined } zs & \iff \# zs = 0 \\
\forall zs : \text{optional } [X] \mid \text{defined } zs \bullet \\
\text{the } zs & = (\mu z : X \mid z \in zs)
\end{align*}
\]

### 5.3 Making Engagements

When a new direct engagement is formed between an agent and a server-agent, the associated engagement chain may be altered in several ways. The different possibilities depend on whether the engaging agent is at the tail, the head, or in the middle of the chain. Consider an engagement chain where $A$ is the autonomous agent at the head of the chain who is directly engaging the server-agent, $S_1$, which is directly engaging server-agent, $S_2$, which is, in turn, directly engaging $S_3$ as shown in Figure 5.1(a).

- If the autonomous agent, $A$, directly engages $O$, a new engagement chain is created solely comprising $A$ and $O$, as in Figure 5.1(b).

- If the last agent in the chain, $S_3$, engages an object, $O$, the chain is extended to include the engagement between $S_3$ and $O$, as in Figure 5.1(c).

- If any *server*-agent, other than that at the tail of the engagement chain, engages $O$, then a new engagement chain is formed comprising the original chain up to the server-agent but extended to include $O$. Thus if $S_1$ engages $O$, the existing chain is unchanged, but a new chain is formed from the engagements up to and including $S_1$ in the original chain, with the addition of the new engagement of $O$ by $S_1$, as in Figure 5.1(d).

The aspects of forming a new engagement common to all three scenarios are described in the next schema. Here, the engaging agent, *agent?*, the engaged object, *e?*, the goal of the engagement, *goal?*,...
and an optional engagement chain, \textit{chain}? , are inputs to the operation, and the set of relationships in the multi-agent system changes. The predicate part states that the object is a system object (but not an autonomous agent), and the agent is a known agent with the goal, \textit{goal}? . If no engagement chain already exists, so that \textit{chain}? is not defined, then \textit{agent}? must be autonomous as in Figure 5.1(b). If \textit{chain}? is defined, \textit{agent}? must be a server-agent, the goal of \textit{chain}? must be \textit{goal}? , and \textit{agent}? must be part of \textit{chain}? . There is no change to the set of cooperations, but the set of direct engagements is updated to include the new engagement between the agent and the object.

![Figure 5.1: Alternative New Engagements](image)

The distinct aspects of the ways in which the set of engagement chains are affected in each scenario are detailed below.

First, the engaging agent is autonomous, and a new engagement chain is formed from \textit{goal}? , \textit{agent}? , and the sequence consisting solely of the newly instantiated agent.
NewChain

GeneralEngage

undefined chain? ⇒

engchains' = engchains \cup

{ MakeCh(goal?, agent?, \{EntityAdoptGoals (e?, \{goal?\})\}) } \]

Second, the engaging agent is the server-agent at the end of the chain so that the chain is extended to include this new direct engagement.

ExtendChain

GeneralEngage

defined chain? \land (e? = last (the chain?).agentchain) ⇒

engchains' = engchains \setminus chain? \cup \{ extendchain((the chain?), e?) \}

Third, if agent? is in the chain of server-agents but not at its head then the original chain is unchanged, and a new chain is formed from the direct engagements in the original chain up to the agent, plus the new direct engagement between agent? and the newly instantiated object.

AddChain

GeneralEngage

defined chain? \land e? ≠ last (the chain?).agentchain ∧

engchains' = engchains \cup

{ extendchain((cutchain((the chain?), agent?), e?) ) }

The operation of making an engagement can then be defined using schema disjunction. Thus, when the Engage operation is applied, either NewChain, ExtendChain or AddChain occur.

Engage == ExtendChain \lor NewChain \lor AddChain

Considering the analysis of goal adoption of the previous chapter, either a new direct engagement creates a server-agent from a neutral-object, or a new server-agent from an existing server-agent. We can directly reuse the schemas defining these cases of goal adoption in order to specify the overall change to the agency society. The two cases are then specified in the following schemas.

SystemEngageObject

Engage

NeutralObjectAdoptGoals

gs? = \{goal?\}

SystemEngageAgent

Engage

ServerAgentAdoptGoals

gs? = \{goal?\}

The operation specifying the change to both the agency components and relationships and is defined by the disjunction of these two schemas.
The structure of the specification used to describe this operation is illustrated in Figure 5.2, which shows that the operation alters the agency society. The GeneralEngage schema defines those aspects of a new engagement that are common to all three scenarios above. The three schemas, ExtendChain, NewChain and CutAndAddChain, are then all refinements of GeneralEngage, each detailing the particular aspects of the three different scenarios, and the Engage schema is defined by their logical disjunction. To specify the change of state of multi-agent system entities, as well as the change to the agency relationships, we must include the schemas that define individual goal adoption by entities from Section 4.4. There are two possibilities as follows.

1. An existing server-agent is instantiated as a new server-agent.
2. A neutral-object is instantiated as a server-agent.

The change to the entire system described by the first case is defined by SystemEngageAgent and includes the Engage and ServerAgentAdoptGoals schemas. The SystemEngageObject schema defines the second case and includes Engage and NeutralObjectAdoptGoals. The operation is finally specified by SystemEngage, which is defined as the logical disjunction of SystemEngageAgent and SystemEngageObject.

5.4 Breaking Engagements

If an autonomous agent or a server-agent in an engagement chain disengages a server-agent it is directly engaging, either through destroying the goal itself or because the agent is no longer required to achieve it, all subsequent engagements in the chain are destroyed. This is because the subsequent
agents no longer satisfy a goal that can be attributed to an autonomous agent. Figure 5.3 shows how broken direct engagements propagate down through an engagement chain in this way. Initially, the engagement chain consists of the autonomous agent, A, and a chain of agents S1, S2 and S3. If A disengages S1, then necessarily S1 disengages S2 and finally S2 disengages S3.

The situation above is addressed by the Disengage schema, which formally defines the breaking of a direct engagement between engaging and engaged agents, engaging? and engaged?, respectively, and specifies the propagation of broken direct engagements for the goal, goal?, down through the associated engagement chain, chain?. All of these components are inputs. The predicates ensure that there is a direct engagement between engaging? and engaged? with respect to goal?, that chain? is an engagement chain, and that the goal of the chain is equal to the input goal?. The set of cooperations remains unchanged, but the engagement chain, chain?, is removed from the system and replaced with the chain resulting from cutting the original chain at the engaging agent. Finally, the direct engagements are updated accordingly.

The act of disengaging an entity either creates a neutral-object from a server-agent, or it creates a new server-agent from an existing server-agent. Analogously to the specification of SystemEngage

---

Disengage

engaging? : Agent
engaged? : ServerAgent
goal? : Goal
chain? : EngagementChain
\(\triangle\) AgencySociety

\[
\text{MakeEng(goal?, engaging?, engaged?)} \in \text{direngagements}
\]
\[
\text{chain?} \in \text{engchains}
\text{chain?.goal} = \text{goal}?
\text{cooperations'} = \text{cooperations}
\text{engchains'} = \text{engchains} \setminus \text{\{chain?\} \cup \{cutchain(chain?, engaging?)\}}
\text{direngagements'} = \text{direngagements} \setminus
\{d : \text{DirectEngagement} | \text{\{(d.client, d.server) in chain?.agentchain\} \land d.goal = \text{chain?.goal} \bullet d}\} \cup
\{d : \text{DirectEngagement} | \text{\{(d.client, d.server) in cutchain(chain?, engaging?),agentchain \land d.goal = \text{chain?.goal} \bullet d}\}
\]

92
System Disengage

SARG = ServerAgentReleaseGoals
SDTA = SystemDisengageToAgent
RTNO = RevertToNeutralObject
SDTO = SystemDisengageToObject

Disengage SARG RTNO
SDTA SDTO

Figure 5.4: Schema Structure for Specifying Breaking Engagements

we directly reuse the RevertToNeutralObject and ServerAgentReleaseGoals schemas, defined in Section 4.4, to formalise these two scenarios.

\[
\text{SystemDisengageToObject} \\
\text{Disengage} \\
\text{RevertToNeutralObject} \\
\{\text{goal}\} = \text{engaged}\?\cdot\text{goals}
\]

\[
\text{SystemDisengageToAgent} \\
\text{Disengage} \\
\text{ServerAgentReleaseGoals} \\
\{\text{goal}\} \subseteq \text{engaged}\?\cdot\text{goals}
\]

The SystemDisengage schema, which defines the entire system change, is the logical disjunction of SystemDisengageToObject and SystemDisengageToAgent.

\[
\text{SystemDisengage} = \text{SystemDisengageToObject} \lor \text{SystemDisengageToAgent}
\]

The structure of the specification used to describe this operation is illustrated in Figure 5.4. This shows that the schemas Disengage and RevertToNeutralObject are used to define the schema, SystemDisengageToObject, and the schemas Disengage and ServerAgentReleaseGoals to define SystemDisengageToAgent. The logical disjunction of these schemas is called SystemDisengage and defines the overall state change when an agent disengages another.

5.5 Joining Cooperations

A cooperation occurs when an autonomous agent generates a goal by recognising that goal in another autonomous agent. There are two scenarios when one autonomous agent, \( B \), adopts the goal, \( g \), of
another, \( A \). If no cooperation exists between any other autonomous agents and \( A \) with respect to \( g \), then a new cooperation structure is created. Alternatively, if a cooperation already exists between another agent and \( A \) with respect to \( g \), then \( B \) joins this cooperation.

Formally, the \textit{GeneralCooperate} schema describes the general system change when a cooperating agent, \textit{coopagent}?., adopts the goal, \textit{goal}?., of the generating agent, \textit{genagent}?.. The predicate part of the schema states that \textit{genagent}? has \textit{goal}?., that \textit{coopagent}? does not, and that both agents are autonomous. The sets of direct engagements and engagement chains remain unchanged.

\begin{align*}
\text{GeneralCooperate} & \\
goal?: \text{Goal} & \\
\textit{genagent}?., \textit{coopagent}? : \text{AutonomousAgent} & \\
\bigtriangleup \text{AgencySociety} & \\
\text{goal}? \in \textit{genagent}?..\text{goals} & \\
\text{goal}? \notin \textit{coopagent}?..\text{goals} & \\
\{\textit{genagent}?., \textit{coopagent}?!\} \subseteq \text{autonomousagents} & \\
\text{direngagements}?! = \text{direngagements} & \\
\text{engchains}?! = \text{engchains} &
\end{align*}

The two scenarios are illustrated in Figure 5.5. Either \( B \) adopts the goal of \( A \) and a new cooperation is created, or \( B \) adopts the goal of \( A \) that others (in this case \( B1 \) and \( C1 \)) have also adopted, in which case \( B \) joins an existing cooperation. Formally, the two scenarios are defined by the schemas, \textit{NewCooperation} and \textit{JoinCooperation}, where \textit{coopagent}? adopts the goal, \textit{goal}?., of the autonomous agent \textit{genagent}?.. In the first case, a new cooperation is formed since there is no existing cooperation with the agent, \textit{genagent}?., as the generating agent and \textit{goal}? as the goal. In this case a new cooperation is formed between \textit{coopagent}?., which now has the goal, \textit{goal}?., and \textit{genagent}?..

\begin{align*}
\text{NewCooperation} & \\
\text{GeneralCooperate} & \\
\neg (\exists c : \text{cooperations} \bullet c.\text{goal} = \textit{goal}? \land c.\text{generatingagent} = \textit{genagent}?) & \\
\land \text{cooperations}?! = \text{cooperations} \cup \{\text{MakeCoop}(\textit{goal}?, \textit{genagent}?., \{\textit{coopagent}?\})\} &
\end{align*}

However, if such a cooperation does exist, then \textit{coopagent}? adopts \textit{goal}?., and joins.
The change to the agency society that occurs when autonomous agents adopt the goals of others is described below and includes the `Cooperate` and `AutonomousAgentAdoptGoals` schemas. The goals of the goal adoption, `gs?`, must be equal to the set containing the one goal of the new cooperation, `goal?`.

The `Cooperate` schema is defined using schema disjunction.

\[
\text{Cooperate} \equiv \text{NewCooperation} \lor \text{JoinCooperation}
\]

The structure of the specification used to describe the operation is in Figure 5.6, which should by now be self-evident to the reader because of previous similar diagrams.

### 5.6 Leaving Cooperations

There are three cases for autonomous agents destroying the goal of a cooperation in which they are involved, illustrated in Figure 5.7. First, the generating agent, `A`, destroys the goal of a cooperation.
with the result that the cooperation is itself destroyed. This does not imply that $B2$ and $B1$ have destroyed the goal. Second, the cooperation is also destroyed when the only cooperating agent destroys the cooperation goal. Finally, when there are many cooperating agents, one of which destroys the cooperation goal, the cooperation is not destroyed but modified so that only one agent leaves the cooperation.

In all three cases the set of cooperations changes but the set of engagements is unaltered which is formalised in the `CommonLeaveCooperation` schema. A goal, `goal?`, a cooperation, `coop?`, and, optionally, two autonomous agents, `genagent?`, and `coopagent?` are inputs. The preconditions state that either `genagent?` or `coopagent?` is input, but not both. In addition, the schema checks that `genagent?` is the generating agent and that `coopagent?` is a cooperating agent of `coop?`. The sets of direct engagements and engagement chains are unchanged.

```plaintext
CommonLeaveCooperation

goal? : Goal
coop? : Cooperation
genagent?, coopagent? : optional [AutonomousAgent]
\triangle AgencySociety

#(genagent? \cup coopagent?) = 1

\begin{align*}
genagent? & \subseteq \{coop?.generatingagent\} 
coopagent? & \subseteq coop?.cooperatingagents 
direngagements' = direngagements 
engchains' = engchains
\end{align*}
```

Each of the three different scenarios can now be specified formally as refinements to this general operation schema. First, the generating agent of the cooperation destroys the goal of the cooperation. The cooperation, `coop?`, is destroyed and removed from `cooperations`.

```plaintext
GeneratingAgentDestroysCooperation

CommonLeaveCooperation

\text{defined } genagent? \Rightarrow cooperations' = cooperations \setminus \{coop?\}
```

Second, the only cooperating agent destroys the goal of the cooperation. In this case, the cooperation is similarly destroyed and removed from `cooperations`.

```plaintext
CooperatingAgentDestroysCooperation

CommonLeaveCooperation

\text{defined } coopagent? \land coopagent? = coop?.cooperatingagents \Rightarrow 
\text{cooperations'} = \text{cooperations} \setminus \{coop?\}
```
Finally, a cooperating agent that is not the only cooperating agent destroys the goal of the cooperation. It is removed from the cooperation and the resulting cooperation is added to `cooperations`.

\[
\text{CooperatingAgentLeavesCooperation}
\]

\[
\text{CommonLeaveCooperation}
\]

\[
\text{defined } \text{coopagent}? \land \text{coopagent}? \subseteq \text{coop}?.\text{cooperatingagents} \Rightarrow \\
\text{cooperations}' = \text{cooperations} \setminus \{\text{coop}\?\} \cup \\
\{\text{MakeCoop}(\text{goal}?\,\text{coop}?.\text{generatingagent}, \\
(\text{coop}\?.\text{cooperatingagents} \setminus \text{coopagent}?))\}
\]

Schema disjunction is then used to define `LeaveCooperation`.

\[
\text{LeaveCooperation} = \text{GeneratingAgentDestroysCooperation} \lor \\
\text{CooperatingAgentDestroysCooperation} \lor \text{CooperatingAgentLeavesCooperation}
\]

The overall change to the system state is then specified by `SystemLeaveCooperation`. The specification structure used to define this operation is illustrated in Figure 5.8.
5.7 An Illustrative Example

This chapter has provided operational details of how the adoption and destruction of goals, engagements and cooperations affects the configuration of agency relationships as well as the entities themselves. In order to explicate our model of agency relationships and how they are altered through the interaction of agents, we provide a worked example.

Suppose a library member, Charlie (C), cannot find a copy of a particular book in his office and visits the library to try to locate a copy. The library contains books (b1, b2, ...) and workstations (W1, W2, ...) that can be used to find information about the books. If one of the librarians, Anne (A), is currently present in the library then the system can be represented in our model as follows.

\[ \text{autonomous agents} = \{ C, A, \ldots \} \]
\[ \text{neutral objects} = \{ W1, W2, \ldots, b1, b2, \ldots \} \]

Now, suppose that Charlie has generated the goal to locate b1 and decides to try to gain Anne’s assistance. Since Anne is autonomous, Charlie cannot engage her, but instead must persuade Anne to cooperate with him to achieve his goal. Let us suppose that Anne recognises Charlie’s goal and generates it for herself. The operation defined by SystemCooperate then changes the current state of the library system, generating a new cooperation, coop, as follows.

\[ \text{coop.goal} = \text{find.b1} \]
\[ \text{coop.generating agent} = C \]
\[ \text{coop.cooperating agents} = \{ A \} \]

Notice that since there are now two autonomous agents with a social relationship between them, according to our definition of Section 4.2 the system is now a multi-agent system.

In attempting to locate b1 Anne uses the workstation, W1, to invoke a computer program, P1, which, in turn, accesses the library database D, and performs the relevant query. The goal of locating the book can then be ascribed to Anne, the workstation, the program and the database. Anne directly engages the workstation, the workstation directly engaging the program, and the program directly engaging the database. In our model, three direct engagements, deng1, deng2, deng3 and one engagement chain, engch1, are created, through repeated application of the SystemEngage operation with the following values.

\[ \text{deng1.goal} = \text{find.b1} \]
\[ \text{deng2.goal} = \text{find.b1} \]
\[ \text{deng3.goal} = \text{find.b1} \]
\[ \text{deng1.server} = A \]
\[ \text{deng2.server} = W1 \]
\[ \text{deng3.server} = P1 \]
\[ \text{deng1.client} = W1 \]
\[ \text{deng2.client} = P1 \]
\[ \text{deng3.client} = D \]
This engagement chain is constructed through the engagement of agents by other agents in order to satisfy the goal of locating the book, and is shown in Figure 5.9. Note that in order to avoid complicating the diagrams in this section, not all of the relations between the entities are shown, though they can be inferred. For example, in Figure 5.9 Anne engages W1 and P1 and D, W1 engages P1 and D, Anne indirectly engages P1 and so on.

Since the workstation can only ever be used by a single-agent (for the moment the assumption is that no remote logins are possible), Anne currently owns the workstation. In addition, Anne is directly engaging the workstation since there are no intermediate agents between Anne and the workstation, and therefore directly owns the workstation. Moreover, since no other agent is engaging the workstation with the same goal Anne uniquely owns the workstation. Lastly, if the workstation is only being used to find the location of the one book, b1, then Anne specifically owns the workstation since she has only one goal.

If only the workstation, W1, is able to run the program then a similar argument can be made about the relationship between W1 and P1, so that the workstation uniquely and specifically owns the program. In addition, Anne specifically owns P1 since it is not necessary that she directly owns it for this relationship to exist. If the database is currently being searched by other programs, the program does not own the database, since the database is being used by other agents for different purposes.

The agency relationships in the library multi-agent system therefore include the following.

\[
\text{engch1.goal} = \text{find}_\text{b1} \\
\text{engch1.autoagent} = A \\
\text{engch1.agentchain} = \langle W1, P1, D \rangle
\]

Suppose then that Charlie leaves Anne searching for the book, but inadvertently discovers b1 for himself. His goal is now satisfied and he destroys it since there is no motivational advantage in
pursuing a satisfied goal. However, there still exists an autonomous agent who has generated this goal (albeit through recognising it first in another) and as such, the agency relationships are illustrated in Figure 5.10.

Charlie might inform Anne of the discovery of the book, since if she invests unnecessary effort in trying to achieve a goal that is already satisfied, it may affect any future cooperation between Anne and Charlie. In general, if the generating agent of a cooperation destroys the goal of that cooperation, whether it becomes satisfied or not, and does not inform other cooperating agents, the possibility of future cooperation may be deleteriously affected. If Charlie does inform Anne then she will, in general, destroy the goal of locating the book, kill the program, and log off from the workstation. In this case, the workstation and program, assuming that they do not figure in some other goal engagement graph, revert to natural objects and the engagement chain is destroyed. The database, if it is still being searched by other programs, remains an agent but with reduced agency obligations.

If, on the other hand, Anne, after destroying the goal to find the book, inadvertently leaves herself logged on to the workstation with the program still searching through the database, it might appear that the goal of locating the book can still be attributed to the workstation, program and database. However, in our model, the goal of finding the book no longer exists since no autonomous agent has this goal. Certainly, the behaviour of the workstation and program might best be understood as server-agents rather than as neutral-objects, and many agents in the library might mistakenly infer that the workstation and program are being engaged.

Suppose again that Charlie does not find the book so that the situation is still as described in Figure 5.9. Also, suppose that Charlie’s friend, Emily (E), generates the goal to locate another book, b2. Since the workstation, W1, is owned by Anne, and cannot be used by multiple agents, Emily cannot share it. In this situation Emily may give the current goal of Charlie priority over her own, and either wait or find an alternative workstation to run another program. Note that this last possibility is not constrained by the database being engaged by Anne, since it is not owned by Anne.

Another option is for Emily to take the workstation forcibly. However, this may drastically affect future cooperations with Anne, Charlie and other autonomous agents in general. Emily could also attempt to persuade Anne to release the workstation. If Anne cannot be persuaded, she may ask another librarian, Bill (B), to take part in separate cooperation negotiations to persuade Anne to release the workstation. Emily may choose this option if she considers that Bill has a better chance than she does of persuading Anne to adopt her goal.

One other option available to Emily is to persuade Anne to cooperate with her so that Anne now has two goals of finding two different books. In this, there are six direct engagements, deng1, deng2, deng3, deng4, deng5 and deng6, two engagement chains, engch1 and engch2, and, since Anne is cooperating with both Charlie and Emily, two cooperations coop1 and coop2, as follows.
uses a different workstation, W2, to access a different program, P2, to locate b2.

The state of this model in terms of the *AgencySociety* schema defined in Section 4.7 is then as follows.

\[
\begin{align*}
\text{autonomousagents} & = \{A, C, E\} \\
\text{serveragents} & = \{W1, P1, D\} \\
\text{neutralobjects} & = \{b1, b2, \ldots\} \\
\text{direngagements} & = \{deng1, deng2, deng3, deng4, deng5, deng6\} \\
\text{engagementchains} & = \{engch1, engch2\} \\
\text{cooperations} & = \{coop1, coop2\}
\end{align*}
\]

From the information contained in this state, it is now possible to derive the precise relation between any two agents as defined by the agency relations taxonomy in Section 4.8. For example, consider the relationship between Anne and the workstation. Since the workstation is not being used by another agent for a different purpose Anne still owns the workstation and the program. Since there are no intermediary agents between Anne and the workstation Anne directly owns the workstation and as she is the only agent engaging the workstation she also uniquely owns it. However, Anne is using the workstation for two separate goals and as such, does not specifically own the workstation. This may be an important consideration for Anne since it may be that the that the achievement of one goal may adversely affect the achievement of the other. Specifically, the following agency relationships can be derived.

\[
\begin{align*}
\{(A, C), (A, E)\} & \subseteq cooperates \\
\{(A, W1), (W1, P1), (P1, D)\} & \subseteq dengages \\
\{(A, P1), (A, D), (W1, D)\} & \subseteq indengages \\
\{(A, W1), (W1, P1), (P1, D), (A, P1), (A, D), (W1, D)\} & \subseteq engages \\
\{(A, W1), (W1, P1), (A, P1)\} & \subseteq owns \\
\{(A, W1), (W1, P1)\} & \subseteq down\s \\
\{\} & \subseteq sou\s \\
\{(A, W1), (W1, P1)\} & \subseteq wouns
\end{align*}
\]

Finally, Figure 5.12 shows another situation that arises when Bill, after being asked by Emily, uses a different workstation, W2, to access a different program, P2, to locate b2.
5.8 Summary

Identifying the relationships between agents in multi-agent systems provides a way of understanding the nature of the system, its purpose and functionality. However, if we are to build systems that need to recognise and exploit these relationships, so that prior relationships are not destroyed when new ones are created, we must extend such analyses into operational areas. This chapter has provided just such an analysis, showing how different configurations of agents and relationships can evolve by invoking and destroying them. In this analysis we have considered a range of scenarios and the effects of changing relationships upon them.

Operational analysis is particularly important for system developers aiming to build programs that are capable of exploiting a dynamic multi-agent environment. It provides a vital link from the structural account to methods for accessing and manipulating these structures. If such analyses are avoided, then the merit of research that aims to lay a foundation for the practical aspects of multi-agent systems is limited.

Now that we have analysed fundamental agent relationships and provided an operational account of their creation and destruction in multi-agent systems, we consider the design of agents required to function in such systems. In the next chapter, therefore, we analyse the requisite dimensions of agents that have the ability to act effectively in multi-agent systems.
Chapter 6

Sociological Agency

6.1 Introduction

In multi-agent systems, individual autonomous agents seek their “own benefit in interacting with others” [84]. Now that we have identified and analysed the inter-agent relationships that arise from the application of the agency framework, we can address the requisite deliberative qualities of such agents for effective and efficient action in order to gain this benefit. An individual agent can achieve this, first through the appropriate use of its own capabilities, and second through the successful exploitation of the capabilities of others.

In both these cases, however, an agent must be able to recognise the relationships between existing entities in order that it can successfully perform the relevant task within the social constraints of the current situation. Once an agent is able to recognise the existing agency relationships not only can it exploit them but it can also reason about altering them without inadvertently and deleteriously affecting its relationships with others.

This chapter is concerned with the dimensions required by agents to achieve such behaviour and we identify several key areas as follows. An agent may:

- have an internal store which can be used to access percepts;
- model the entities in its environment (we define models in Section 6.3);
- model the agency relationships in its environment;
- represent plans and be able to reason about them to take advantage of the capabilities of other entities; and
- be able to quantify how the current agency relationships affect the selection of plans for execution.

In this chapter we develop our existing models of agents to provide models of deliberative agents with prescribed dimensions of modelling others and representing and reasoning about plans. These are presented as refinements at the agency framework level of agents as opposed to autonomous agents for two reasons. First, an agent does not have to be autonomous in order to have the dimensions discussed in this chapter. Non-autonomous agents may be able to model others and reason about plans, and descriptions at the agent level are the most general. Second, when we move to issues of autonomy, the role of motivation in directing behaviour can be explicitly evaluated. This enables a clear analysis of the increased functionality that arises through the motivations of autonomous agents as opposed to their non-autonomous counterparts.

Agents able to represent plans, or model others, need an internal store to record the model or plan. Section 6.2 therefore considers internal store or memory agents that have this capacity. Agents who can model their environment, the entities within it, and the relationships between them are then developed in Section 6.3. We show how these models can be applied by autonomous agents when
making decisions that affect the current social state. Section 6.4 provides an account of the requisite data structures required for effective social planning. This section first provides a high-level account of the basic functionality required for representing and reasoning about plans without detailing the nature of the agent’s environment within which it is situated. This high-level description is then refined to a social environment, which we take to be one containing other agents. In this way initial high-level definitions of agents are provided that can apply equally well to single-agent systems. Further, we are able to isolate those aspects of an agent’s design that enable them to act effectively in multi-agent systems.

6.2 Agent Store

In Chapter 3, a definition of agency was presented that relies solely on the environment, and the agent’s perceptions, goals and motivations (if available) to determine action. However, if agents need to model their environment, or evaluate competing plans, they need more than just actions, goals and motivations, they will require an *internal store*. In general, agents without internal stores are extremely limited since their experience cannot direct behaviour, and actions can only be selected reflexively [8, 106]. In addition, as well as being able to capture historical information agents with a store can cache local information in order to avoid repeating complex computational tasks.

Modelling agents at a lower level enables us to distinguish the *external* environment of an agent from some other, *internal*, repository that can be used to access percepts. The distinction between perceiving a tree in a physical environment and *recalling* a tree that was perceived previously, for example, is that the latter does not feature as a component of the possible percepts available in the current external environment and must be derived from an internal store. This store exists as part of an agent’s state in an environment but it must also have existed *prior* to that current state. We call this feature an *internal store or memory*, and define *store agents* as those with memories. Store-agents therefore have the facility to access a shared, or individual, internal store, which can generate percepts.

In this section, we outline these agents, providing the full specification, but omitting excessive technical explanation. Formally, the definition of a store-agent is a refinement of the agent schema defined in Section 3.6, and includes the variable *store*, represented as a non-empty set of attributes. Notice that the type of a store is *Environment* which, when accessed, produces perceptions of type *View*.

```
StoreAgent
Agent
store : Environment
```

Now, since there is both an external environment and a memory, it is necessary to distinguish between internal and external perceiving actions, where internal actions can access the store, and external perceiving actions the external environment. The internal perceiving actions must be non-empty since otherwise the store cannot be accessed.

In defining perception, *StoreAgentPerception* includes *AgentPerception* and *StoreAgent*, with *internalperceivingactions* and *externalperceivingactions* referring to the external and internal perceiving actions, respectively. The *storecanperceive* function determines the set of perceptions that can be currently generated from its internal store, while *extcanperceive* determines those percepts possible from the external environment. The internal perceiving actions must be non-empty, and the internal and external perceiving actions are disjoint and together comprise the set of all perceiving actions. Note that those percepts that the agent actually selects make up a subset of these available attributes and are dependent on the goals of the agent as defined previously. Thus the *willperceive* function from the *AgentPerception* schema is still applicable, since the store is carried through possible percepts and actual percepts to the action-selection function, *agentactions*.
The state of such an agent is specified by \textit{StoreAgentState}. Once a store-agent is placed in an external environment, the values of the potential and actual sets of percepts, and the next set of actions can be determined. The latter two of these are specified in the base \textit{AgentState}. However, the possible internal percepts, \textit{possinternalpercepts}, are derived from applying its internal perceiving actions to its internal store, and the possible external percepts, \textit{possexternalpercepts}, are derived by applying its external perceiving actions to the current external environment. The possible percepts then available to a store-agent are equal to the union of these two sets.

\textit{StoreAgentState}:
\begin{align*}
\text{StoreAgentPerception} \\
\text{StoreAgentAction} \\
\text{AgentState} \\
p\text{ssinternalpercepts}, p\text{ssexternalpercepts} : \text{View} \\
\text{extenv} : \text{Environment} \\
p\text{ssinternalpercepts} &= \text{storecanperceive store internalperceivingactions} \\
p\text{ssexternalpercepts} &= \text{extcanperceive environment externalperceivingactions} \\
p\text{sspercepts} &= p\text{ssinternalpercepts} \cup p\text{ssexternalpercepts} \\
\text{extenv} \cup \text{store} &= \text{environment}
\end{align*}

The consequences of an action on the external environment are represented by the same type as the base function \textit{effectinteraction}. However, the way in which the internal store is updated is different and depends on the design of the agent as shown in \textit{UpdateStore}. Such an update depends on the current percepts, internal store, current goals, and the actions the agent has most recently performed. Goals are relevant here because they may constrain what is recorded in the internal store, and what is not.

\begin{align*}
\text{externaleffectinteraction} : \text{Environment} \to \mathbb{P} \text{Action} \to \text{Environment}
\end{align*}
Now, when an agent interacts with its environment, both the external environment changes, and the store changes, specified in StoreAgentInteracts, which is a refinement of AgentInteracts. When a store-agent acts, it does not necessarily record just those attributes that are currently available in the external environment, but may also store some other attributes regarding more general learned or acquired information. Certain agent designs may not allow for their store to be updated, in which case there is no learning facility, as specified by FixedStoreAgentInteracts.

The model of store-agents described in this section allows us to describe agents at any level of abstraction depending on how the store is defined. We can model the store as a series of binary digits but more commonly we model this store as containing information at the knowledge-level [121]. Such agents are referred to by Genesereth and Nilsson as knowledge-level agents [65], where an agent’s mental actions are viewed as inferences on its database, so that prior experience and knowledge can be taken into account when evaluating which action to take. These agents can be reformulated by application of the store-agent model. However, before we proceed, we first consider hysteretic agents, which are strongly related to knowledge-level agents only more general than them.

### 6.2.1 Hysteretic Agents

Hysteretic agents are similar to store-agents since they both are able to retain information internally. These agents have an internal store that can be in one of several states from the set \( I \) of all possible internal states. As with tropistic agents (see Section 3.8), external perceptual capabilities are limited and partition the set \( S \) of external environments as described by \( see \). In addition — but in contrast to modelling external perceptual abilities as limited — it is assumed that hysteretic agents are able to distinguish each possible internal state.

Hysteretic action-selection is now dependent on the agent’s internal state as well as the perceptions of the external environment. Updating the internal state is defined by \( internal \), which is a function of elements of \( I \) and \( T \). The following tuple determines Hysteretic agents.

\[
(I, S, T, A, see : S \rightarrow T, \text{do} : A \times S \rightarrow S, \text{action} : I \times T \rightarrow A, \text{internal} : I \times T \rightarrow I)
\]

Recall that the functions defined in this section for store-agents are as follows.

\[
\begin{align*}
\text{storecanperceive} &: \quad \text{Environment} \rightarrow \mathbb{P} \text{Action} \rightarrow \text{View} \\
\text{extcanperceive} &: \quad \text{Environment} \rightarrow \mathbb{P} \text{Action} \rightarrow \text{View} \\
\text{updatestore} &: \quad \text{View} \rightarrow \text{Environment} \rightarrow \mathbb{P} \text{Goal} \rightarrow \mathbb{P} \text{Action} \rightarrow \text{Environment}
\end{align*}
\]
The set of hysteretic agent internal states is defined to be the type \( \text{Environment} \) from the agency framework and the type definitions for tropistic agents still apply.

\[
I == \text{Environment} \land S == \text{Environment} \land T == \text{View} \land A == \text{Action}
\]

Hysteretic agents perceive each internal state completely and correctly at every step in the agent’s cycle, which is formalised by equating \( \text{storec anp er c eive} \) with the identity function. The possible external perceptions are defined in the same way as those for tropistic agents and the \( \text{willperceive} \) function is again equated to the identity function, since goals do not constrain either the hysteretic agent’s internal or external perceptions.

\[
\forall i : I \bullet (\text{storec anp er c eive } i \text{ internalc per c eivingacti ons}) = i \\
\forall e : S \bullet \text{see } e = (\text{extc anp er c eive } e \text{ perceivingacti ons}) \\
\text{willperceive } g = \{ e : \text{Environment} \bullet (e, e) \}
\]

As described earlier in this section, the set of percepts perceived by a store-agent with goals \( gs \), store, \( i \), and environment, \( e \) is calculated as follows.

\[
\text{willperceive } gs ((\text{extc anp er c eive } e \text{ perceivingacti ons}) \cup (\text{storec anp er c eive } i \text{ internalc per c eivingacti ons}))
\]

Using the definitions above, this simplifies to the following predicate, which states that for an internal store, \( i \), and an external environment, \( e \), the agent perceives \( i \) and a subset of the external environment, \( e \), as determined by \( \text{extc anp er c eive} \).

\[
\forall i : I, e : S \bullet \text{actualc per c epts } = (\text{extc anp er c eive } gs \ s) \cup i
\]

Hysteretic action selection and internal store update are both reformulated simply below. Since the internal update function for hysteretic agents is not dependent on the actions-performed, this parameter is fixed in \( \text{update } \text{store} \) to the empty set.

\[
\forall i : I, t : T \bullet \text{action } (i, t) = \text{agent } \text{actions } \{ gs \} (i \cup t) \} \\
\forall v : \text{View}; i : \text{Environment} \bullet \text{internal } (v, i) = \text{update } \text{store } v \ i \ \{ gs \} \}
\]

### 6.2.2 Knowledge-Based Agents

The conceptualisation of store or hysteretic agents allows the description of agents at any level of detail. The most common of these levels is the knowledge-level [121] where, in many cases, the store consists entirely of formulae in first order predicate calculus. The tuple defining these agents is exactly the same as for hysteretic agents and is not presented here. Instead we consider the representation of the internal store since many agents contain databases or knowledge bases that consist of predicate calculus formulae. This also demonstrates how our abstract agency framework types can be refined to describe more practical and definite concepts.

A formula is often called an \( \text{atom} \) [132] and can be defined as a \( \text{predicate symbol} \) (denoted by the given set, \( \text{PredSym} \)), and an associated sequence of \( \text{terms} \) as its argument.

\[
\text{Atom} \\
\text{head} : \text{PredSym} \\
\text{terms} : \text{seq } \text{Term}
\]

A \( \text{term} \) is defined as either a \( \text{constant} \), a \( \text{variable} \), or a \( \text{functor symbol} \) with a sequence of terms as a parameter. If the set of all constants, variables and functor symbols are all given sets, the definition of a term is as follows.
Agents whose internal store consists solely of predicates have been referred to as Knowledge-Based Agents [65]. Sometimes the predicates continued in the store of these agents contain ground atoms only; they do not contain variables. Ground atoms can be formalised using the auxiliary functions below that return the set of variables for terms and atoms respectively.

\[
\text{termvars} : \text{Term} \rightarrow (\mathbb{P}\text{Var})
\]
\[
\text{atomvars} : \text{Atom} \rightarrow (\mathbb{P}\text{Var})
\]

\[
\forall \text{at} : \text{Atom}; c : \text{Const}; v : \text{Var}; f : \text{FunSym}; ts : \text{seq Term} \bullet
\]
\[
\text{termvars} (\text{const} c) = \emptyset \wedge
\]
\[
\text{termvars} (\text{var} v) = \{v\} \wedge
\]
\[
\text{termvars} (\text{functor}(f, ts)) = \bigcup (\text{ran} (\text{map termvars ts})) \wedge
\]
\[
\text{atomvars} \text{at} = \bigcup (\text{ran} (\text{map termvars at.terms}))
\]

The generic higher-order function map used above takes another function as argument and applies it to every element in a sequence.

\[
\text{map} : (X \rightarrow Y) \rightarrow (\text{seq } X) \rightarrow (\text{seq } Y)
\]

\[
\forall f : X \rightarrow Y; x : X; xs, ys : \text{seq } X \bullet
\]
\[
\text{map } f \emptyset = \emptyset \wedge
\]
\[
\text{map } f \{x\} = \{f x\} \wedge
\]
\[
\text{map } f \{xs \cup ys\} = \text{map } f xs \cup \text{map } f ys
\]

It is then a simple matter to define the set of ground atoms.

\[
\text{BaseAtoms} == \{a : \text{Atom} | \text{atomvars } a = \emptyset\}
\]

We can now provide a semantic base for type, Attribute, defined originally as a given set, by specifying attributes as ground predicates.

\[
\text{Attribute} == \text{BaseAtoms}
\]

As when defining tropistic agents using the agency framework, we have shown how the store-agent model can be applied directly to reformulate agent architectures with an internal memory. The knowledge-level agent example also demonstrates how the specification types can be elaborated as required at an appropriate level of detail.

6.3 Agent Models

We now turn our attention to the dimensions specifically required for an agent to function effectively in multi-agent systems. In order that agents can take advantage of the capabilities of others they will generally need models of them. We take models to be representations recorded in the internal store. (For example, the Acquaintance Models of ARCHON [85] and the modelling layer of TouringMachines [52] contain models of other entities encoded in an internal store.) Here we construct agent models at this level of abstraction by application of the agency framework and argue that, in general, agents not only need models of others but also of the relationships between them in order to act effectively.
6.3.1 Entity Models

The ability to isolate components in a complex external environment and to perceive them as a whole is the first task of any situated agent that is required to operate effectively. We call representations of models of entities entity models. The way in which agents group attributes together to form entity models is purely subjective, being constrained by their capabilities and goals. Formally, we distinguish representations of the entities themselves from representations of models of those entities by defining EntityModel as the representation of a model that an agent has of an entity and other model-types analogously as follows.

\[
\begin{align*}
\text{EntityModel} &\equiv \text{Entity} \\
\text{ObjectModel} &\equiv \text{Object} \\
\text{AgentModel} &\equiv \text{Agent} \\
\text{AutonomousAgentModel} &\equiv \text{AutonomousAgent} \\
\text{NeutralObjectModel} &\equiv \text{NeutralObject} \\
\text{ServerAgentModel} &\equiv \text{ServerAgent}
\end{align*}
\]

Following the structure of the agency framework, the most basic agents in this category are those that can distinguish entities. This kind of modelling is used by mechanisms such as a robot arm on a production line. The arm is only concerned with the perceptual stimuli needed for it to perform appropriate actions on an entity, and is not concerned with its capabilities and goals. The AgentModelEntities schema, which refines the StoreAgent schema, describes such an agent and includes the modelentities variable to represent its entity models.

\[
\begin{align*}
\text{AgentModelEntities} & \subseteq \text{AgentModel} \\
\text{AgentModel} & \subseteq \text{AgentState} \\
\text{modelentities} &\equiv P_1 \text{EntityModel}
\end{align*}
\]

If an agent can associate capabilities with its models of an entity, then it can also model objects. The AgentModelObjects schema describes this increased capability, which, for example, describes a robot able to test components in a production line for specified capabilities.

\[
\begin{align*}
\text{AgentModelObjects} & \subseteq \text{AgentModelEntities} \\
\text{modelobjects} &\equiv P_1 \text{ObjectModel} \\
\text{modelobjects} &\equiv \text{modelentities}
\end{align*}
\]

Increasing the capability with which an agent can model its environment according to the agency framework we proceed to specify agents able to distinguish agents from neutral-objects in the schema, AgentModelAgents. The subsequent AgentModels schema specifies those agents who are, in addition, able to distinguish autonomous agents from server-agents. Agents that can model the autonomy of others can then, in theory, understand the origination of goals in a multi-agent system. This schema includes an optional modelself variable, which is the model an agent has of itself.

\[
\begin{align*}
\text{AgentModelAgents} & \subseteq \text{AgentModelObjects} \\
\text{modelagents} &\equiv P_1 \text{AgentModel} \\
\text{modelneutralobjects} &\equiv P \text{NeutralObjectModel} \\
\text{modelagents} &\equiv \text{modelneutralobjects} \\
\text{modelobjects} &\equiv \text{modelagents} \cup \text{modelneutralobjects}
\end{align*}
\]
6.3.2 Sociological Agents

In general, it is not enough for agents to model other entities such as objects, agents or autonomous agents in isolation; they must also model the agency relationships between them. A robot without this capability could not model the relationship between users and workstations, for example, and could not reason about negotiating with the user to release the workstation for its use. Agents must model the engagements, cooperations and engagement chains of the system in which they are situated.

Definition. A sociological agent is an agent that models other agents, and the set of agency relationships (cooperations, engagements and engagement chains) between them.

There are many definitions of what constitutes a social agent rather than a sociological agent. For example, Wooldridge states that any agent in a multi-agent system is necessarily social [170] and Moulin and Chaib-draa [115] take an agent to be social if it can model others. However, the term is more often associated with social activity such as provided by Wooldridge and Jennings [173] who refer to the process of interaction. We chose “sociological” since we are considering agents that can model their social environment rather than act in it socially. We argue that an agent must be sociological before it can be a generally effective social agent.

A sociological agent therefore views its environment as containing a collection of entities with engagements, engagement chains and cooperations between them. Such an agent is specified below in SociologicalAgent which includes DirectEngagementModel, EngagementChainModel, and CooperationModel, which spell out the mental representations an agent has of the agency relationships. For formal consistency, we place constraints on the mental representations of relationships and entities. Thus, if a relationship is modelled by an agent, the agents involved in that relationship must themselves be modelled. For example, the first predicate of the SociologicalAgent schema states that the client and server-agents for a modelled direct engagement must be contained in the set of modelled agents.
Sociological agents can automatically derive the type of the relationship between two agents from these models as defined by the agency relations taxonomy. If in a sociological agent’s view the only agent engaging the server-agent \(s\) is \(c\), then the sociological agent will model \(c\) as owning \(s\). The relations \textit{modelengages}, \textit{modedengages} and \textit{modelcooperates} are formalised in the \textit{ModelAgencyRelationships} schema, and the other relations in the taxonomy can be defined similarly.
Any recognised agency relationships can then be exploited by intelligent sociological agents for more effective operation. Each agent must maintain information about the different entities in the environment, so that both existing and potential relationships between those entities may be understood and consequently manipulated as appropriate. For example, neutral-objects are not involved in relationships with agents, so that they can be engaged without affecting existing agency relationships. If an entity is viewed correctly as a server-agent, this must imply that it is engaged by another autonomous agent, either directly or through a chain of intermediate agents, grounded with an autonomous agent at the head of the chain. Knowledge of the agency of an entity allows viewing agents to reason about its role and the agents engaging it.

### 6.3.3 Modelling the Motivations of Others

Understanding the motivations that generated the goal of an agency relationship provides further information. It is motivation that is the ‘force’ that causes engagement chains to be created, satisfying goals that appease the motivation. In attempting to understand the nature of the relationships between entities in a multi-agent system it is therefore necessary to be able to assess the relative strengths of motivation that caused the current agency relationships.

Consider, for example, the situation in which \( A \) is using a pencil and \( B \) correctly views this pencil as \( A \)'s agent satisfying her goal of writing notes. If \( B \) wishes to use this pencil, he must consider the strength of the motivations that generated \( A \)'s goal, and his model may lead him to predict whether he will succeed in securing use of the pencil. Now, if \( B \) understands that \( A \)'s goal was generated because of an imminent important deadline, then he may decide that an attempt to break the agency will not be successful. Alternatively, if the motivation for using the pencil was weak, then \( B \) may rate his chances more highly.

In order to further illustrate how autonomous agents can use their models to make informed decisions about potential courses of action, we provide a more general example where one autonomous agent wishes to use the owned entity of another autonomous agent. This example requires the previous definition of the **motivational effect** on an agent of satisfying a goal, **satisfy**, defined in the AssessGoals schema in Section 4.3. We also adopt the following conventions.

- The expression, \( \text{satisfy}^A_B(gs) \), denotes the motivational effect on the autonomous agent, \( A \), of satisfying the set of goals, \( gs \), according to \( B \)'s model.

- The expression, \( \text{model}^A_{B,goals} \), denotes \( B \)'s model of \( A \)'s goals.

- The expression, \( \text{goal} \), denotes the goal to prevent goal.

Let us assume that \( B \) models \( A \) as being autonomous, having the goal, \( g_A \), and directly owning the server-agent \( S \), for the goal \( g_A \). We can write the following.
A ∈ B.modelautonomousagents
modeludegA,gB = \{gA\}

MakeDirectEngagement(\(A, S, gA\)) ∈ B.modeldirengagements
(\(A, S\)) ∈ B.modeldowns

The current motivational affect that B models A as having is represented by the following predicate.

\(satisfyB^2(\{gA\})\)

Further, suppose that B wants to use S for some other goal \(gB\). There are several possible courses of action for B.

- B can persuade A to share S.
- B can persuade A to release S.
- B can attempt to take S by force without A’s permission.
- B can give A priority and find an alternative.

Any decision as to which alternative B takes requires an analysis of both B’s motivations and B’s models of A’s motivations.

- \(satisfyB^2(gB, gA) > satisfyB^A(\{gA\})\). If \(gB\) and \(gA\) do not conflict, it is possible for S to adopt both of the goals of B and A without violating any motivational constraints. So long as the motivational effect on A of satisfying both goals is more than satisfying just her own, A will be disposed to share S.

- \(satisfyB^2(gB) > satisfyB^A(\{gA\})\). B understands that A stands to gain more from enabling B to satisfy his goal than from A satisfying her own goal. This is due to the effect that a positive change in B’s motivations will have on A. This may require that B explains and persuades A of the degree of effect that \(gB\) will have on him and hence on her. For example, if Anne is currently reading a book that Bill wants to borrow, his goal of borrowing the book may conflict with Anne’s goal. However, if Anne wants to please Bill and does not need to read the book now, she may happily lend the book to him.

- \(satisfyB^2(gB) > satisfyB^A(\{gB\})\). It may seem obvious that the motivational effect on B of satisfying his goal should be greater than the motivational effect on B of satisfying A’s goal. However, if A’s goal is not satisfied, then the motivational effect on A will be negative, and this results in a state which must be considered in terms of its effect on B. (In other words, a negative motivational effect on A, particularly if it was a consequence of some action of B, may result in a negative motivational effect on B.) Thus this alternative may be chosen if there is a positive motivational effect from B’s goal being satisfied, and this is greater than the negative consequences of A’s goal not being satisfied. Note that this relies on the relationship of B to A. Normally, B’s motivations will be such that negative motivational effect on other agents will lead to some negative motivational effect on B himself. If, however, B is motivated by malicious concerns, then it is certainly possible that the consequences of A’s goal not being satisfied may have a positive motivational effect on B. While we do not envisage such a situation arising regularly, and though this is a case typically not considered in related work, it ought to be possible within any formalism. By using motivations in the way we describe, we allow the possibility of perverse configurations leading to such malicious behaviour, but envisage appropriate design of motivations so that this does not arise. Returning to the example where Anne is reading a book that Bill wants to borrow, he can simply take the book from her without permission as before. It only makes sense for him to do this, however, if the benefit he gains from having the book is more significant than the bad feeling caused in Anne by Bill having taken it forcibly.

113
• \( \text{satisfy}_B^{\bar{g}} \{ g_B \} < \text{satisfy}_B^{\bar{g}} \{ g_A \} \). B understands that the motivational effect of satisfying his goal will be less than the effect of causing a negative motivational effect on A through \( g_A \) not being satisfied. This affects B’s behaviour, because his motivations are configured in such a way that he is concerned for A. In summary, our model captures normal social behaviour by which we act so as to avoid annoying others, but allows for situations where we may deliberately choose to annoy them as in the previous case.

6.3.4 Modelling the Models of Others

In some situations, agents may be designed not only to model other agents but also to model the \textit{models of other agents}. This enables agents to consider the view of others on which they base their actions. As Durfee acknowledges [44], if agents are to coordinate their activities they may need to know not only about each other and about themselves but about how others view themselves and how others view others. For example, if A models B as a sociological agent then A assumes B acts based on his current model of the agency relationships. Such an agent is defined formally in the \textit{ModelSociological} schema.

\[
\begin{aligned}
\text{ModelSociological} \\
\text{SociologicalAgent} \\
\text{modelSociologicalagents : Agent} \rightarrow \text{SociologicalAgent} \\
\text{dom modelSociologicalagents} \subseteq \text{modelagents}
\end{aligned}
\]

Such agents can then exploit their environment more effectively than agents described by the \textit{SociologicalAgent} schema. As an example, suppose we have the following situation:

• agent B is specified by \textit{ModelSociological};
• agent A is specified by \textit{SociologicalAgent};
• B desires to engage the entity E;
• E cannot be shared but must be owned;
• A owns E; and
• B’s model of A’s model of their shared environment includes the fact that A does not realise the agency relationship between herself and E (A might be a very forgetful agent).

In this scenario B can forcibly use E thus destroying the agency relationship between A and E but in such a way as not to affect any relationship between himself and A. This level of reasoning is not possible for sociological agents, but only for those agents able to model \textit{others} as sociological.

Extensions to describe agents able to model the models that agents have of others can be formalised easily reusing existing schemas as shown in the \textit{ModelModels} schema. Schemas can be extended in this way to develop nested models such as those proposed in Recursive Agent Modelling [164]. This shows how the agency framework can be applied to the incremental development and subsequent analysis of the social capabilities of agents.

\[
\begin{aligned}
\text{ModelModels} \\
\text{ModelSociological} \\
\text{modelSociological : Agent} \rightarrow \text{ModelSociological} \\
\text{dom modelSociological} \subseteq \text{modelagents}
\end{aligned}
\]
6.4 Agent Plans

6.4.1 Introduction

Sometimes, agents may select an action, or a set of concurrent actions, to achieve goals directly. At other times, however, there may not be such a simple correspondence between goals and actions, and appropriately designed agents may perform sequences of actions, or plans, to achieve their goals. In multi-agent systems, agents have at their disposal not only their own capabilities but, potentially, the capabilities of others. Agents with plans requiring actions outside their competence need models of others to consider making use of their capabilities. If these agents are sociological then they can evaluate plans with respect to their model of the current agency relationships. For example, agents can decide how much plans can exploit current relationships as well as consider how they may impinge on them. In general, agents must reason about how to exploit existing and potential agency relationships without inadvertently or unnecessarily destroying them.

We do not model the process of planning, which is the ability to construct sequences of actions that are executed in an attempt to bring about a goal [98], but instead consider how plans can be modelled and then how agents can evaluate plans using these models. Many agent systems, including the Procedural Reasoning System [69], do not construct their plans in this sense, but many systems do, such as the partial global planning of agents in the distributed vehicle monitoring testbed [45]. Clearly, a representation of plans is required before planning algorithms can be specified and this is what we attempt in this section. Further work will show how different planning techniques can be incorporated into our models.

There are many different notions of agent plans both at the theoretical and practical level and in order to substantiate the claim that the agency framework and ensuing models can be generally applied it is necessary that different representations of plans can all be equally accommodated. Whilst we do not specify every type of plan of which we are aware, we do intend to show how the agency framework can be extended to describe familiar notions of plans, and to impress upon the reader how other models of plans can be similarly accommodated. We aim to achieve this by specifying general theoretical plan representations that we call total plans, partial plans and tree plans.

6.4.2 Plan-Agents

One methodological characteristic of our work is the incremental development of the models in it. Therefore, we first construct a high-level model of plan-agents, which apply equally well to reactive or deliberative, single-agent or multi-agent, planners. It represents a high-level of abstraction because nothing is decided about the nature of the agent, the plan representation, or of the agent’s environment; we simply distinguish categories of plan and possible relationships between an agent’s plans and goals. Specifically, we define non-active plans as those which have not been identified as a means of achieving a current goal; active plans as those identified as candidate plans not yet selected for execution; and executable plans as those active plans that have been selected for execution.

In general, an agent will have a repository of goals, and a repository of plans that have either been designed before the agent starts executing [68] or acquired and developed over the course of the agent’s life [136]. In addition, plans may be associated with one or more goals, identifying the plan as a potential means of achieving those goals. Sugawara [159], for example, proposes a plan-reuse framework which maintains information that includes associations between plans and goals. Agents can then reuse plans that have successfully achieved their goal, by recording a relationship between the plan and goal. Similarly, in dMARS goals are associated with plans by means of the invocation condition of plans [68]. In addition, a set of active plans may be associated with a current goal signifying that they are alternatives for execution.

Formally, we initially define the set of all agent plans to be a given set ([Plan]), so that at this stage we abstract out any information about the nature of plans themselves. Our highest-level description of a plan-agent is formalised in the PlanAgent schema below. Since plans must be encoded as aspects of an internal store, and further since active and executable plans are clearly aspects of the agent’s state, the StoreAgentState schema is included. We chose to refer to plan-agents rather than ‘planning’ agents or ‘planners’ since we remain neutral on whether the agent has the ability to construct or modify
its own plans. The variables goallibrary, planlibrary, activeplans and executableplans represent the agent’s repository of goals, repository of plans, active plans and executable plans, respectively. Each active plan is necessarily associated with one or more of the agent’s current goals as specified by activeplangoal. For example, if the function contains the pair \( (g, \{p_1, p_2, p_3\}) \), it indicates that \( p_1, p_2 \) and \( p_3 \) are competing active plans for \( g \). Whilst active plans must be associated with at least one active goal the converse is not true, since agents may have goals for which no plans have been considered. Analogously the plangoallibrary function relates the repository of goals, goallibrary, to the repository of plans, planlibrary. However, not necessarily all library plans and goals are related by this function. Here we have distinguished three categories relating to the state of agent plans. Others (such as, for example, the suspended plans of PRS [69]) can be added similarly.

\[
\begin{align*}
\text{PlanAgent} & : \text{P Goal} \\
\text{StoreAgentState} & : \text{P Goal} \\
\text{goallibrary, planlibrary, activeplans, executableplans} & : \text{P Plan} \\
\text{activeplangoal, plangoallibrary} & : \text{Goal} \rightarrow \text{P Plan} \\
\end{align*}
\]

\[
\begin{align*}
\text{dom activeplangoal} & \subseteq \text{goals} \land \bigcup (\text{ran activeplangoal}) = \text{activeplans} \\
\text{dom plangoallibrary} & \subseteq \text{goallibrary} \land \bigcup (\text{ran plangoallibrary}) \subseteq \text{planlibrary} \\
\text{goals} & \subseteq \text{goallibrary} \land \text{executableplans} \subseteq \text{activeplans} \subseteq \text{planlibrary}
\end{align*}
\]

Agents can be constrained in their design by including additional predicates. For example, it is possible to restrict the active plans of an agent with respect to a current goal, to those which are related to that goal by the function plangoallibrary. This can be achieved in the specification of such an agent by including the following predicate:

\[
\forall ag : \text{PlanAgent}; g : \text{Goal}; ps : \text{Plan} \bullet (g, ps) \in ag.\text{activeplangoal} \Rightarrow \\
(\exists qs : \text{Plan} \mid qs \subseteq ag.\text{plangoallibrary} \bullet (g, (ps \cup qs)) \in ag.\text{plangoallibrary})
\]

The actions performed by such agents are a function of their plans as well as their goals and perceptions, as described by the PlanAgentAction schema that refines the AgentAction schema, which specifies the selection of actions for a plan-agent. Other definitions formalising the perception of store-agents can be specified similarly.

\[
\begin{align*}
\text{PlanAgentAction} & : \text{P Goal} \rightarrow \text{P Plan} \rightarrow \text{View} \rightarrow \text{Environment} \rightarrow \text{P Action} \\
\forall gs : \text{P Goal}; ps : \text{Plan}; v : \text{View}; env : \text{Environment} \bullet \\
(\text{planagentactions gs ps v env}) & \subseteq \text{capabilities} \\
\text{dom planagentactions} & = \{\text{goals}\} \\
\text{dom (planagentactions goals)} & = \{\text{executableplans}\}
\end{align*}
\]

### 6.4.3 Multi-Agent Plans

In order for agents to reason about plans involving others it is necessary to analyse the nature of the plans themselves. This involves defining first the components of a plan, and then the structure of a plan. The components, which we call plan-actions, each consist of a composite-action and a set of related entities as described below. The structure of plans defines the relationship of the component plan-actions to one another. For example, plans may be total and define a sequence of plan-actions, partial and place a partial order on the performance of plan-actions, or trees and, for example, allow choice between alternative plan-actions at every stage in the plan’s execution.
Composite-Actions and Plan-Actions

We identify four types of action that may be contained in plans, which we call primitive, template, concurrent-primitive and concurrent-template. There may be other categories and variations on those we have chosen but not only do they provide a starting point for specifying systems, they also illustrate how different representations can be formalised and incorporated within the same model. A primitive action is simply a base action as defined in the agency framework. An action template provides a high-level description of what is required by an action, defined as the set of all primitive actions that may result through an instantiation of that action-template. An example where the distinction is manifest is in dMARS, where template actions would represent action formulae containing free variables. Once all the free variables are bound to values, the action is then a primitive action and can be performed. We also define a concurrent-primitive action as a set of primitive actions to be performed concurrently and a concurrent action-template as a set of template actions that are performed concurrently. We define a new type, ActionComp, as a compound-action to include all four of these types.

\[
\begin{align*}
Primitive & \rightarrow Action \\
Template & \rightarrow P\ Action \\
ConcPrimitive & \rightarrow P\ Action \\
ConcTemplate & \rightarrow P(P\ Action) \\
ActionComp & \rightarrow Prim(P\ Primitive) \\
& \mid Temp(P\ Template) \\
& \mid ConcPrim(P\ ConcPrimitive) \\
& \mid ConcTemp(P\ ConcTemplate) \\
\end{align*}
\]

Actions must be performed by entities, so we associate every composite-action in a plan with a set of entities, such that each entity in the set can potentially perform the action. At some stage in the planning process this set may be empty, indicating that no choice of entity has yet been made. We define a plan-action as a set of pairs, where each pair contains a composite-action and a set of those entities that could potentially perform the action. Plan-actions are defined as a set of pairs rather than a single pair so that plans containing simultaneous actions can be represented.

\[
PlanAction \rightarrow P( ActionComp \times P\ EntityModel )
\]

The following examples illustrate this representation. First, action, \( a_1 \), is to be performed by either the plan-agent itself or the entity \( entity1 \). The second example describes the two separate actions, \( a_2 \) and \( a_2 \), being performed simultaneously by the two entities \( entity1 \) and \( entity2 \) respectively. Then, the third example states that the actions \( a_3 \) and \( a_3 \) are to be performed simultaneously. No entity has been established as a possibility to perform \( a_3 \), and \( a_3 \) is to be performed by either \( entity2 \) or \( entity3 \).

1. \( \{(a_1, \{self, entity1\})\} \)
2. \( \{(a_2, \{entity1\}), (a_2, \{entity2\})\} \)
3. \( \{(a_3, \{\}), (a_3, \{entity2, entity3\})\} \)

Three auxiliary functions, useful for analysing composite-actions and plan-actions are defined below: actions returns the set of actions from an composite-action; actionsofPA returns the set of actions of a plan-action; and entitiesofPA returns the set of entity-models of a plan-action.
Plan Structure

We specify three commonly-found categories of plan according to their structure as discussed earlier. Other types may be specified similarly. 2

• **Partial Plans.** A partial plan imposes a partial order on the execution of actions, subject to two constraints. First, an action cannot be performed before itself and, second, if plan-action \(a\) is before \(b\), \(b\) cannot be before \(a\). Formally, a partial plan is a relationship between plan-actions such that the pair \(\langle a, a\rangle\) is not in the transitive closure and, further, if the pair \(\langle a, b\rangle\) is in the transitive closure of the relation then the pair \(\langle b, a\rangle\) is not. The standard definition of the transitive closure of a relation can be found in Appendix A.2.

• **Total Plans.** A plan consisting of a total order of plan-actions is a total plan. Formally, this is represented as a sequence of plan-actions.

• **Tree Plans** A plan that allows a choice between actions at every stage is a tree. In general, a tree is either

1. a leaf node containing a plan-action, or
2. a fork containing a node, and a (non-empty) set of branches each leading to a tree.

These are formalised as follows replacing the definition of \(Plan\) as a given set by a free-type definition to include the three plan categories thus defined.

\[
\begin{align*}
\text{PartialPlan} &= \{ ps : PlanAction \leftrightarrow PlanAction \mid \forall a, b : PlanAction \bullet (a, a) \notin ps^+ \land (a, b) \in ps^+ \Rightarrow (b, a) \notin ps^+ \bullet ps \} \\
\text{TotalPlan} &= \text{seq PlanAction} \\
\text{TreePlan} &= \text{Tip PlanAction} \\
&\quad | \text{Fork PlanAction} \\
\text{Plan} &= \text{Partial Plan} \\
&\quad | \text{Total TotalPlan} \\
&\quad | \text{Tree TreePlan}
\end{align*}
\]

Next we define auxiliary functions, useful for analysing plans. The \(planPairs\) function returns the plan-actions of a plan, \(planActions\) returns the set of actions contained in a plan, and \(planEntities\) returns the set of entities included in a plan. These definitions invoke the function, \(TreeNodes\), which takes a tree-plan and returns all the action-plans in that tree. This, in turn, relies on \(mapset\), which applies a function to every element in a set.

---

2 For example, PRS plans [69] contain goals as specified in Chapter 8.
6.4.4 Multi-Agent Plan-Agents

For single-agent systems all the actions of an executable plan must be within its capabilities. For such an agent, sap, all plans must therefore satisfy the following predicate.

\[ \forall sap : \text{PlanAgent}; plan : \text{Plan} \mid plan \in sap.planlibrary \cdot \text{planactions plan} \subseteq sap.capabilities \]

Now, plan-agents in multi agent systems on the other hand can consider executing plans containing actions not within their capabilities as long as they can model the capabilities of others. However, agents only able to model entities at the object level cannot make informed decisions about plan suitability since they would attempt to involve other entities without regard to their agency or autonomy. However, if agents can distinguish agency and autonomy, they can identify neutral-objects as most appropriate for using in plans. We refer to such agents as multi-agent plan-agents since they are able to evaluate the potential of involving other agents to execute their plans. This is similar to existing descriptions such as that proposed by Lux and Steiner [105], who describe multi-agent plans as plans that are executed by more than one agent.

Clearly, any entity included in an agent’s plan must be modelled by that agent. Further, any entity must be able to perform the actions with which it is associated in a plan according to the models of the plan-agent. The precise relationship of a plan-action to an entity depends on the type of plan-action.

- If the action is a primitive then it must be an element of the capabilities of the associated entities.
- If the action is a concurrent-primitive then the set of actions must be a subset of the capabilities of the associated entities.
• If the action is a template-action, then at least one of the actions that belong to this set must be within the capabilities of the associated entities.

• If the action is a concurrent-template action then each of the template-actions must contain at least one action in the capabilities of any associated entity.

The MultiAgentPlanAgent schema which refines PlanAgent and includes AgentModels, defines such an agent and includes these constraints.

\[
\text{MultiAgentPlanAgent}\]

\[
\text{AgentModels} \\
\text{PlanAgent}
\]

\[
\forall p : \text{planlibrary} \bullet \text{planentities } p \subseteq \text{modelobjects} \\
\forall p : \text{planlibrary}; ac : \text{ActionComp}; em : \text{EntityModel}; \\
ems : \mathbb{P} \text{EntityModel} \mid (ac, ems) \in \bigcup \text{(planpairs } p) \bullet \\
ac \in \text{ran Prim} \Rightarrow \left( \forall em : ems \bullet \text{actions } ac \subseteq \text{em.capabilities} \right) \land \\
ac \in \text{ran ConcPrim} \Rightarrow \left( \forall em : ems \bullet \text{actions } ac \subseteq \text{em.capabilities} \right) \land \\
ac \in \text{ran Temp} \Rightarrow \left( \forall em : ems \bullet \text{actions } ac \cap \text{em.capabilities} \neq \{ \} \right) \land \\
ac \in \text{ran ConcTemp} \Rightarrow \left( \forall em : ems \bullet \forall as : \mathbb{P} \text{Action} \right) \\
as \in \text{ConcTemp}^{-1}ac \bullet as \cap \text{em.capabilities} \neq \{ \}
\]

6.4.5 Sociological Plan-Agents

If multi-agent plan-agents are also sociological agents then we claim that they can make more informed choices about plan selection. We take such an agent to be a sociological plan-agent as defined simply in the SociologicalPlanAgents schema.

\[
\text{SociologicalPlanAgent}\]

\[
\text{SociologicalAgent} \\
\text{MultiAgentPlanAgent}
\]

To illustrate the greater reasoning capacity of sociological agents over their non-sociological counterparts we define the following categories of plans and goals, which can be determined by a sociological plan-agent but not by a non-sociological one. Note, all these categories are with respect to the models of the sociological plan-agent.

• A self-sufficient plan is any plan that involves only neutral-objects, server-agents the plan-agent owns, and the plan-agent itself. Self-sufficient plans can therefore be executed without regard to other agents and exploit the current agency relationships. Formally, this category is defined in the SelfSuffPlan schema where selfsuff represents the self-sufficient plans of an agent. This schema uses the relational image operator defined in Appendix A.2. In general, the relational image \( R(\{ S \} \) of a set \( S \) through a relation \( R \) is the set of all objects \( y \) to which \( R \) relates to some member \( x \) of \( S \).

\[
\text{SelfSuffPlan}\]

\[
\text{ModelAgencyRelationships} \\
\text{SociologicalPlanAgent} \\
\text{selfsuff} : \mathbb{P} \text{Plan}
\]

\[
\text{selfsuff} = \{ p : \text{planlibrary} \mid \text{planentities } p \subseteq \\
\text{modelneutralobjects} \cup \text{modelself} \cup \text{modelowns}(\text{modelself} \mid \bullet p) \} 
\]
• A *self-sufficient goal* is any goal in the goal library that has an associated self-sufficient plan. These goals can then, according to the agent’s model, be achieved independently of the existing social configuration. Formally, a goal is self-sufficient if, according to the function, \( plangoallibrary \), there is an associated self-sufficient plan.

\[
\text{SelfSuffGoal} \\
\text{SelfSuffPlan} \\
\text{selfsuffgoal} : \mathcal{P} \text{ Goal} \\
\text{selfsuffgoal} = \{ g : \text{goallibrary} | \exists p : \text{Plan} \mid p \in \text{plangoallibrary} \ g \bullet p \in \text{selfsuff} \} \bullet g
\]

• A *reliant-goal* is any goal that has a non-empty set of associated plans that is not self-sufficient. Formally, a goal is reliant if no plan in the non-empty set of associated plans as determined by \( plangoallibrary \) is self-sufficient.

\[
\text{ReliantGoal} \\
\text{SelfSuffGoal} \\
\text{reliantgoal} : \mathcal{P} \text{ Goal} \\
\text{reliantgoal} = \{ g : \text{goallibrary} | \text{plangoallibrary} \ g \neq \emptyset \} \land \\
\neg (\exists p : \text{Plan} \mid p \in \text{plangoallibrary} \ g \bullet p \in \text{selfsuff} ) \bullet g
\]

For each plan that is not self-sufficient, a sociological plan-agent can establish those autonomous agents that may be affected by its execution. The number of such agents is an important criterion when a plan needs to be selected from competing alternative active plans. An autonomous agent may be affected by a plan in one of two ways: either it is required to perform an action directly, or it is engaging a server-agent required by the plan. In this latter case, a sociological plan-agent can reason about either persuading \( A \) to share or release \( S \), taking \( S \) without permission, or finding an alternative server-agent or plan as discussed in the previous section. In order that sociological agents can analyse their plans in more detail we introduce further definitions.

• The *cooperating autonomous agents* of a plan are those autonomous agents, other than the plan-agent itself, that are involved in performing actions of that plan. These agents will need to cooperate with the plan-agent for the plan to be executed. Formally, an agent is a cooperating autonomous agent with respect to a plan, if it is contained in the set entities required for the plan.

\[
\text{CooperatingAgents} \\
\text{ModelAgencyRelationships} \\
\text{SociologicalPlanAgent} \\
\text{cooperatingagents} : \text{Plan} \rightarrow \mathcal{P} \text{ AutonomousAgentModel} \\
\forall p : \text{Plan} \bullet \text{cooperatingagents} p = \\
\{ a : \text{modelautonomousagents} | a \in \text{planentities} p \bullet a \} \setminus \text{modelsel}
\]

• The *affected autonomous agents* of a plan are those autonomous agents, other than the plan-agent itself, that are engaging an entity required in the plan. Formally, an autonomous agent is affected with respect to a plan if there exists a server-agent contained in the set of entities required by the plan that is currently engaged by the autonomous agent. These agents may need to cooperate with the plan-agent. Notice that the affected autonomous agents do not include the cooperating agents.
6.4.6 An Illustrative Example

To illustrate the value to a sociological plan-agent of being able to analyse plans using the categories above, consider an autonomous sociological plan-agent, $A$, and suppose that it models the agent relationships in its environment as follows. Autonomous agent $B$ directly owns the server-agent $S2$ and directly engages $S3$, autonomous agent $C$ directly engages $S3$, and $A$ directly owns $S1$. In addition, in $A$’s view, $O1$ and $O2$ are neutral-objects. This agent configuration can be seen in Figure 6.1 and would be represented in $A$’s models as follows.

\[
\{A, B, C\} \in \text{modelautonomousagents}
\]
\[
\{A\} = \text{modelself}
\]
\[
\{S1, S2, S3\} \in \text{modelserversagents}
\]
\[
\{O1, O2\} \in \text{modelneutralobjects}
\]
\[
\{(A, S1), (B, S2), (B, S3), (C, S3)\} \subseteq \text{modeldirengagements}
\]
\[
\{(A, S1), (B, S2)\} \subseteq \text{modeldirengagements}
\]
Consider also that agent $A$ generates the goal, $g_A$, and activates four total plans $p_1$, $p_2$, $p_3$ and $p_4$ to achieve $g_A$ as follows. The four plans are then in the set of active plans, and the pair $(g_A, \{p_1, p_2, p_3, p_4\})$ is in the function $\text{activeplangoal}$ relating current goals to candidate active plans.

$$\{g_A\} \subseteq \text{goals}$$
$$\{p_1, p_2, p_3, p_4\} \subseteq \text{activeplans}$$
$$(g_A, \{p_1, p_2, p_3, p_4\}) \in \text{activeplangoal}$$

In addition, suppose the plans are as follows.

$$p_1 = \text{Total} \{\{(a_1, \{B, C\}), (a_2, \{A\})\}, \{(a_3, \{S2, S3\})\}\}$$
$$p_2 = \text{Total} \{\{(a_1, \{01\}), (a_1, \{A\})\}, \{(a_1, \{S1\}), (a_2, \{A\})\}, \{(a_3, \{01\}), (a_3, \{A\})\}, \{(a_1, \{S1\}), (a_4, \{A\})\}, \{(a_1, \{S1\}), (a_4, \{A\})\}, \{(a_1, \{S1\}), (a_4, \{A\})\},\{(a_{15}, \{01\}), (a_5, \{A\})\}\}$$
$$p_3 = \text{Total} \{\{(a_1, \{A\})\}, \{(a_2, \{S3\})\}\}$$
$$p_4 = \text{Total} \{\{(a_1, \{A\})\}, \{(a_2, \{S2\})\}\}$$

Notice, that since in the plan, $p_1$, the action, $a_1$, can be performed by either the agents, $B$ or $C$, and the action, $a_3$, by either $S2$ or $S3$, there are four possible ways of executing this plan. We can represent these by $p_{1_1}$, $p_{1_2}$, $p_{1_3}$ and $p_{1_4}$ as follows.

$$p_{1_1} = \{\{(a_1, \{B\}), (a_2, \{A\})\}, \{(a_3, \{S2\})\}\}$$
$$p_{1_2} = \{\{(a_1, \{B\}), (a_2, \{A\})\}, \{(a_3, \{S3\})\}\}$$
$$p_{1_3} = \{\{(a_1, \{C\}), (a_2, \{A\})\}, \{(a_3, \{S2\})\}\}$$
$$p_{1_4} = \{\{(a_1, \{C\}), (a_2, \{A\})\}, \{(a_3, \{S3\})\}\}$$

The agent then has seven alternative plans for execution selection. Now, by inspection, the entities required by the plan $p_2$ are $A$, $S1$ and $O1$.

$$\text{planentities}_p = \{A, S1, 01\}$$

The previous definition of a self-sufficient plan for an agent $A$ is any plan that only requires neutral-objects, agents owned by $A$, and $A$ itself. In this case the union of the set of neutral-objects, owned agents and $A$ itself is simple to calculate.

$$\text{modelneutralobjects} \cup \text{modelself} \cup \text{modelowns} \cup \text{modelself} = \{O1, O2, O3, A, S1\}$$

The set of entities required by the plan is a subset of this set which means that the plan $p_2$ is self-sufficient as is the associated goal $g_A$. 

Figure 6.1: Example: A Sociological Agent’s Model
<table>
<thead>
<tr>
<th>Plan</th>
<th>$p_{11}$</th>
<th>$p_{12}$</th>
<th>$p_{13}$</th>
<th>$p_{14}$</th>
<th>$p_{15}$</th>
<th>$p_{16}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>cooperating agents</td>
<td>${B}$</td>
<td>${B}$</td>
<td>${C}$</td>
<td>${C}$</td>
<td>${}  $</td>
<td>${}  $</td>
</tr>
<tr>
<td>autoagents affected</td>
<td>${B, C}$</td>
<td>${B}$</td>
<td>${B, C}$</td>
<td>${C}$</td>
<td>${B, C}$</td>
<td>${B}$</td>
</tr>
</tbody>
</table>

Table 6.1: Example: A Sociological Agent’s Evaluation of its Plans

\[
\{A, S1, 01\} \subseteq \{O1, O2, O3, A, S1\} \Rightarrow p_2 \in selfsuff
\]
\[
p_2 \in selfsuff \Rightarrow g_A \in selfsuffgoal
\]

A is thus able to achieve $g_A$ without affecting other autonomous agents and can act without regard to them whilst exploiting the current set of agency relationships. However, it may decide that, even though $p_2$ is self-sufficient, it involves too much physical effort, dismisses this possibility, and evaluates the six other alternatives to give the information shown in Table 6.1. It can be seen that the least-fuss and least-direct-fuss plans are as follows.

\[
leastfuss\  g_A = \{p_{12}, p_{14}, p_4\} \land leastdirectfuss\  g_A = \{p_3, p_4\}
\]

Based on this analysis, $p_4$ may seem like the best candidate plan for execution since it does not involve the direct cooperation of other entities, and only affects one autonomous agent, $B$. The plan-agent can then analyse options concerning how to engage $S2$, as discussed previously. Clearly, the final decision about plan selection will be based on other considerations also such as the motivations of the plan-agent, and its models of the motivations of others affected by plans. However, we have shown how sociological agents can use the plan categories defined in this section as an important criteria for evaluating alternative active plans.

### 6.4.7 Modelling the Plans of Others

If agents can model the plans of others, or produce agreed multi-agent plans, then they can coordinate their actions in order to achieve their local goals more effectively. In fact, many authors argue that agents must model the plans of others for effective coordination to occur [82, 157]. Agents can then take advantage of the plans of others to avoid duplication of effort and to avoid conflict, which arises, for example, when two agents require direct ownership of the same entity at the same time.

Once agents are designed with the ability to reason about the plans of other agents, bargaining can take place between agents able to help each other in their plans. As an example, suppose agent $A$ has a plan that necessarily involves the cooperation of $B$. It may be appropriate for $A$ to consider the plans of $B$ that involve $A$’s cooperation since $A$ may then realise that $B$ has a high-priority plan that can only be achieved with $A$’s cooperation. In this case $A$ would consider herself to be in a strong bargaining position. This level of modelling, where agents can model the plans of others, has been mapped out in the work of Social Dependence Networks, which are considered in detail in Chapter 9.

The actual level at which other agents are modelled is clearly critical in directing behaviour. For example, a sociological agent with models of other agents as non-sociological may realise that these agents are unable to recognise agency relationships. The sociological agent may then be concerned that these other agents may use entities that destroy their own existing agency relationships.

Again we can develop models of increasingly more sophisticated agents incrementally. For example, a sociological agent able to model the plans of others. Such an agent is defined below in the schema, **SociologicalPlanAgentModelsPlanAgent**. Finally, the schema which defines agents that can model others as sociological plan-agents is **SociologicalPlanAgentModelsSociologicalPlanAgent**.
6.5 Summary and Conclusions

Having identified a set of inter-agent relationships that underlie all multi-agent systems, this chapter has provided models of the requisite agent dimensions for effective social functioning in such systems. We refer to these models collectively as the sociological agency model. The specification structure used to define this model can be found in Appendix B.

We have developed models of the following dimensions by incrementally developing the agency framework to specify agents that:

- have an internal store;
- can model others in their environment;
- can model the agency relationships in their environment; and
- can evaluate competing plans with respect to their model of the current agency relationships.

Models are critical because effective social agents need to represent the inter-agent relationships in the environment in which they are situated. Such agents are able to reason about acting to exploit these relationships without inadvertently or unnecessarily affecting them. They can then make more enlightened choices between alternative plans based not only on the entities involved but the current agency obligations of the entities involved.

In general, we have shown that computational agents able to recognise the agency relationships in their environment can reason with much more sophistication than those agents who can only model other agents. We call these agents sociological agents and have shown the increased functionality of such agents, and we argue that effective agents must necessarily be sociological.

Our model of deliberative, sociological agents has been developed by extending the agency framework justifying the claim that it can accommodate both deliberative and reflexive agents equally. We argue that the model developed is still generally applicable since care has been taken to ensure that the chosen representation of plans can be applied, and easily extended where necessary, to existing theories and systems. (Details of aspects of this work can be found in [34, 40] as well as in Chapter 9.)

Not only have we refined the schemas of the agency framework to develop our model of sociological agency, but we have shown how it necessarily impacts on the design of effective agents. Similarly, we have shown that the agency relationships previously identified affect the development of generally effective social agents who must recognise, exploit and manipulate them. The models we have developed in this thesis can, therefore, not only be used to reason about agents and their relationships, but, significantly, in developing formal descriptions of the agents themselves.
Chapter 7

The Contract Net as a Goal Directed System

7.1 Introduction

We claim that the agency framework, agency relations model and sociological agency model together provide a strong and fundamental structure that can be applied directly to describe and analyse multi-agent systems and theories. In order to demonstrate this applicability we must show that our work is directly relevant to such systems and theories by deriving models that highlight relevant aspects of them. Such models, founded from a single framework, can then be more readily evaluated, compared and integrated. In this chapter, we complete the path from our initial framework to the modelling of specific entities in the theory and practice of multi-agent systems, by describing a mechanism for implementing multi-agent systems known as the contract net protocol, to show how the agency framework and subsequent models relate directly to it and how they can be applied.

We choose the contract net protocol [27, 152, 153] as the subject of this chapter. According to Parunak, it is the most common protocol between agents in both real applications and detailed simulations based on real applications [128]. Several extensions from the contract net protocol [127, 141] have been proposed and there have been attempts at formalisation, by authors such as Werner [165] and Wooldridge [170]. Another indication of the centrality of the contract net protocol in multi-agent systems, and the extent to which it is widely understood, is the number of authors, such as Wooldridge and Fisher [60, 170], Müeller [118], and McCabe and Clark [109] who use it to demonstrate the applicability of their own theories and models.

The contract net protocol is important because it is very definitely situated in the practical and experimental camp. Moreover, it is relatively well-defined and understood, and hence very suitable to be used as an exemplar for the kind of work described here. In specifying the contract net protocol, we aim to provide a bridge between the formality on the one hand and the practical work on the other.

7.2 Contract Net Protocol

As discussed in Section 2.5, the contract net provides a mechanism where nodes can dynamically enter into relationships in response to the current processing requirements. It has been used in manufacturing, transport systems and assembly problems, which are described in more detail by Parunak [128].

In this chapter, we take a contract net (CN) to be any system that uses the contract net protocol (CNP) as the means by which opportunistic task allocation can take place. A CN is thus a collection of components, or equally, nodes, with no pre-determined control hierarchy, which are able to take on roles dynamically according to contracts. A contract is an agreement between a manager and a contractor that results when the contractor node has successfully bid for the contract. The contractor node has agreed to perform a task for the manager node who is responsible for monitoring the execution of the task and processing any results.
Contracts are established by mutual agreement between two nodes after an exchange of information. The structure of this exchange is determined beforehand by the CNP, which is illustrated in Figure 7.1. The CNP is initiated when a node is given the responsibility to undertake and satisfy a newly generated task. These tasks are decomposed into sub-tasks and, when there may be inadequate knowledge or data to undertake these sub-tasks directly, they are offered for bidding by other agents. This is achieved by a task announcement message, which is broadcast advertising the existence of this task to other nodes. The message includes information relating to the task which enables other nodes to evaluate whether they are suited to perform that task. For example, when the CNP is applied to the simulation of a distributed sensing system a task announcement includes three types of information [151].

Task Abstraction Slot specifies the identity and position of the manager that enables potential contractors to reply to the task announcement.

Eligibility Specification specifies the location and capabilities required by any bidders.

Bid Specification indicates that a bidder must specify its position and its sensing capabilities.

In response to the current task announcements, nodes can evaluate their interest using task evaluation procedures specific to the problem at hand. If a node has sufficient interest in a task announcement, it will submit a bid to undertake to perform the task. The owner of the task announcement (called the manager) selects nodes using bid evaluation procedures based on the information supplied in the bid. It sends award messages to successful bidders who then become contractors to the manager. These contractors may in turn subcontract parts of their task by announcing their own task announcements and the CNP is repeated with the contractor eventually becoming a manager for these sub-contracts. This leads to a hierarchical configuration in the CN. The contractors issue reports to the manager, which may be interim reports or final reports containing a result description. The manager finally terminates a contract with a termination message.

A key feature of the case study in this chapter is the ease with which the CN can be accommodated by the agency framework and agency relations model. Section 7.3 provides an analysis of the components that can be found in a CN and Section 7.4 analyses the inter-agent relationships. The additional
functionality required by CN agents that enables them to take part in the CNP is then considered in Section 7.5, which includes the ability of CN nodes to analyse task announcements and make bids for contracts. This enables the state of a CN during the CNP to be presented. The CNP itself is then modelled in Section 7.6 as operations on this system state, which describes how bids and task announcements are made and how contracts are awarded and terminated. The final section contains conclusions of this chapter.

7.3 Contract Net Components

We refine the agency framework to arrive at a formal specification of the CN thereby retaining the structure of the agency framework. The first task is to specify the different kinds of component that can be found in a system employing the CNP. Three categories of entity are important: CN nodes, CN agents, and CN monitor nodes, which are refinements of objects, agents and autonomous agents, respectively.

7.3.1 Nodes

A node in the CN has capabilities and attributes such as its processing potential and physical location, for example. A CN Node is therefore simply an object. Formally, it is defined in ContractNetNode, which simply includes the Object schema.

```
ContractNetNode
  Object
```

7.3.2 Agents

A CN agent is a CN node that performs a task or, equally, can be ascribed a set of goals. Such agents are defined by ContractNetAgent, which includes the ContractNetNode and Agent schemas.

```
ContractNetAgent
  ContractNetNode
  Agent
```

7.3.3 Monitor Agents

Davis and Smith [27] also describe a single-processor node in a distributed sensing example, called a monitor node, that starts the initialisation as the first step in the CN operation. If this is just a node that passes on information to another, then it is no different from the manager specified above. However, if it generated the goal or task to perform, then it is autonomous. In this case monitors are defined in the Monitor schema, which includes the AutonomousAgent and ContractNetAgent schemas.

```
Monitor
  AutonomousAgent
  ContractNetAgent
```

A CN then comprises a set of CN nodes of which at least two are CN agents, and at least one of these is a monitor agent.
This schema is analogous to *MASystem* in Section 4.2 which defines the set of entities in a multi-agent system. This feature can be exploited directly by using the Z language, which allows the renaming of schema components. Thus, it is possible to rename the components of the *MASystem* schema to define the *AllComponents* schema so that the schema below is identical to the one above.

The agency framework also specifies neutral-objects and server-agents, which describe *idle nodes* and CN server-agents, respectively.

### 7.3.4 Idle Nodes

We model an *idle node* as a neutral-object. Formally, it is defined by *IdleNode* and includes the schemas *ContractNetNode* and *NeutralObject*.

### 7.3.5 Server-Agents

For completeness, we define CN server-agents as CN agents that are not autonomous. Such agents are defined by *ContractServerAgent*, which includes *ContractNetAgent* and *ServerAgent*.

### 7.3.6 Contract Net Components

Nodes in a CN are either idle or CN agents, and CN agents are either CN monitors or CN server-agents. A more detailed definition of the CN, which includes idle nodes and CN server-agents, is thus provided in the next schema. The union of *monitors* and *contractserveragents* is equal to *contractagents* and the union of *idlenodes* and *contractnodes* is equal to *nodes*.
In the same way that *AllComponents* is defined by renaming the components of *MASystem*, *ContractNetEntities* can be defined by directly applying *MultiAgentSystem*, presented in Section 4.2.

This completes the description of the components in a CN and we proceed to an analysis of the *inter-agent relationships* in a CN in terms of node dependencies by application of the agency relationship model.

### 7.4 Contract Net Relationships

When a node is awarded a contract by another the first node is *engaged* by the second. The first node becomes the contractor, and the second the manager of the contract. The contractor then performs the task of the contract for the manager and sends back progress reports and results. Tasks specify a state of affairs to be achieved and so have the same type as a goal in the agency framework.

\[
\text{Task} \equiv \text{Goal}
\]

A contract can be represented as a task, and two distinct nodes called the manager and the contractor. Necessarily both agents must be ascribed the task of the contract. Formally, a contract is defined by the *Contract* schema, which includes the variables *task*, *manager* and *contractor* such that *task* is included in the intersection of the goals of the manager and contractor. Whilst the manager of a contract may be autonomous the contractor must be a server-agent.

\[
\text{Contract} \\
\text{task} : \text{Task} \\
\text{manager} : \text{ContractNetAgent} \\
\text{contractor} : \text{ContractServerAgent} \\
\text{manager} \neq \text{contractor} \\
\text{task} \in (\text{manager.goals} \cap \text{contractor.goals})
\]

A contract is thus a specific type of *direct engagement* in which the client of the engagement is the manager of the contract, the server is the contractor, and the goal of the engagement is the task of the contract. In this way, contracts can be defined by renaming the components of a direct engagement.
The configuration of a CN is the result of the set of all contracts between agents. An example of a possible CN configuration is shown in Figure 7.2. In this example node \( C \) has been awarded a contract by \( A \) and, in turn, has awarded a contract to \( I \). Formally, we model the set of contracts in the \( \text{ContractNetRelations} \) schema. The set of managers are those nodes that are managing a current contract, and the set of contractors are defined similarly. The predicate asserts that the combination of all contractors and managers from all contracts is equal to the set of all CN agents.

Again, this schema can be derived by renaming the components of the \( \text{SystemEngagements} \) schema defined in Section 4.5.

Engagement chains as well as engagements arise naturally in a CN. For example, in Figure 7.2, whilst \( H \) may be engaging \( N \) with a task that is unrelated to the contract between \( B \) and \( H \), it is also possible that the sequence of nodes \( \langle B, H, N \rangle \) is an engagement chain. This arises when the contract between \( H \) and \( N \) is a direct consequence of the agency relationship between \( B \) and \( H \). In fact, modelling the CN as engagement chains is exactly right, since these reveal the inter-node dependencies for any task, and reveal the flow of contracts as they are formed between agents. Moreover, if \( A \) and \( B \) are autonomous monitor nodes then \( A \) may adopt the task of \( B \) to be cooperating with \( B \) as illustrated in Figure 7.3. It is therefore possible that cooperations as well as engagements can exist in a CN depending on the autonomy of the agents involved.
7.5 Contract Net State

We now develop our model further to describe the state of the CN which, as well as including the components and their relationships, also includes the set of task announcements and bids.

7.5.1 Task Announcements

A node intending to become the manager of a contract for one of its tasks, first makes a task announcement that includes information relating to the task. In particular, it provides an eligibility specification that states the attributes and capabilities required to perform that task. A node is only eligible for a task if its actions and attributes satisfy the eligibility of the task announcement. Formally, Eligibility is a type comprising a set of actions and attributes satisfy and therefore has the same type as an object.

\[ \text{Eligibility} \equiv \text{Object} \]

A task announcement is issued by a sender to a set of recipients to request bids for a particular task from agents that match eligibility requirements. The sender must be an agent, whereas the recipients can be any CN node.

\[ \text{Sender} \equiv \text{ContractNetAgent} \]
\[ \text{Recipient} \equiv \text{ContractNetNode} \]

A task announcement is defined below, and comprises the prospective manager, sender, the non-empty set of recipients, which do not include the sender, recipients, the task to be performed, task, and an eligibility requirement, eligibility. Notice that the combination of a task together with an eligibility is, in fact, an agency requirement since it specifies attributes, capabilities and goals.

```
TaskAnnouncement
sender : Sender
recipients : \text{P}_1 \, \text{Recipient}
task : Task
eligibility : Eligibility
```

\[ \text{sender} \not\in \text{recipients} \]
7.5.2 Bids

In response to a task announcement, agents can evaluate their interest in the task by using task evaluation procedures. If there is sufficient interest after evaluation, agents will submit a bid to undertake to perform the task. A bid involves a node describing a subset of itself in response to an eligibility specification, which will be used in evaluating the bid. A bid is formalised by the schema, \textit{Bid}, which includes the associated node and its eligibility.

\begin{align*}
\text{Bid} & \quad \text{bidnode} : \text{Recipient} \\
& \quad \text{eligibility} : \text{Eligibility} \\
& \quad \text{eligibility.capabilities} \subseteq \text{bidnode.capabilities} \\
& \quad \text{eligibility.attributes} \subseteq \text{bidnode.attributes}
\end{align*}

7.5.3 System State

The state of the CN can now be represented as the current set of nodes, contracts, task announcements and bids. Each task announcement will have associated with it a set of bids, which are just eligibility specifications as described above. The following schema defines the state of a CN and includes the set of components, contracts, and the set of task announcements and associated bids. Every bid associated with a task announcement must have been made by a node that is in the list of recipients of that task announcement. The redundant variable, \textit{taskannouncements}, defines the set of current system task announcements.

\begin{align*}
\text{ContractNetState} & \quad \text{ContractNetRelations} \\
& \quad \text{bids} : \text{TaskAnnouncement} \rightarrow \mathbb{P} \text{Bid} \\
& \quad \text{taskannouncements} : \mathbb{P} \text{TaskAnnouncement} \\
& \quad \text{taskannouncements} = \text{dom} \text{ bids} \\
& \forall t : \text{TaskAnnouncement}; \ b : \text{Bid}; \ bs : \mathbb{P} \text{Bid} \mid b \in bs \bullet \\
& \quad (t, bs) \in \text{bids} \Rightarrow b.\text{bidnode} \in t.\text{recipients}
\end{align*}

7.6 Contract Net Protocol

We can now specify the CNP as outlined in Figure 7.1 by specifying the changes to the state of the CN as the protocol progresses, by using operation schemas. Before this task is undertaken, however, we must specify some axiomatic definitions that are necessary for determining those tasks that nodes may be interested in, and how bids are ranked.

7.6.1 Axiomatic Definitions

First, each node has a means of deciding whether it is capable of, and interested in, performing certain tasks given certain eligibilities. This determines whether nodes bid for these tasks. The function, \textit{Interested}, determines whether a node can bid for a task, and is a function of the node, the task and the eligibility and returns a boolean value. Second, each node has a means of rating bids for the task announcements it has broadcast. This is achieved by \textit{Rating}, which maps the node making the rating, its task announcement and an associated bid to a natural number.

\begin{align*}
\text{Interested} : \text{ContractNetNode} \rightarrow \text{Task} \rightarrow \text{Eligibility} \rightarrow \text{Bool} \\
\text{Rating} : \text{ContractNetNode} \rightarrow \text{TaskAnnouncement} \rightarrow \text{Bid} \rightarrow \mathbb{N}
\end{align*}
In addition, we can define a CN node being instantiated with a new task, and a CN agent removing one of its goals easily in terms of the existing agency framework functions, EntityAdoptGoals and EntityRemoveGoals, respectively.

\[\text{ContractNodeNewTask} \equiv \text{EntityAdoptGoals}\]
\[\text{ContractNodeRemoveTask} \equiv \text{EntityRemoveGoals}\]

The former of these is used in the auxiliary function, makecontract, which forms a contract from its constituent components as long as the manager has task as a goal and the contractor does not. A contract is then formed to include the newly instantiated contractor agent.

\[
\text{makecontract} : \text{Task} \rightarrow \text{ContractNetAgent} \rightarrow \text{ContractNetNode} \rightarrow \text{Contract}
\]

\[
\forall t : \text{Task}; m : \text{ContractNetAgent}; c : \text{ContractNetNode}; con : \text{Contract} \mid
t \in (m.g\text{oals}) \land t \notin (m.g\text{oals}) \land m \neq c ::
makecontract t c m = con \iff
\begin{align*}
\text{con.task} & = t \\
\text{con.manager} & = m \\
\text{con.contractor} & = \text{ContractNodeNewTask}(c, \{t\})
\end{align*}
\]

7.6.2 Making Task Announcements

When a node makes a task announcement, there is no change to the node dependencies, so that the components and contracts of the CN remain unaltered. Any node that issues a task announcement must be an agent since it must have a current task. In addition, both the recipients and the sender must be nodes of the CN, the task must be in the goals of the sender, and the sender must not be able to satisfy the eligibility requirements of the task alone. As a result of this operation, a new system task announcement is created that is associated with an empty set of bids, since at this time none will have been offered for it. The operation is defined formally in the following schema, which changes \text{ContractNetState} but not \text{ContractNetRelations}. The preconditions of the operation ensure that the task announcement is well-defined, and that the announcing agent does not have the eligibility to perform the associated task. As a result of this operation, a mapping from the new task announcement to the empty set is included.

\[
\text{MakeTaskAnnouncement}
\]

\[
\begin{align*}
\triangle \text{ContractNetState} \\
\equiv \text{ContractNetRelations} & \\
\text{an?} & : \text{ContractNetAgent}; \text{ta?} : \text{TaskAnnouncement} \\
\text{an?} \in \text{nodes} & \land \text{ta?.recipients} \subseteq \text{nodes} \\
\text{ta?.sender} & = \text{an?} \land \text{ta?.task} \in \text{an?.goals} \\
\neg ((\text{ta?.eligibility}.\text{capabilities} \subseteq \text{an?.capabilities}) & \land \neg ((\text{ta?.eligibility}.\text{attributes} \subseteq \text{an?.attributes})) \\
\text{bids'} & = \text{bids} \cup \{(\text{ta?}, \{\})\} \\
\text{taskannouncements'} & = \text{taskannouncements} \cup \{\text{ta?}\}
\end{align*}
\]

7.6.3 Making Bids

In response to a task announcement, a node may make a bid as long as it is both eligible for, and interested in, the task. This bid is then added to the set of other bids that have been received in response to this task announcement. Again, the state of the inter-node relationships is unaffected by this operation, which is formalised below and which includes preconditions to ensure that nodes can
only bid for task announcements for which they are recipients. As a result of a node making a bid, the
set of task announcements does not change, but the bids associated with the task announcement are
updated to include the new bid. The final predicate defining this update to the bids function use the
relational override operator which is defined in Appendix A.2. Essentially, the relation $Q \uplus R$ relates
everything in the domain of $R$ to the same objects as $R$ does, and everything else in the domain of $Q$
to the same objects as $Q$ does.

### MakeBid

<table>
<thead>
<tr>
<th>MakeBid</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\triangle ContractNetState$</td>
</tr>
<tr>
<td>$\equiv ContractNetRelations$</td>
</tr>
<tr>
<td>$biddingnode? : ContractNode \land biddingnode? \in nodes$</td>
</tr>
<tr>
<td>$ta? \in taskannouncements \land biddingnode? \in ta?.recipients$</td>
</tr>
<tr>
<td>$ta?.eligibility.capabilities \subseteq bid?.eligibility.capabilities$</td>
</tr>
<tr>
<td>$ta?.eligibility.attributes \subseteq bid?.eligibility.attributes$</td>
</tr>
<tr>
<td>$Interested(biddingnode?(ta?.task)(ta?.eligibility) = True$</td>
</tr>
<tr>
<td>$taskannouncements' = taskannouncements$</td>
</tr>
<tr>
<td>$bids' = {ta?, bids ta? \cup {bid?}}$</td>
</tr>
</tbody>
</table>

### Awarding Contracts

**7.6.4** Awarding Contracts

After receiving bids, the issuer of a task announcement awards the contract to the highest rated bid. In order to choose the best bid with respect to a task announcement, the Rating function is used by
which the bid with the highest rating is selected and a contract is formed with that node. Since a new contract is formed, the set of CN relationships changes and now contains a contract for the task with
the issuer of the task announcement as manager, and the awarded bidder as contractor.

In the following schema, the set of agent relationships in the CN is altered as well as the state. The owner of the task announcement awards a contract to a bidder. As a result of the operation, the new contract formed between the owner, as manager, and the contract server-agent that results from
instantiating the bidding node with the additional task is added to the set of system contract server-agents. If, prior to the operation, the bidding agent was an idle node then it must be removed from
the set of idle nodes. If, however, the node was previously an agent then it must be removed from
the set of server-agents. Finally, the task announcement is removed from the set of task announce-
ments and from the domain of the bids function using the anti-domain restriction operator defined in
Appendix A.2.

### AwardContract

<table>
<thead>
<tr>
<th>AwardContract</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\triangle ContractNetState$</td>
</tr>
<tr>
<td>$\forall b : Bid \mid b \in bids ta? \bullet Rating man? \mid ta? \geq Rating man? \mid ta? \mid b$</td>
</tr>
<tr>
<td>$contracts' = contracts \cup {makecontract ta?.task man? \mid bid?.bidnode}$</td>
</tr>
<tr>
<td>$contractagents' = contractagents \cup$</td>
</tr>
<tr>
<td>${ContractNodeNewTask(bid?.bidnode,{ta?.task})}$</td>
</tr>
<tr>
<td>$bid?.bidnode \in idlenodes \Rightarrow idlenodes' = idlenodes \setminus {bid?.bidnode}$</td>
</tr>
<tr>
<td>$bid?.bidnode \in contracts\serveragents \Rightarrow$</td>
</tr>
<tr>
<td>$contractagents' = contractagents \setminus {bid?.bidnode}$</td>
</tr>
<tr>
<td>$taskannouncements' = taskannouncements \setminus {ta?}$</td>
</tr>
<tr>
<td>$bids' = {ta?} \smallsetminus bids$</td>
</tr>
</tbody>
</table>
7.6.5 Terminating Contracts

Finally, a manager can terminate a contract with one of its contractor nodes. The task of that contract is removed from the contractor’s goals and this (former) contractor may revert to being an agent with fewer goals, or an idle node. We specify the situation where the manager retains the task of the contract, though it may be possible that in some situations the task is also dropped by the manager. In the following schema, if the former contracted node is still an agent, then it remains in the set of CN agents. However, if the node had one task only, it is removed from the set of CN agents and added to the set of idle nodes.

```
TerminateContract
  ∆ ContractNetState
  contract?.task = t? ∧ contract?.manager = man?
  contract?.contractor = con? ∧ contract? ∈ contracts
  contracts' = contracts \ {contract?}
  ContractNodeRemoveTask (con?, {t?}) ∈ ContractNetAgent ⇒
    contractagents' = contractagents \ {con?} ∪
    {ContractNodeRemoveTask(con?, {t?})}
  ContractNodeRemoveTask (con?, {t?}) ∉ ContractNetAgent ⇒
    contractagents' = contractagents \ {con?} ∧
    idlenodes' = idlenodes ∪ {ContractNodeRemoveTask(con?, {t?})}
```

The schemas describing the awarding and terminating of contracts are analogous to the schemas describing the making and breaking of engagements presented in Chapter 5 by the SystemEngage and SystemDisengage schemas.

7.7 Summary and Conclusions

The specification structure used to describe the components and relationships of a system that employs the contract net protocol is shown in Figure 7.4. The components are derived from the entities of the
agency framework, and the relationships are derived from the agency relations model. The state can then be defined by including the set of bids and task announcements which, in turn, are all defined using the agency framework types. The contract net protocol is then defined using four operation schemas on the system state as shown in Figure 7.5.

By applying our existing models to the contract net protocol we have produced a formal specification that serves to make precise both the operation of nodes in the contract net and the state of the contract net at various points during the protocol. The nature of the dependencies between the nodes in the contract net can be readily explicated according to the agency relations model.

Indeed, the contract net protocol provides exemplars of the agent relationships we have called engagements and engagement chains. Even though this is not an aspect of the work in this thesis, it does suggest the possibility of specifying agent systems and using the specification to highlight agent relationships which can then be generalised and integrated into our models.

There have been many attempts at formalising the contract net protocol (e.g. [60, 109, 118, 165, 170]). However, our approach differs markedly: first, we use a well-known generic specification language, which ties in closely with implementation issues and, second, we situate our formalisation in the broader context of a general agent framework. We are not concerned with the development of the formalisation itself, but with the application of the abstract agency framework and the ensuing agency relationship model, to the specification of concrete systems. We have thus shown a clear link between our formal models and an existing implemented multi-agent mechanism by producing a formal specification of the contract net protocol by direct application of these models.

Figure 7.5: Schema Structure for Specifying Contract Net Protocol
Chapter 8

A Computational Architecture for AgentSpeak(L)

8.1 Introduction

While many different and contrasting single-agent architectures have been proposed, perhaps the most successful are those based on the belief-desire-intention (BDI) framework. The examples presented in Chapter 2 touch upon the wealth of research that has accumulated on both the formal and theoretical aspects of BDI agents through the use and development of various logics, for example, on the one hand, and on the practical aspects through the development of implementations of BDI agents on the other. Indeed, so many deliberative agent architectures and systems are based on the BDI framework that it is viewed as being as central to single-agent systems as the contract net is to multi-agent systems.

As we have seen in Chapter 2, however, there is a fragmentation between the theoretical and practical BDI camps. In response, Rao has recently tried to address this concern in two ways. First, he has provided an abstract agent architecture that serves as an idealization of an implemented system and as a means for investigating theoretical properties [133]. A second effort also started with an implemented system and formalised its operational semantics in an agent language. This work resulted in the AgentSpeak(L) language, discussed previously in Section 2.6, which provides an abstraction for implemented BDI systems such as PRS and dMARS.

We choose this example for the following reasons: it is a representative of a large family of systems inspired by the BDI framework that figure so predominantly in agent-based work; it is an attempt to integrate theoretical and practical concerns and; it demonstrates how specific agent architectures may be incorporated into the agency framework. We have already demonstrated part of this latter issue by applying our models to tropistic and hysteretic agents. However, we ought to consider not only small examples such as these, but also those at the leading edge of agent research (exemplified by AgentSpeak(L)) in order to demonstrate the general utility of our work.

In this chapter, we develop a computational architecture that implements this language by application of the generic skeletal architectures of the store-agent model and agency framework. We do not provide a complete specification of a working architecture here, since we only wish to show how the main components of this novel approach can be readily and simply accommodated within the agency framework. A complete specification can be found elsewhere [40].

The organisation of this chapter reflects the organisation of the architecture as follows. The first section provides an overview of the AgentSpeak(L) system and Section 8.3 introduces and defines the types and primitives necessary for the specification of the system, including beliefs, goals, plans and intentions. In Section 8.4, we proceed to the specification of an AgentSpeak(L) agent and its state by building on the definition of a store agent. Section 8.5 provides details of the operation of AgentSpeak(L) agents and their action-reasoning cycle. The chapter ends with a discussion of the contribution of this work in continuing Rao’s original efforts to unify practical and theoretical aspects of BDI-inspired models and systems.
8.2 AgentSpeak(L)

As discussed in Section 2.6, the basic operation of agents in AgentSpeak(L) is constructed around their beliefs, desires and intentions. An agent has beliefs (about itself and the environment), desires (in terms of the states it wants to achieve in response) and intentions as adopted plans. In addition, agents also maintain a repository of available plans, known as the plan library. Agents respond to changes in their goals and beliefs, which result from perception, and which are packaged into data structures called events, representing either new beliefs or new goals. They respond to these changes by selecting plans from the plan library for each change and then instantiating one of these plans as an intention. These intentions comprise plans which, in turn, comprise actions and goals to be achieved, with the latter possibly giving rise to the addition of new plans to that intention.

The operation of an AgentSpeak(L) agent can be summarised as follows.

- If there are more events to process, select one.
- Retrieve from the plan library all the plans triggered by this event that can be executed in the current circumstances.
- Select for execution one of the plans thus generated, and generate an instance of that plan, known as the intended means.
- Add the intended means to the appropriate intention. This will be a new intention in the case of an external event arising through changes to beliefs, and an existing intention in the case of an internal event resulting from attempting to satisfy a goal in that intention.
- Select an intention and consider the next step in its top plan. If this is an action, perform it, and if it is a goal, add a corresponding event to the set of events yet to be processed.
- If the top plan is complete, consider the next plan, and if the intention is empty, the intention has succeeded and can be removed from the set of intentions.

This description captures the essential operation of AgentSpeak(L) agents. More detail will be added to the description, however, when the operation cycle is elaborated further with the formal specification that follows.

8.3 Types

Designing an AgentSpeak(L) agent consists of building its plan library. This agent is initialised with a set of beliefs and events and subsequently these events are processed during run-time by manipulating a set of intentions. Therefore, the store of an AgentSpeak(L) agent comprises plans (which are fixed), beliefs, events and intentions (which are all are in flux). As when defining the knowledge-level agents in Section 6.2, we provide a semantics for the agency framework type attributes. Specifically, an attribute is either an AgentSpeak(L) plan, intention, event or belief.

\[
\text{Attribute ::= plan}\langle\text{ASPlan}\rangle \\
\quad | \quad \text{intention}\langle\text{ASIntention}\rangle \\
\quad | \quad \text{event}\langle\text{ASEvent}\rangle \\
\quad | \quad \text{belief}\langle\text{ASBelief}\rangle
\]

Next, we specify each of these four different attribute types.
8.3.1 Beliefs

Beliefs are either belief literals or conjunctions of two beliefs.

\[ \text{ASBelief} ::= \text{ASLiteral} \mid \text{ASLiteral} \land \text{ASBelief} \]

A belief literal is either an atom or the negation of an atom, where atoms are defined in Section 6.2.

\[ \text{ASLiteral} ::= \text{pos} \langle \text{Atom} \rangle \mid \text{not} \langle \text{Atom} \rangle \]

For example, if the type, \( \text{Atom} \), contained only the elements \( P(X, Y) \) and \( Q(Y) \), then the type, \( \text{ASLiteral} \), would be equal to the following set.

\[ \{ \text{not} P(X, Y), \text{pos} P(X, Y), \text{not} Q(Y), \text{pos} Q(Y) \} \]

8.3.2 Events

Events are the addition or deletion of a trigger, which is either a belief or a goal, and may also be associated with an intention. In AgentSpeak(L), events result either from an external source, in which case they are just external triggering events unrelated to an intention, or from the execution of a current intention, in which case they are (subgoal) triggering events with an explicit connection to an intention. Thus, we define the Event type as follows, where int represents an optional intention. (The definitions of optional and related concepts, defined, undefined and the, can be found in Section 5.2).

\[
\text{ASEvent} = \text{ASTriggerEvent} \\
\text{int} : \text{optional} [\text{ASIntention}]
\]

\[ \text{ASTrigger} ::= \text{bel} \langle \text{ASBelief} \rangle \mid \text{goal} \langle \text{ASGoal} \rangle \]

\[ \text{TriggerSymbol} ::= \text{add} \mid \text{remove} \]

\[ \text{ASTriggerEvent} ::= \text{TriggerSymbol} \times \text{ASTrigger} \]

As stated above, in an external event, the intention is not defined, while in an internal event, it is defined.

\[
\text{ExternalEvent} = \text{ASEvent} \\
\text{undefined} \text{ int}
\]

\[
\text{InternalEvent} = \text{ASEvent} \\
\text{defined} \text{ int}
\]

Goals are defined as either achieving an atom (making some predicate true) or querying an atom (testing whether some predicate is true).

\[ \text{ASGoal} ::= \text{achieve} \langle \text{Atom} \rangle \mid \text{query} \langle \text{Atom} \rangle \]

We delay the specification of intentions until after we have specified AgentSpeak(L) plans considered next.
8.3.3 Plans

Plans in AgentSpeak(L) comprise three components. The *invocation condition* details the circumstances, in terms of beliefs or goals, that have caused a plan to be triggered. Similarly, a *context* specifies the beliefs of the agent that must hold for the plan to be selected for execution. Finally, the part of the plan that specifies the sequence of *formulae* the agent needs to perform is known as the plan *body*. This determines what the agent must *do*.

\[
\begin{align*}
\text{ASPlan} & : = \text{ASTriggerEvent} \\
\text{inv} & : \text{ASTriggerEvent} \\
\text{context} & : \text{P ASBelief} \\
\text{body} & : \text{seq ASFormula}
\end{align*}
\]

A formula is either an *action* to be executed, an ‘achieve goal’ to be satisfied, or a test goal to be answered.

\[
\text{ASFormula} ::= \text{actionformula} \langle \text{ASAction} \rangle | \text{goalformula} \langle \text{ASGoal} \rangle
\]

Actions are represented by an *action symbol* and a sequence of terms.

\[
\text{ASAction} \\
\text{name} : \text{ActionSym} \\
\text{terms} : \text{seq Term}
\]

Consider a very simple example of a plan, given below, which is triggered whenever a compulsive robot car-theif finds itself next to a car. In order to select this plan, the robot must believe that the car is currently empty. The plan body, which details how to gain control of the car, consists of a sequence of formulae. In this case, the robot must first perform the primitive action to move to the car, then the primitive action to get in the car, and then achieve the subgoal to start the car. The actual achievement of this subgoal may require further plans.

\[
\begin{align*}
\text{inv} & = (+, \text{belief adjacent(Robot, Car)}) \\
\text{context} & = \text{literal rule pos empty(Car)} \\
\text{body} & = \text{actionformula} \text{MoveTo(Car)}, \text{actionformula} \text{GetIn(Car)}, \\
& \quad \text{goalformula} \text{achieve Start(Car)}
\end{align*}
\]

8.3.4 Intentions

Intentions are simply non-empty sequences of plans which need to be executed in order.

\[
\text{ASIntention} ::= \text{seq} \text{ASPlan}
\]

8.4 AgentSpeak(L) Agents

Now that we have formally described the mental components and data structures of AgentSpeak(L) agents we are in a position to specify how they can be incorporated to specify the agents themselves. This can be achieved by refining our schema definition of a store-agent as defined in Section 6.2. Before run-time an AgentSpeak(L) agent is defined by its plan library which comprises the entire store.
The state of an agent at run-time includes the agent’s beliefs, intentions and events as well as its plan library. In addition, the set of attributes the agent can perceive from its internal store is defined by the current value of the variable events. If events is non-empty it chooses an event to process as determined by the function eventselect, specified below. The function storecanperceive is thus the identity function since it can perceive all its events, and the function willperceive, which is independent of the agent’s goals, is simply equal to the function eventselect. According to the agency framework, the possible percepts of an AgentSpeak(L) agent at any time are simply the set of events in its store. (The posspercepts variable is defined in the AgentState schema in Section 3.6).

In addition, though this is not mentioned in Rao’s original description, it is necessary to record the status of each current intention, in order to ensure that when intentions become suspended they are not selected for execution.

\[
\text{eventselect} : \mathbb{P}_1 \text{ASEvent} \rightarrow \text{ASEvent}
\]

\[
\text{Status} ::= \text{active} \mid \text{suspended}
\]

The beliefs and events are supplied when the agent is initialised, at which time the set of intentions is empty.

Now that the AgentSpeak(L) agent has been defined, as well as its general and initial state, we can specify its actual operation.
8.5 AgentSpeak(L) Agent Operation

At any time an agent may receive a new external event that is added to the store. Agents have no choice about the events they receive from the external environment. We model this by defining the agency framework function `extcanperceive` (which we have so far left unspecified for the AgentSpeak(L) agent) as the identity function.

\[\text{NewExternalEvent} \quad \text{attribute}\_? : \text{Attribute} \quad \triangle \text{ASAgentState} \]

\[
\begin{align*}
\text{attribute}\_? &\in (\text{ran event}) \\
\text{store}' &= \text{store} \cup \{\text{attribute}\_?\} \\
\forall as : \mathbb{P} \text{Attribute} &\mid as \subseteq (\text{ran event}) \cdot \\
\text{extcanperceive as externalperceivingactions} &= as
\end{align*}
\]

Whilst the perception of an AgentSpeak(L) is straightforward, the mechanism by which it selects its actions is not. As discussed in Section 2.6, an AgentSpeak(L) agent is either processing an event or it is executing an intention. The former is illustrated in Figure 8.1 and is described below.

1. The agent selects an event (e).

2. The agent generates all the plans whose invocation condition matches this event. These plans are known as the relevant plans (\{p(1), p(2), p(3)\}) and are generated using the function `genrelplans` which is applied to the selected event and the plan library. Only the signature of this function is presented here.

\[\text{genrelplans} : \text{ASEvent} \rightarrow \mathbb{P} \text{ASPlan} \rightarrow \mathbb{P} \text{ASPlan}\]

3. From these relevant plans, the agent identifies those with pre-conditions that are currently satisfied with respect to the agent’s beliefs. These plans are called applicable plans (\{p(2), p(3)\})

---

1Defining this ‘matching’ requires a detailed specification of the standard aspects of unification. This is not provided here since it is unnecessary for our treatment of agent architecture in this chapter. Details can be found in [65], pages 66-67.
and are generated using the function \textit{genapplplans} which is applied to the relevant plans and the current set of beliefs.

\[
\text{\textit{genapplplans}} : \mathbb{P} \text{ASPlan} \to \mathbb{P} \text{ASBelief} \to \mathbb{P} \text{ASPlan}
\]

4. If there are several plans that are applicable, one is chosen nondeterministically (by applying the function \textit{planselect} specified below). The set of bindings used to unify the trigger and context with the selected event and current beliefs is called a \textit{unifier} and is found by applying the function \textit{unify} to the selected plan, beliefs and selected event. This unifier is applied to the selected plan using the function \textit{ApplySubPlan}. The resulting plan (the intended means), is therefore a partially instantiated copy of the plan chosen from the plan library. The agent’s intentions are then updated in the following way: if the selected event is external, a new intention is generated with an active status; if the selected event is internal, the plan is added to the head of the intention that posted it, and the status of this intention is reset to active.

\[
\text{\textit{planselect}} : \mathbb{P}_1 \text{ASPlan} \to \text{ASPlan} \\
\text{\textit{ApplySubPlan}} : \text{Substitution} \to \text{ASPlan} \to \text{ASPlan} \\
\text{\textit{unify}} : \text{ASPlan} \to \mathbb{P} \text{ASBelief} \to \text{ASEvent} \to \text{Substitution}
\]

The schema below specifies an AgentSpeak(L) agent processing an event (\textit{selectedevent}), generating the relevant plans (\textit{relevantplans}), then the applicable plans (\textit{applicableplans}), selecting one of these (\textit{selectedplan}), and applying the unifier (\textit{applicableunifier}) that was used to match this plan with the selected event and current beliefs to generate the intended means (\textit{intendedmeans}). The way in which the intentions are updated is then dependent on whether the selected event is internal or external as described above. The pre-condition simply states that the operation can only occur when there are events to process.

\[
\text{\textit{ProcessEvent}} \\
\text{\textit{\triangle ASAgentState}}
\]

\[
\begin{align*}
\text{\textit{events}} & \neq \emptyset \\
\text{let } \text{\textit{selectedevent}} & \equiv \text{\textit{eventselect}} \text{\textit{events}} \quad \bullet \\
\text{let } \text{\textit{relevantplans}} & \equiv \text{\textit{genapplplans}} \text{\textit{selectedevent}} \text{\textit{planlibrary}} \quad \bullet \\
\text{let } \text{\textit{applicableplans}} & \equiv \text{\textit{genapplplans}} \text{\textit{relevantplans}} \text{\textit{beliefs}} \quad \bullet \\
\text{let } \text{\textit{selectedplan}} & \equiv \text{\textit{planselect}} \text{\textit{applicableplans}} \quad \bullet \\
\text{let } \text{\textit{applicableunifier}} & \equiv \text{\textit{unify}} \text{\textit{selectedplan}} \text{\textit{selectedevent}} \text{\textit{beliefs}} \quad \bullet \\
\text{let } \text{\textit{intendedmeans}} & \equiv \text{\textit{ApplySubPlan}} \text{\textit{applicableunifier}} \text{\textit{selectedplan}} \quad \bullet \\
\text{\textit{selectedevent}} & \in \text{\textit{ExternalEvent}} \Rightarrow \\
& (\text{\textit{intentions}' = } \text{\textit{intentions}} \cup \{\{\text{\textit{intendedmeans}}\}\} \land \text{\textit{status}' = } \text{\textit{status}} \cup \{((\text{\textit{intendedmeans}}), \text{active})\}\} \land \\
\text{\textit{selectedevent}} & \in \text{\textit{InternalEvent}} \Rightarrow \\
& (\text{\textit{let } \text{\textit{oldintention}} = (\text{\textit{the } selectedevent.int})} \quad \bullet \\
& \text{\textit{let } \text{\textit{newintention}} = (\text{\textit{intendedmeans}}) \smallsetminus \text{\textit{oldintention}} \quad \bullet \\
& (\text{\textit{intentions}' = } \{\text{\textit{intentions}} \smallsetminus \{\text{\textit{oldintention}}\}\} \cup \{\text{\textit{newintention}}\} \land \text{\textit{status}' = } (((\text{\textit{oldintention}} \smallsetminus \text{\textit{status}}) \cup \{((\text{\textit{newintention}}, \text{active})\}\}))))
\end{align*}
\]

The agent’s other mode of operation is the execution of intentions. During this phase the agent selects an intention (\textit{selectedintention}) and attempts to execute the next formula (\textit{executingformula}) in the topmost plan (\textit{executingplan}). There are three cases depending on whether the formula is an achieve goal, a query goal or an action. First, if the formula is an achieve goal, it is assumed that
it cannot immediately be achieved, and a goal event (sometimes called an internal event) is created. This event is added to the set of events in the store for future processing. It alerts the agent to finding a plan to achieve the goal so that the execution of the current intention stack can continue.

This schema includes a reference to the auxiliary function \textit{MakeEvent}, which simply constructs an element of type \textit{Event} from its constituent components.

\begin{verbatim}
PostAchieveGoal
\triangle ASAgentState
\begin{align*}
& \text{executingformula } \in \text{ran goalformula} \\
& (\text{goalformula}\^{-1}\text{executingformula}) \in \text{ran achieve} \\
& \text{let achievegoal } == \text{goalformula}\^{-1}\text{executingformula} \\
& \text{events'} = \text{events} \cup \{\text{MakeEvent}((\text{add, goal achievegoal}), \{\text{selectedintention}\})\}
\end{align*}
\end{verbatim}

In this case the intention becomes suspended so that it cannot be chosen for execution until the newly-posted subgoal has been achieved.

\begin{verbatim}
SuspendIntention
\triangle ASAgentState
\begin{align*}
& \text{status'} = \text{status} \oplus \{(\text{selectedintention}, \text{suspended})\}
\end{align*}
\end{verbatim}

Second, if the formula is a query goal that can be unified (matched) with the set of current beliefs, then the substitution that achieves this unification is applied to the rest of the executing plan. In the following schema, \textit{querygoal} represents the query goal, \textit{mgu} the substitution, and the substitution that unifies the query goal with the beliefs is found by applying \textit{mguquery} to \textit{querygoal} and \textit{beliefs}. The \textit{executingplan'} variable, which represents the state of the executing plan after the operation, is the result of applying the bindings in \textit{mgu} to the variables in \textit{executingplan}.

\begin{verbatim}
AchieveQueryGoal
\triangle ASAgentState
\begin{align*}
& \text{executingformula } \in \text{ran goalformula} \\
& \text{goalformula}\^{-1}\text{executingformula} \in \text{ran query} \\
& \text{let querygoal } == \text{goalformula}\^{-1}\text{executingformula} \\
& \text{let mgu } == \text{mguquery(querygoal, beliefs)} \\
& \text{executingplan'} = \text{ApplySubPlan mgu executingplan}
\end{align*}
\end{verbatim}

Third, the formula is an action in which case it is posted for future performance to a buffer called \textit{actionbuffer} as described in the schema below.

\begin{verbatim}
PostAction
\triangle ASAgentState
\begin{align*}
& \text{actionbuffer, actionbuffer'} : \mathbb{P} \text{ Action} \\
& \text{executingformula } \in \text{ran actionformula} \\
& \text{actionbuffer'} = \text{actionbuffer} \cup \{\text{actionformula}\^{-1}\text{executingformula}\}
\end{align*}
\end{verbatim}

In the last two of these cases, the executing formula is removed from the executing plan. If there is then no next formula in the executing plan, but a next plan in the selected intention, the unifier of the invocation condition of the \textit{second plan} on the stack and the invocation of the \textit{executing plan} is found,
and this substitution is applied to the second plan in the stack. If there is no next formula and no next plan, then the intention has succeeded since it is empty, and can be removed from the set of intentions. (Note that this latter possibility is not addressed in Rao’s original operational semantics [132], but is an important case that demands explicit consideration. Though this case and the previous one are not specified here a formal treatment of both can be found elsewhere [40].)

In addition, according to the agency framework, there are aspects of specifying the general operation of an agent that have not been considered by AgentSpeak(L). First, the language does not specify the performance of the actions contained in the agent’s buffer and second, it does not detail how actions change the state of the environment (defined by the agency framework functions agentactions and effectinteraction). We are thus able to identify omissions in the scope of the language which need to be addressed before it can provide a comprehensive definition of the operation of an agent in practice.

8.6 Summary and Conclusions

In this chapter we have shown how we can use the set of generic templates from the store-agent model to formalise the specific architecture of AgentSpeak(L) agents. First, the representation of attributes required to model the mental and data components of the store of an AgentSpeak(L) agent are defined. These artifacts are then incorporated to define the architecture of the AgentSpeak(L) agent by refining the schemas of the agency framework store-agent, which specifies perception and action capabilities. Finally, after the agent’s state at run-time is described, the operation of the agent is specified.

This reformalization has revealed a number of errors and omissions in the original formulation, including some relating to the specification of aspects of binding and plans, an omission of one possible case in the agent operation cycle, and the incorrect assertion that an intention stack can only be executed if the event queue is non-empty, which is inappropriate. In developing the specification, we add to the original work of Rao, and progress beyond the description of a particular language, by giving a formal specification of a general belief-desire-intention architecture that can be used as the basis for providing such formal specifications of more sophisticated systems. By using the standard Z specification language, we also tackle the problem from a software engineering perspective and make the specification accessible and amenable to implementation by providing a clean and explicit representation of the state and operations on state (including identifying data structures required for operation) that must underlie any implementation. This specification provides a strong foundation from which to implement systems, validate properties and animate operations by using widely available tools such as CADiZ [163], Z/EVES [140] and PiZA [77]. Thus, we have provided a simple demonstration of how our models can be refined to build a computational architecture for recent and important (in the sense that it has created significant interest) research.
Chapter 9

Evaluating Social Dependence Networks

9.1 Introduction

Social dependence networks [147] are structures that form the basis of a computational model of social power theory as originally proposed by Castelfranchi [13]. Essentially, they are a taxonomy of social relationships that can be derived from the ‘power’ agents have over one another as a result of their ability to achieve each other’s goals. Based on this taxonomy, Sichman et al. develop social reasoning mechanisms whereby agents can reason about the dependencies between agents.

In this chapter we re-formulate this work and do so for a number of reasons. First, whilst there are implementations, industrial applications and different types of analysis of the lower-level contract net mechanism and the higher, mentalistic-level BDI model, social dependence networks constitute an ambitious new theoretical model and, whilst a simple simulation of the model exists [146], it has not been applied or evaluated in the same way. Since we claim the agency framework is generally applicable to new theoretical developments as well as to established systems, social dependence networks are a suitable counterpoint to the contract net protocol and AgentSpeak(L) examples. Second, this model is of particular relevance to this thesis since it provides a taxonomy of inter-agent relationships similar in spirit to our own. It is therefore appropriate to analyse the model in terms of our work.

Before reformulating social dependence networks in this way, it is first necessary to provide a detailed account of the model. Subsequently, the process of reformulation will reveal a number of ambiguities and inconsistencies in the model and presentation of social dependence networks, a problem that did not arise, for example, when applying our work of the well-established contract net protocol. Consequently, this chapter often appears to be a critique. This should not indicate that the work does not make a significant contribution to understanding multi-agent systems, but only that there are problems, ambiguities and contradictions in its conception. Indeed, the choice of social dependence networks in this chapter is a measure of the importance we attach to it.

We aim to show that social dependence networks can be accommodated and reformulated within our work, and how this process forces implicit assumptions and ambiguous definitions made in the original description to be isolated. Indeed, many of these problems arise because the work is not built from a well-defined platform and vocabulary set such as that provided by the agency framework, and by appealing to our work many of these problems can be addressed.

To this end, Section 9.2 first provides an introduction to social dependence networks. The next five sections introduce different aspects of this model. Each of these sections discusses issues that arise from its original formulation and then reformulates it by application of the agency framework, agency relations model and sociological agency model.

9.2 Social Dependence Networks

Social power theory (SPT) is relevant to multi-agent systems because it attempts to provide an explanation of why autonomous agents adopt the goals of others. The theory is proposed by Castelfranchi,
partly as a reaction against the assumption of benevolence in the design of agents in distributed agent architectures [11, 15], which requires that agents always adopt the goals of others whenever they are able to do so. Castelfranchi states that benevolence severely limits the behavioural possibilities of agents and argues for accounts of more autonomous goal adoption. SPT is proposed as a mechanism for determining why non-benevolent agents adopt each other’s goals.

The theory is based on notions of dependence and reciprocation, which arise because agents have limited capabilities and resources and may therefore depend on others to achieve their goals. If an agent depends on another for one of its goals, then the latter agent has power over the former. This power provides an explanation of why the former agent may autonomously adopt the goals of the latter. For example, suppose that agent $A$ has the goal $g_A$, which requires the action $a$ to achieve it. If $a$ is not in the capabilities of $A$, but in the capabilities of another agent $B$, then $B$ is said to have power over $A$. In this situation, $B$ may be able to persuade $A$ to adopt another goal because of this power and the autonomous agents can reciprocate. Sichman et al. subsequently developed a taxonomy of inter-agent relationships known as social dependence networks (SDNs) [147, 146] based on SPT.

The following premises are made before the taxonomy is developed.

- Each agent has goals, actions, resources and plans.
- Goals are achieved by plans.
- Goals are associated with a set of plans to achieve that goal.
- Plans are sequences of actions.
- Each action in a plan may require resources.
- Each plan may require resources.
- Some resources are owned by agents.
- Agents maintain a model, called an external description, of the goals, actions, resources and plans of all system agents. An example of an external description can be found in Table 9.1. In this table, Jamie has four goals ($g_1$, $g_2$, $g_3$ and $g_4$), is able to perform one action ($a_1$), owns one resource ($r_1$), and has a plan for achieving each of the four goals. The plan for achieving $g_1$, for example, consists of performing the action $a_3$ using resource $r_1$. However, since Cristiano is the only agent capable of performing $a_1$, in the situation where Jamie wishes to achieve $g_1$, Cristiano is said to have “social power” over Jamie.
subsequent four sections consider each of the categories above in turn.

Action and Resource Autonomy. An agent can establish its autonomy with respect to a goal and presentation of the SDN model falls readily into four aspects discussed below.

Dependence Situations. If agents are not autonomous with respect to a goal according to a set of plans to achieve a goal. All subsequent inter-agent relationships in the SDN model are defined with respect to the external descriptions of agents who analyse their situation with respect to the set of plans of an agent to achieve a goal. These plans can either belong to the reasoning agent or to any other agent in the system. The presentation of the SDN model falls readily into four aspects discussed below.

Dependence Relations. If agents are not autonomous with respect to a goal according to a set of plans as described above, they will depend on other agents for actions or resources in order to achieve it. Such situations are modelled using dependence relations. Agents either a-depend, r-depend or s-depend according to whether the dependence is for actions, resources or both.

Locally and Mutually Believed Dependence. If the dependence situations between A and B are independent of whether they are determined using either A’s or B’s plans, then the dependence is said to be mutually believed. If the dependence situation between A and B only exists according to A’s plans, and not according to B’s plans, then A is said to locally believe the dependence situation.

In the next section we evaluate and reformulate the SDN model of external descriptions, and the subsequent four sections consider each of the categories above in turn.

9.3 External Descriptions

The original formalisation of external descriptions can be found in Table 9.2. An agent, \( ag_i \), has a set of external descriptions of the other agents and itself as formalised in definition 9.1. Each external description that agent \( ag_i \) has of \( ag_j \) is denoted by \( Ext_{ag_i}(ag_j) \), formalised in definition 9.2, and

\[
Ext_{ag_i} \equiv \bigcup_{j=1}^{n} Ext_{ag_i}(ag_j) \quad (9.1)
\]

\[
Ext_{ag_i}(ag_j) \equiv \{ G_{ag_i}(ag_j), A_{ag_i}(ag_j), R_{ag_i}(ag_j), P_{ag_i}(ag_j) \} \quad (9.2)
\]

\[
p_{ag_i}(ag_j, g_k) \equiv \{ g_k, R(p_{ag_i}(ag_j, g_k)), I(p_{ag_i}(ag_j, g_k)) \} \quad (9.3)
\]

\[
i_m(p_{ag_i}(ag_j, g_k)) \equiv \{ a_m, R_{am}(p_{ag_i}(ag_j, g_k)) \} \quad (9.4)
\]

Table 9.2: Original Definition of External Descriptions

- Any two agents have the same external description of any other agent including themselves.
consists of the goals, actions, resources and plans that $ag_i$ believes that $ag_j$ has. Goals, actions and resources are defined textually by Sichman et al. as those an agent “wants to achieve”, those an agent is “able to perform” and those over which an agent has “control”. Plans, which are formalised in terms of these notions, are described as those that “an agent has using any actions and resources to achieve a goal”. The expression $P_{ag_i}(ag_j, g_i)$ represents the set of plans that agent $ag_i$ believes that agent $ag_j$ has in order to achieve the goal $g_i$. Each plan within this set is denoted by $p_{ag_{ij}}$ and defined in 9.3 where $R(p_{ag_{ij}})$ represents the set of resources required for the plan, and the expression $I(p_{ag_{ij}})$ represents a sequence of instantiated actions used in this plan. Each instantiated action within a plan is defined in 9.4 by the action itself and the set of resources used in the instantiation of this action.

The authors adopt what is referred to as the hypothesis of external description compatibility, which states that any two agents agree on their external descriptions. The authors introduce this simplification since they are “interested in analysing the impacts of such a mechanism on the internal structure of an agent”. This hypothesis states that for any two agents $i$ and $j$, $i$’s external description entry for $i$ is the same as $j$’s external description for $i$ and $i$’s external description for $j$ is the same as $j$’s external description for $j$. This is formalised below.

$$E_{xt_{ag_i}}(ag_i) = E_{xt_{ag_j}}(ag_i) \land E_{xt_{ag_j}}(ag_j) = E_{xt_{ag_i}}(ag_j) \quad (9.5)$$

There are several issues concerned with external descriptions that arise directly from comparison with the agency framework. This enables implicit assumptions and ambiguous premises to be isolated.

**Resources versus Agents.** In the agency framework no distinction is made between agents and resources, and instead we consider only agents, but with different functionalities. As a result, plan-agents need only to consider the set of agents required in a plan. Plans can therefore be represented as sets of actions, where each action is associated with a set of agents, each one potentially capable of performing the action. Crucially, the distinction between a resource and an agent in SDN is not clear; the resources of an agent are defined simply as “the resources an agent has control on”. Presumably, an arbitrary distinction between agents and resources must be made for SDN to be applied. The distinction is important because the nature of a plan assumes that whilst all the resources required by an action have already been identified, the agents that might perform that action have not. In this respect, incomplete plans, where the resources required have not yet been considered, cannot be represented. Equally, a plan where the agent who is to perform an action has been established cannot be represented. Thus the formalism is inflexible in this respect, and too critically influenced by whatever distinction is chosen to differentiate between resources and agents. By allowing agents with different functionalities, the restriction on SDN plans that requires that the resources needed for each action have been selected, but the agents have not, can be removed.

**Simultaneous and Concurrent Actions.** According to our model of plans and plan-agents in Section 6.4, agents may plan to perform concurrent actions and, further, plans can comprise several actions that need to be performed simultaneously by different agents. These are included because they are important requirements when describing multi-agent systems where cooperation is likely to ensue. However, neither simultaneous nor concurrent action is represented in SDN.

**Ownership.** The definition of ownership in SDN is vague and ambiguous, which detracts from any subsequent definitions. This contrasts to the definition provided by the agency relations model where an agent owns another entity if it is not engaged by any other agent and can be used as required without regard to others. Further, sociological agents are able to understand and reason about such relationships, which enables them to determine the interaction possibilities available. An agent believes it owns an entity if, according to its models, no other agent is engaging that entity for another purpose, indicating that the entity can be used by the agent without regard to others. A similar analysis of the key inter-agent relationships in multi-agent systems has not been undertaken in the development of SDN. Consequently, SDN agents are not able to recognise the relationships of the agency relationship model. In addition, there is a strong indication of an underlying simplification that resources cannot be shared. This is clearly restrictive since, in general, resources can be shared. We address this issue at the end of this section.
Agent Perspective. In our view, agents can easily have different and conflicting views of their environment and can never know about the plans, goals and capabilities of others. In general, autonomous agents have subjective models. However, the hypothesis of external description compatibility used in the development of SDNs ensures that any two agents will agree on their external description entry for a given agent. Though the authors argue that there is no loss of generality, it is difficult to see how this can be so, since autonomous agents may have incomplete and conflicting models of each other.

In other work [146], the authors do not use assumption but instead make the following assertion.

“For simplicity, let us consider that the plans of the agents are the same and both of them know the plans of the other.”

Consequently, there can be agent-level inconsistency in terms of what agents believe about the capabilities, resources and goals of each other, assuming they know the plans of every agent. However, this is still a limiting assumption that is untenable for general interacting autonomous agents.

We now state how we model SDNs within the sociological agency model, making any assumptions about SDN explicit. Without any loss of generality we take actions and goals in the SDN model to be actions and goals in the agency framework. As the meaning of a resource is not clear, and in order that our re-formulation makes no limiting assumptions, we take a resource to be an entity, the most general type possible. We therefore take the owned resources of an agent to be a set of entities. The relationship of SDN plans to sociological agency model plans is more complicated and is discussed below.

1. All SDN plans are what we have defined to be total plans (see Section 6.4).

2. In our definition of plans, composite-actions are either primitive actions, concurrent actions, template actions, or concurrent template actions (see Section 6.4). In SDN however, plans contain only primitive actions. According to our model of plans, the associated entities of a primitive action must have the necessary capabilities to perform that action. This is clearly an underlying assumption of the SDN model that has not been made explicit.

3. Plans in the sociological agency model consist of plan-actions, which are sets of ordered pairs. The first element of each ordered pair is a compound-action, and the second is the set of entities who may perform that action. Sets of pairs are used in order to represent simultaneous actions. In the SDN model, no two entities can simultaneously perform actions so that plan-actions contain only one pair containing a primitive action, as we have seen. Actions in SDN plans are associated with a set of entities used to perform that action. Since, in our existing formalisation, an action in a plan is also associated with a set of entities, our representation of a plan can be directly applied.

4. Goals in the SDN model are associated with sets of plans to achieve that goal, where each plan is considered as a potential means of achieving a goal. In our definition of a plan-agent, this association is represented by the functions, activeplangoal or plangoallibrary, in the PlanAgent schema depending on whether the goal is a current one.

5. The resources required by an action in a plan as well as the resources required by a plan are represented in the SDN model. However, if the resources required by each action within a plan are known, then so must the set of all the resources required by the plan. Specifically, if \( R(p_{a,q}(ag_1,g_k)) \) is known for each action, then \( R(p_{a,q}) \) is also known. Modelling the resources of a plan is therefore redundant and need not be included in our re-formulation.

According to the discussion above, the schema formalising external descriptions is a refinement of a plan-agent as defined in the PlanAgent schema in Section 6.4. The ExternalDescription schema that defines an external description therefore includes this schema. It is only necessary to include one
additional variable, \textit{ownedresources}, to represent the set of resources owned by an agent. The predicates in the schema also make \textit{explicit}, assumptions that are \textit{implicit} in the SDN model as discussed above, which assert that plan-actions are sets containing only one pair, any action in a plan-action is primitive, and third, all plans are total.

\begin{verbatim}
  \textbf{ExternalDescription}
  \textbf{PlanAgent}
  \texttt{ownedresources : \forall Entity}

  \forall \texttt{pa : PlanAction} . \forall \texttt{p : Plan} | 
  \texttt{p \in planlibrary} \land \texttt{pa \subseteq \bigcup\{planpairs p\}} \land 
  \#\texttt{pa} = 1 \land 
  \texttt{dom pa \subseteq ran Prim} \land 
  \texttt{p \in ran Total}
\end{verbatim}

Each SDN agent has an external description of every other agent. However, the authors introduce what they call the \textit{hypothesis of external reality}, which states that any two agents have the same external description for each agent. The set of external descriptions in a multi-agent system can therefore be modelled simply by associating an external description with every agent. The external description associated with an agent, \texttt{A}, is then the model that every agent (including \texttt{A}) has of \texttt{A}.

\begin{verbatim}
\textbf{MultiAgentSystemExternalDescriptions}
\textbf{MultiAgentSystem}
\texttt{extdes : Agent \to ExternalDescription}
\texttt{dom extdes = agents}
\end{verbatim}

Then, according to the external description of some agent, \texttt{A},
- \texttt{(extdes A).planlibrary} is its set of library plans,
- \texttt{(extdes A).activeplan} is its set of active plans,
- \texttt{(extdes A).capabilities} is its set of actions,
- \texttt{(extdes A).ownedresources} is its set of resources,
- \texttt{(extdes A).goals} is its set of current goals, and
- \texttt{(extdes A).goallibrary} is its set of library goals.

If resources cannot be shared, there is a further restriction on the state that can be expressed explicitly using the vocabulary of the agency relations model. Specifically, if an agent \texttt{directly engages} an entity then it must \texttt{directly own} that entity. This is a restriction placed on the environment itself, not on the way in which agents model the environment and can be formalised as shown below by using components from the \textit{AgencyRelationsTaxonomy} schema, which is derived from the \textit{AgencySociety} schema defined in Section 4.7.

\begin{verbatim}
\forall \texttt{system : AgencyRelationsTaxonomy; A : Agent; S : ServerAgent} \bullet 
\texttt{(A, S) \in system.dengages \Rightarrow (A, S) \in system.downs}
\end{verbatim}

### 9.4 Action and Resource Autonomy

Using external descriptions, Sichman et al. distinguish three distinct categories of autonomy referred to as \texttt{s–autonomous}, \texttt{r–autonomous} and \texttt{s–autonomous}. According to these definitions agents are autonomous if they have the necessary capabilities and resources to achieve a goal and so do not need the help of others.
An agent $ag_i$ will be $a-$autonomous for a given goal $g_k$, according to a set of plans $P_{qk}$ if there is a plan that achieves this goal in this set and every action appearing in this plan belongs to $A(\text{ag}_i)$:

$$a_{aut}(ag_i, g_k, P_{qk}) \iff \exists g_k \in G(\text{ag}_i) \exists p_{ik} \in P_{qk} \forall i_m \in (p_{ik}) \exists r_m \in (p_{ik}) a_m \in A(\text{ag}_i)\quad(9.6)$$

An agent $ag_i$ will be $r-$autonomous for a given goal $g_k$, according to a set of plans $P_{qk}$ if there is a plan that achieves this goal in this set and every resource appearing in this plan belongs to $R(\text{ag}_i)$:

$$r_{aut}(ag_i, g_k, P_{qk}) \iff \exists g_k \in G(\text{ag}_i) \exists p_{ik} \in P_{qk} \forall r_m \in R(p_{ik}) r_m \in R(\text{ag}_i)\quad(9.7)$$

An agent $ag_i$ will be $s-$autonomous for a given goal $g_k$, according to a set of plans $P_{qk}$ if he is both $a-$autonomous and $r-$autonomous for this goal:

$$s_{aut}(ag_i, g_k, P_{qk}) \iff a_{aut}(ag_i, g_k, P_{qk}) \land r_{aut}(ag_i, g_k, P_{qk})\quad(9.8)$$

Table 9.3: Original Definition of Action and Resource Autonomy

<table>
<thead>
<tr>
<th>Definition</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_{aut}$</td>
<td>$\exists g_k \in G(\text{ag}<em>i) \exists p</em>{ik} \in P_{qk} \forall i_m \in (p_{ik}) \exists r_m \in (p_{ik}) a_m \in A(\text{ag}_i)$</td>
</tr>
<tr>
<td>$r_{aut}$</td>
<td>$\exists g_k \in G(\text{ag}<em>i) \exists p</em>{ik} \in P_{qk} \forall r_m \in R(p_{ik}) r_m \in R(\text{ag}_i)$</td>
</tr>
<tr>
<td>$s_{aut}$</td>
<td>$a_{aut}(ag_i, g_k, P_{qk}) \land r_{aut}(ag_i, g_k, P_{qk})$</td>
</tr>
</tbody>
</table>

Table 9.4: Original Formalisation of Action and Resource Autonomy

The original definitions and their formal representations as presented by Sichman at al. can be found in Table 9.3 and Table 9.4, respectively. However, there are a number of difficulties with them. First, the textual definitions are slightly deceptive since $P_{qk}$ is an abbreviation of $P_{ag_i}(ag_i, g_k)$ and represents a very specific set of plans rather than any set of plans. It represents the set of plans that $ag_i$ believes that agent $ag_i$ has to achieve the goal $g_k$. Further, since every plan in this set necessarily achieves $g_k$, the textual definition includes unnecessary redundancy. All that is necessary is that $P_{qk}$ is non-empty.

The definition of the category $s-$autonomous is also deceptive. An agent is in this category if, within the set of plans being analysed, there is one plan that contains actions within the agent’s capabilities, and another plan that involves resources all owned by the agent. However, there is no stipulation that these plans are the same. In other words, an agent can be $s-$autonomous for a goal, and still not be able to achieve it, since there may be no specific plan that requires just the capabilities and resources of that agent. In this situation, all plans would then involve either the resources or the capabilities of other agents, so it makes little sense to say that the agent is autonomous with respect to this goal when it necessarily depends on others.

In addition, it is not clear in these definitions whether agents can reason with respect to their current goals or to any goal in their goal library. A goal of an agent is simply described as something “an agent wishes to achieve”. However, if we are guided by the examples provided by the authors such as the one that can be found in Figure 9.1, it does appear that goals are something an agent currently wishes to achieve. The distinction is important, since it has ramifications for subsequent SDN categorisations as will be discussed.

To address these concerns we first provide slightly altered textual definitions, which relate more strongly to the original SDN formalisms.

- An agent, $A$, is $a-$autonomous for a given goal according to a set of plans of some agent (either $A$ itself or another agent) to bring about that goal, if there is a plan in this set that contains actions all in the capabilities of $A$.

- An agent, $A$, is $r-$autonomous for a given goal according to a set of plans of some agent (either $A$ itself or another agent) to bring about that goal, if there is a plan in this set that contains
resources all owned by $A$.

- An agent, $A$, is $s$–autonomous for a given goal according to a set of plans of some agent (either the agent itself or another agent) to bring about that goal, if $A$ is both $a$–autonomous and $r$–autonomous for this goal.

Next, we formalise two possible interpretations of the meaning of goals. First, we define agents who can reason with respect to any goal in any agent's goal library using the $goals$ \textit{achieves} relation. It holds between an agent, a goal and a set of plans from the plan library if the plans are non-empty, and related to the goal according to the function $planlibrary$ from the $PlanAgent$ schema.

$$goalsachieves(A, g, ps) \iff (g, ps) \in (extdes A).planlibrary \land ps \neq \emptyset \quad (9.9)$$

Alternatively, agents may only be able to reason with respect to current goals. This scenario is formalised by the predicate, $achieves(A, g, ps)$, which holds precisely when an agent, $A$, has goal $g$, and the non-empty set of active plans $ps$ associated with $g$ in order to achieve it, according to the external description of $A$.

$$achieves(A, g, ps) \iff (g, ps) \in (extdes A).activeplanlibrary \land ps \neq \emptyset \quad (9.10)$$

In the following schema, we define these three classes of autonomy using the second relation, $achieves$. The second predicate states that an agent, $A$, is $a$–autonomous with respect to a current goal, $g$, according to some (non-empty) set of plans, $ps$, if and only if there is some agent, $C$, with goal, $g$, and plans, $ps$, to achieve $g$ such that some plan, $p$ in $ps$, contains actions all in the capabilities of $A$. Similar predicates are specified for $r$–autonomous. Since the variables $planactions$ and $planentities$ defined in Section 6.4 are $global$ variables, their definition is still in scope and they can therefore be used in the schema below.

\begin{center}
\begin{tabular}{|c|}
\hline
\textbf{AutonomyRelations} \\
\hline
\textbf{MultiAgentSystemExternalDescriptions} \\
achieves \textbf{\_} : \mathbb{F}(Agent \times Goal \times \mathbb{P} Plan) \\
a–autonomous \textbf{\_} r–autonomous \textbf{\_} s–autonomous \textbf{\_} : \mathbb{F}(Agent \times Goal \times \mathbb{P} Plan) \\
\hline
\end{tabular}
\end{center}

\[\forall A : Agent; g : Goal; ps : \mathbb{P} Plan \quad :\]

\[
achieves(A, g, ps) \iff (g, ps) \in (extdes A).activeplanlibrary \land ps \neq \emptyset \quad \land
\]

\[
a–autonomous(A, g, ps) \iff (\exists C : Agent \quad :achieves(C, g, ps)) \land
\]

\[
(\exists p : ps \quad : (planactions p \subseteq (extdes A).capabilities)) \land
\]

\[
r–autonomous(A, g, ps) \iff (\exists C : Agent \quad :achieves(C, g, ps)) \land
\]

\[
(\exists p : ps \quad : (planentities p \subseteq (extdes A).ownedresources))) \land
\]

\[
s–autonomous(A, g, ps) \iff a–autonomous(A, g, ps) \land r–autonomous(A, g, ps)
\]

As we have mentioned, however, this definition of $s$–autonomous does not entail that agents can achieve a goal using solely their own capabilities and resources. It makes better sense to say that an agent is only $s$–autonomous if there exists a specific plan that requires only the capabilities and resources of the agent, since it is only in this situation that the goal can be achieved independently of others. This notion is defined below.
An agent \( ag_i \) \( a \)-depends on another agent \( ag_j \) for a given goal \( g_k \), according to a set of plans \( P_{qk} \) if he has \( g_k \) in his set of goals, he is not \( s \)-autonomous for \( g_k \) and there is a plan in \( P_{qk} \) that achieves \( g_k \) and at least one action used in this plan is in \( ag_i \)'s set of actions.

An agent \( ag_i \) \( r \)-depends on another agent \( ag_j \) for a given goal \( g_k \), according to a set of plans \( P_{qk} \) if he has \( g_k \) in his set of goals, he is not \( r \)-autonomous for \( g_k \) and there is a plan in \( P_{qk} \) that achieves \( g_k \) and at least one resource used in this plan is in \( ag_i \)'s set of resources.

An agent \( ag_i \) \( s \)-depends on another agent \( ag_j \) for a given goal \( g_k \), according to a set of plans \( P_{qk} \) if he either \( a \)-depends or \( r \)-depends on this latter [agent]:

### Table 9.5: Original Definition of Dependence Networks

\[
\begin{align*}
\text{a}_{\text{dep}} (ag_i, ag_j, g_k, P_{qk}) \overset{\text{def}}{=} & \exists g_k \in G(ag_i) \neg a_{\text{out}} (ag_i, g_k, P_{qk}) \land \\
& \exists p_{ik} \in P_{qk} \exists i_m (p_{ik}) \in I(p_{ik}) a_m \in A(ag_j) \quad (9.12) \\
\text{r}_{\text{dep}} (ag_i, ag_j, g_k, P_{qk}) \overset{\text{def}}{=} & \exists g_k \in G(ag_i) \neg r_{\text{out}} (ag_i, g_k, P_{qk}) \land \\
& \exists p_{ik} \in P_{qk} \exists r_m \in R(p_{ik}) r_m \in R ag_j \quad (9.13) \\
\text{s}_{\text{dep}} (ag_i, ag_j, g_k, P_{qk}) \overset{\text{def}}{=} & a_{\text{dep}} (ag_i, ag_j, g_k, P_{qk}) \lor r_{\text{dep}} (ag_i, ag_j, g_k, P_{qk}) \quad (9.14)
\end{align*}
\]

### Table 9.6: Original Formalisation of Dependence Networks

- An agent, \( A \), is \( s \)-autonomous for a given goal according to a set of plans of some agent to bring about that goal if there is a plan in this set that contains both actions within the capabilities of \( A \) and resources owned by \( A \).

The quantification of the specific plan \( p \) in the set \( ps \), in the formal definition below, then ensures that the agent is both action-autonomous and resource-autonomous with respect to the same plan.

\[
s\text{-autonomous} (A, g, ps) \iff (\exists C: \text{Agent} \bullet \text{achieves} (C, g, ps)) \land \\
(\exists p: ps \bullet ((\text{planentities} p \subseteq \text{extdes } A) \land \text{ownedresources}) \land \\
(\text{planactions} p \subseteq (\text{extdes } A).\text{capabilities}))) \quad (9.11)
\]

Writing \( \text{achieves} (C, g, ps) \) and \( \text{extdes } A \) to represent \( C \)'s plans to achieve \( g \) and \( A \)'s external description, respectively, makes it explicit as to precisely whose external description is being used for any analysis.

This reveals one further issue. It can be seen by considering the previous schema that the definition of autonomy relies on the external descriptions of two agents: the plans and goals of \( C \), and the capabilities of \( A \). In the agency framework, autonomy is an absolute concept: an agent is either autonomous or it is not, and it does not rely on the relative views of two agents.

### 9.5 Dependence Relations

If agents are not autonomous with respect to a certain goal, then they may rely on others to achieve it. The dependencies that are considered between agents as originally formulated by Sichman et al. are presented in Table 9.5 and Table 9.6. Again we rewrite these textual definitions for clarity.
There is a plan to achieve $g$, if $g$ is a goal of $A$, $A$ is not $\text{a--autonomous}$ for $g$, and there is a plan in $ps$ that contains an action in $B$’s capabilities.

An agent, $A$, $r$--depends on another agent, $B$, for a given goal, $g$, according to some set of plans of some agent, $C$, to achieve $g$, if $g$ is a goal of $A$, $A$ is not $\text{a--autonomous}$ for $g$, and there is a plan in $ps$ that contains one of $B$’s resources.

An agent, $A$, $s$--depends on another agent, $B$, for a given goal, $g$, according to some set of plans of some agent, $C$, to achieve $g$, if $g$ is a goal of $A$, and $A$ either $r$--depends or $s$--depends on $B$.

The DependenceRelations schema formalises these definitions. As an example, the first predicate states that for two distinct agents, $A$ and $B$, a goal, $g$, and a set of plans $ps$, $A$ $\text{a--depends}$ on $B$ for $g$ with respect to $ps$, if and only if $g$ is a goal of $A$, $A$ is not $\text{a--autonomous}$ with respect to $g$ and $ps$, and there is some agent, $C$, with the goal, $g$, and plans to achieve $g$, $ps$, such that at least one plan in $ps$ has an action in the capabilities of $B$.

<table>
<thead>
<tr>
<th>DependenceRelations</th>
</tr>
</thead>
<tbody>
<tr>
<td>AutonomyRelations</td>
</tr>
<tr>
<td>$\text{a--depends}$, $\text{r--depends}$, $\text{s--depends}$ : $\mathbb{P}(\text{Agent} \times \text{Agent} \times \text{Goal} \times \mathbb{P} \text{Plan})$</td>
</tr>
<tr>
<td>$\forall A, B : \text{Agent}; g : \text{Goal}; ps : \mathbb{P} \text{Plan} \mid A \neq B$ $\bullet$</td>
</tr>
<tr>
<td>$\text{a--depends} (A, B, g, ps)$ $\iff$</td>
</tr>
<tr>
<td>$(g \in (\text{extdes} A).goals) \land \text{a--autonomous} (A, g, ps) \land$</td>
</tr>
<tr>
<td>$(\exists C : \text{Agent} \bullet \text{achieves} (C, g, ps) \land$</td>
</tr>
<tr>
<td>$(\bigcup { p : ps \bullet \text{planactions} p \cap (\text{extdes} B).\text{capabilities} \neq { } }) \land$</td>
</tr>
<tr>
<td>$\text{r--depends} (A, B, g, ps)$ $\iff$</td>
</tr>
<tr>
<td>$(g \in (\text{extdes} A).goals) \land \text{r--autonomous} (A, g, ps) \land$</td>
</tr>
<tr>
<td>$(\exists C : \text{Agent} \bullet \text{achieves} (C, g, ps) \land$</td>
</tr>
<tr>
<td>$(\exists p : ps \bullet \text{planentities} p \cap (\text{extdes} B).\text{ownedresources} \neq { }) \land$</td>
</tr>
<tr>
<td>$\text{s--depends} (A, B, g, ps)$ $\iff$</td>
</tr>
<tr>
<td>$\text{a--depends} (A, B, g, ps) \lor \text{r--depends} (A, B, g, ps)$</td>
</tr>
</tbody>
</table>

According to the SDN model, $A$ depends on $B$ if and only if there is a plan to achieve a goal of $A$’s that involves an action in the capabilities of $B$. However, in this situation it makes little sense to say that $A$ depends on $B$ for a goal if the actions that achieve that goal are also in $A$’s capabilities. In other words, the SDN formalisation of $\text{a--depends}$ does not define a dependence at all, rather it indicates the existence of a plan where there is a potential for $B$ to help $A$. Similarly, it makes little sense to say that $A$ depends on another agent for a resource if that resource is owned (in the SDN sense) by both.

If we accept the assumption that a resource cannot be shared then, since actions and resources are treated in the same way in the SDN analysis, it would also seem to indicate that no two agents may have an action in common, a possibility further supported by Table 9.1 which is used by Sichman et al. [147] to give examples of inter-agent dependencies. In this table all the resources and actions of the agents are distinct, whereas the goals are not. However, constraints that allow no shared resources or common actions between agents are severely limiting in describing general multi-agent systems.

A better definition of $A$ depending on $B$, which allows for agents having common actions, is that there is a plan to achieve $A$’s goal that requires an action that is in the capabilities of $B$ but not in the capabilities of $A$. This interpretation of dependence is defined below.

An agent, $A$, $\text{a--depends}$ on another agent, $B$, for a given goal, $g$, according to some set of plans of some agent, $C$, to achieve $g$, if $g$ is a goal of $A$, $A$ is not $\text{a--autonomous}$ for $g$, and there is a plan in $ps$ containing an action in $B$’s capabilities that is not in $A$’s capabilities. This is formalised by the following predicate.
\[ a \text{--depends } (A, B, g, ps) \Leftrightarrow (g \in (extdes A).goals) \land \\
\neg a \text{--autonomous } (A, g, ps) \land \\
(\exists C : Agent \bullet achieves (C, g, ps) \land \\
(\exists action : (\bigcup \{p : ps \bullet planactions p\}) \bullet \\
action \in (extdes B).capabilities \land \\
action \notin (extdes A).capabilities)) \] 
\text{(9.15)}

However, even when \( B \) is capable of an action of which \( A \) is not, and which is required in a plan to achieve \( A \)'s goal, it still makes little sense to say there is a dependence, because there may be other plans that do not require \( B \)'s assistance. It is more appropriate to say that there is a possibility of \( B \) being able to help \( A \) in achieving her goal because there happens to be a plan where \( A \) needs \( B \)'s help.

A further problem in the original definitions is that the dependence is evaluated with respect to a particular plan rather than a set of plans. A better notion of actual dependence between \( A \) and \( B \) with respect to a goal and a set of plans would be if every plan in the set of plans associated with \( A \)'s goal required \( B \)'s assistance. In this respect \( A \) would have a real dependence on \( B \) in order to achieve her goal, since the goal could not be achieved without \( B \)'s help. It may be that the authors are assuming that there is only ever one plan in the set of plans associated with a goal, a possibility enhanced by the example presented in Table 9.1. If this is so, it is misleading that the definitions always refer to a set of plans. However, the notion of dependence we describe here is defined and formalised below.

- An agent, \( A \), a--depends on another agent, \( B \), for a given goal, \( g \), according to some set of plans of some agent, \( C \), to achieve \( g \), if \( g \) is a goal of \( A \), \( A \) is not a--autonomous for \( g \), and every plan in \( ps \) contains an action in \( B \)'s capabilities that is not in \( A \)'s capabilities.

\[ a \text{--depends } (A, B, g, ps) \Leftrightarrow (g \in (extdes A).goals) \land \\
\neg a \text{--autonomous } (A, g, ps) \land \\
(\exists C : Agent \bullet achieves (C, g, ps) \land \\
(\forall p : ps \bullet \exists action : planactions p \bullet \\
action \in (extdes B).capabilities \land \\
action \notin (extdes A).capabilities)) \] 
\text{(9.16)}

There is, possibly, an argument about which of the alternatives is chosen, although in our view the second definition seems the more appropriate. However, the contribution we have made here is to show how it is possible to generalise the SDN model and remove the need for limiting assumptions so that the model can be used to describe systems that contain shared resources and agents having actions in common. By reformulating SDN we have shown that several options exist for a range of different types of dependencies. These definitions can easily be modified or updated, depending on which assumptions about the nature of the multi-agent system under investigation are chosen.

### 9.6 Dependence Situations

The total set of dependence relations produces a dependence network that can be used to classify distinct dependence situations between two agents shown in Table 9.7. We first note that there is inconsistency in Sichman’s definition of dependence. In the mathematical description, the dependence refers to the set of plans any agent has, but the diagrams refer only to the plans of the reasoning agent. In the interpretation that follows, we adopt the more restrictive version since it is consistent with the notions of independence and unilateral dependence discussed later, and is more intuitive in reflecting our understanding of the nature of autonomous agents in general.

Again, these categories would be more appropriately named if they were based on dependencies for actions that an agent does not have, as considered in the previous sub-section. For example, the SDN definition of \( A \)'s independence of \( B \) implies that it is not possible for \( B \) to help \( A \) in performing
**mutual dependence** is a situation where an agent \( ag_i \) infers that he and [an]other agent \( ag_j \) a-depend on each other for the *same goal* \( g_k \), according to a set of plans \( P_{qk} \).

**reciprocal dependence** is a situation where an agent \( ag_i \) infers that he and [an]other agent \( ag_j \) a-depend on each other, but for *different goals* \( g_k \) and \( g_l \), according to the sets of plans \( P_{qk} \) and \( P_{ql} \) (both sets belonging to the same external description entry).

**independence** is a situation where, using his own plans, \( ag_i \) infers that he does not a-depend on \( ag_j \) for \( g_k \).

**unilateral dependence** is a situation where, using his own plans, \( ag_i \) infers that he a-depends on \( ag_j \) for \( g_k \), but this latter does not a-depend on him for any of his goals.

<table>
<thead>
<tr>
<th>Table 9.7: Original Definition of Dependence Situations</th>
</tr>
</thead>
<tbody>
<tr>
<td>an action. A more intuitive definition of independence would be that ( A ) does not <em>need</em> ( B ) to perform any action.</td>
</tr>
<tr>
<td>Consider the definition of mutual dependence between ( A ) and ( B ). This states that ( A ) and ( B ) both have a goal ( g ), and according to ( A )'s plans, ( ps_A ) to achieve ( g ), there is some plan in which ( B ) could perform an action, and some plan (not necessarily the same plan) in which ( B ) could perform an action. Rather than mutual dependence, this categorisation describes a potential for cooperation (if the agents are autonomous, or mutual engagement otherwise). It merely distinguishes a situation where a single plan or possibly two plans can involve both ( A ) and ( B ). A more appropriate definition of mutual dependence should be that <em>every</em> plan in the set, ( ps_A ), <em>needs</em> both agents to act as discussed previously. A further alternative would be to define mutual dependence with respect to a <em>single</em> plan rather than a set of plans. Agents could then have a mutual dependence with respect to a plan if the plan required the capabilities of both agents.</td>
</tr>
<tr>
<td>Reciprocal dependence occurs when, according to two sets of plans, ( A ) could help ( B ) achieve some goal ( g_B ), and ( B ) could help ( A ) achieve some goal ( g_A ). The mechanism is described as <em>social exchange</em> and the authors state that “one of them will have to adopt the other’s goal first in order to achieve his own one in the future”. However, if agents can only reason with respect to sets of plans associated with a <em>current</em> goal, and since the definition is with respect to ( A )'s plans, ( A ) must currently <em>desire</em> both ( g_A ) and ( g_B ). This is restrictive since bargaining can occur (for example, when ( A ) and ( B ) have single different goals) in order for each to adopt the other’s goal. Even using the other interpretation of a goal as a library goal, the model is restrictive since both plans may be carried out concurrently, whereas the mechanism of social exchange does not acknowledge this possibility.</td>
</tr>
<tr>
<td>We now reformulate the original textual SDN definitions using our vocabulary. Consider the situation where two agents, ( A ) and ( B ), are such that ( A ) is <em>not a–autonomous</em> for some goal, ( g_A ), according to ( A )'s plans, ( ps_A ), to achieve ( g_A ). The following situations are recognised by Sichman et al. for action dependence.</td>
</tr>
<tr>
<td>• ( A ) is <strong>independent</strong> (<strong>IND</strong>) with respect to ( B ) for ( g ) if, according to ( ps_A ), it infers that it does not a–depend on ( B ) for ( g ).</td>
</tr>
<tr>
<td>• ( A ) is <strong>unilaterally dependent</strong> (<strong>UD</strong>) on ( B ) for ( g_A ) if, according to ( ps_A ), ( A ) a–depends on ( B ), but there is no goal for which ( B ) a–depends on ( A ).</td>
</tr>
<tr>
<td>• Two agents are <strong>mutually dependent</strong> (<strong>MD</strong>) if they a–depend on each other for the same goal ( g_A ) according to ( ps_A ).</td>
</tr>
</tbody>
</table>

158
If, in addition, $B$ is not $a$–autonomous for some other goal, $g_B$, according to $A$’s plans $ps_B$ to achieve $g_B$ then we can write the following.

- Two agents are **reciprocally dependent** (RD) if they $a$–depend on each other for $g_A$ and $g_B$, according to two sets of plans, $ps_A$ and $ps_B$, respectively.

We can now map these textual definitions onto formal ones. Given two agents, $A$ and $B$, where $A$ is not $a$–autonomous for some goal, $g$, we define the dependence situations in the schema below. As an example, reciprocal dependence is defined as follows. For any two agents, $A$ and $B$, any two goals $g_A$ and $g_B$ and any two plans $ps_A$ and $ps_B$ such that $A$ and $B$ are distinct, $g_A$ and $g_B$ are distinct, $ps_A$ is $A$’s set of plans to achieve the goal $g_A$, $A$ is not $a$–autonomous with respect to $g_A$ according to $ps_A$, $A$ is reciprocally dependent on $B$ according to the goals $g_A$ and $g_B$ and the plans $ps_A$ and $ps_B$ if and only if $ps_B$ are $A$’s plans to achieve $g_B$ and, according to $ps_A$, $A$ $a$–depends on $B$ for $g_A$, and according to $ps_B$, $B$ $a$–depends on $B$ for $g_B$.

<table>
<thead>
<tr>
<th>Dependence Situations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dependence Relations</td>
</tr>
<tr>
<td>IND $\models$ UD $\models F(Agent \times Agent \times Goal)$</td>
</tr>
<tr>
<td>MD $\models F(Agent \times Agent \times Goal \times P\ Plan)$</td>
</tr>
<tr>
<td>RD $\models F(Agent \times Agent \times Goal \times P\ Plan \times P\ Plan)$</td>
</tr>
</tbody>
</table>

\[
\forall A, B : Agent; g_A, g_B : Goal; ps_A, ps_B : P\ Plan \mid (A \neq B) \land (g_A \neq g_B) \land
\text{achieves} (A, g_A, ps_A) \land \neg \text{a–autonomous} (A, g_A, ps_A) \bullet
\]

\[
\text{IND} (A, B, g_A) \leftrightarrow
\neg \text{a–depends} (A, B, g_A, ps_A) \land
\text{UD} (A, B, g_A) \leftrightarrow
\text{a–depends} (A, B, g_A, ps_A) \land
\neg (\exists g : Goal; ps : P\ Plan \mid
\text{achieves} (A, g, ps) \bullet \text{a–depends} (B, A, g, ps)) \land
\text{MD} (A, B, g_A, ps_A) \leftrightarrow
\text{a–depends} (A, B, g_A, ps_A) \land
\text{a–depends} (B, A, g_A, ps_A) \land
\text{RD} (A, B, g_A, g_B, ps_A, ps_B) \leftrightarrow
\text{achieves} (A, g_B, ps_B) \land
\neg \text{a–autonomous} (B, g_B, ps_B) \land
\text{a–depends} (A, B, g_A, ps_A) \land
\text{a–depends} (B, A, g_B, ps_B) \land
\]

In the schema above, the \text{achieves} predicate is only ever used to describe the plans of $A$, which allows the easy checking of all dependence situations being categorised using only $A$’s plans.

### 9.7 Locally and Mutually Believed Dependence

Dependence situations are either locally or mutually believed. A dependence is local if it exists with respect to $A$’s plans but not with respect to $B$’s, and it is mutual if it exists with respect to both agents’ plans. For example, mutually believed mutual dependence between $A$ and $B$ exists when, according to both $A$’s plans and $B$’s plans, $A$ and $B$ $a$–depend on each other for the same goal. The original formalisation of this category can be seen below. For reasons of brevity we do not detail the others here.
However, there is an inconsistency between this description and the mathematical definition of mutually believed mutual dependence. The textual description taken from [146] is presented below.

“As an example, if \( i \) infers a MBMD between himself and \( j \) for a certain goal \( g \), this means he believes that (i) both of them have this goal and at least one plan to achieve it (ii) there is an action needed in this plan that he can perform and \( j \) cannot perform (iii) there is an action needed in this plan that \( j \) can perform and he cannot perform.” [146]

A translation of their mathematical definition, however, provides the following description.

\[
MBMD \equiv \text{MD}(ag_i, ag_j, g_k, P_{ik}) \land \text{MD}(ag_i, ag_j, g_k, P_{jk}) \quad (9.17)
\]

A refers a MBMD between himself and \( B \) if they both have goal \( g \); according to \( A \)’s plans \( A \) and \( B \) are not \( a \)-autonomous with respect to \( g \); according to \( B \)’s plans \( A \) and \( B \) are not \( a \)-autonomous with respect to \( g \); there is some plan of \( A \)’s, which contains an action, which \( B \) can do and a plan, possibly the same possibly not, that contains an action which \( A \) can do, and there is some plan of \( B \)’s that contains an action which \( A \) can do and a plan, possibly the same plan possibly not, that contains an action which \( B \) can do.

In particular, the mathematical definition does not specify actions not being in some agent’s capabilities, neither does it mention a certain plan within the set of plans. The set of mutually and locally believed mutual and reciprocal dependencies are formalised in the following schema. This schema emphasises that all categories require an analysis with respect to the plans of \( A \) and the plans of \( B \). In other words, mutual and local belief require an analysis of the plans of both agents that contradicts the following original claim.

“An agent locally believes a given dependence if he uses exclusively his own plans when reasoning about the others . . .” [147]
9.8 Summary and Conclusions

This chapter has examined the work of Sichman et al. in developing computational models of dependence networks, by reformulating them in the language and structure of the agency framework, agency relationship model and sociological agency model. It is chosen as a case study for three reasons: it is a useful contribution to the theory of multi-agent systems, it is a relatively new and ambitious theoretical development, and it is concerned with producing a taxonomy of inter-agent relationships. By applying our models we have isolated the vague and limiting assumptions on which SDNs are developed. This has been made possible because of a well-defined, universal vocabulary for describing multi-agent systems.

We have reformulated social dependence networks to present a stronger and more consistent formal model of which there are three aspects. First, implicit assumptions can be isolated and made explicit. Any theoretical developments are then more strongly underpinned and in a well-defined context. Second, restrictive assumptions about agents, such as no shared resources or no agents having actions in common, can be removed. This allows the model to be applied in more general systems. Third, we have addressed issues that have been neglected. Specifically, there was no undertaking to analyse the key agent relationships or how agents model their environment before the dependence networks were formulated.

This last point can be appreciated by referring to Figure 9.1. This shows that the schemas MultiAgentSystem and PlanAgent have been used as the basis from which to reformalise social dependence networks. It also shows that the schemas AgencySociety and SociologicalAgent were not used. This highlights what has been neglected in the original analysis: first, no analysis of the key social relationships was presented; and second, no model was developed to show how agents would interpret them.

This chapter has outlined a significant and immediate use for our work in this thesis in reformulating existing models and theories in general. First, by making explicit the conditions under which they are valid; second, by extending their application to more general agent scenarios; third, to consider the dimensions agents require in order to exploit them; and finally, to reveal details of any analysis that has been neglected in developing them.
Chapter 10

Conclusions

...our road does not lie on a level, but ascends and descends; first ascending to axioms, then descending to works.

— Francis Bacon, Novum Organum

10.1 Introduction

As we have described and as has been documented extensively elsewhere, multi-agent technology has a great deal to offer. Not only do multi-agent systems include all the standard advantages of distributed systems, they also provide a much greater degree of flexibility through the increased sophistication of the interacting agents.

It is a new and varied field, bringing together work from many different camps such as software engineering, artificial intelligence, cognitive science and sociology. However, and partly as a consequence of this diversity in its background, it suffers from having no commonly-agreed conceptual foundation. It is perhaps not surprising then that there is a risk of fragmentation in the field between various theoretical and practical camps. This is a serious concern, since if theoretical work is based on different concepts or languages, it becomes difficult to compare results or evaluate them in a wider context. Comparing and relating different work is a crucial requirement and one whose omission in the multi-agent field has often led to much unintentional re-invention of existing work. Just as critically, if systems cannot be related strongly to theory, then not only is the design of each system theoretically ungrounded, but experience gained from the deployment of a practical system cannot easily feed back into the theoretical work from which it is based. Moreover, there is then little chance that components of multi-agent systems can be integrated systematically.

Central to the functioning of multi-agent systems, both from the design perspective and from that of an agent interacting with others, is a precise knowledge of the fundamental inter-agent relationships that can arise in these systems. We have addressed this issue, whilst striving not to create just another theory, language or architecture.

Below, we provide an assessment of our work before summarising its contributions. The limitations of the approach we have taken and the methods employed are discussed and some potential areas for further investigation are outlined. Finally, we present some concluding thoughts.

10.2 Recapitulation

The work in this thesis can be evaluated under three different headings: generality, application and methodology. First, we discuss how our models have been structured and organised so that they are not biased towards any design or theory but are generic and encompassing with respect to the field in general. Second, we show the way in which our work has been applied to both existing and new
artifacts and ideas so as to demonstrate its practical benefits. Finally, we consider the methods we have employed.

10.2.1 Generality

The ability to conceptualise systems at different levels of abstraction is generally acknowledged as critical in the design of software systems. This is particularly true in many agent-based approaches that provide the means for abstracting out internal system features and interpreting or designing systems at higher levels such as the mentalistic or intelligent level. In our work, every model we have defined is related to every other model according to the level of abstraction at which it is described. We thus have a complete hierarchy of abstraction levels where each subsequent level is a refinement of the previous one.

The principal aim of this thesis is to determine and analyse fundamental relationships in multi-agent systems. In order to undertake an analysis of this kind it is first important to understand the nature of the agents themselves. However, if we are to build models of agents, it is critical that they are general, not biased towards any theory, architecture, design or implementation. This is not only so that the agent models can themselves be generally applied without prejudice to the wide spectrum of existing agent systems, but also so that the subsequent analysis of inter-agent relationships is also universally applicable. Developing generally applicable models of agents using generic abstract architectures has enabled us to use them as the basis from which to determine the key inter-agent relationships that arise naturally in multi-agent systems.

10.2.2 Application

One issue that has hampered the progress and development of the multi-agent field is the absence of any retrospective integration. As we have already suggested, existing ideas are often presented as new ones simply because a new conceptual foundation, research agenda or formal language is being employed. This arises because conceptual or theoretical foundations are are often uniquely tailored to the specific requirements of individual research agendas; agent frameworks are seldom built for application to other work.

To demonstrate that our models are generally applicable to existing research we have constructed formal models of two different multi-agent mechanisms and shown how these could be structured within the agency framework so that the nature of their sociological relationships could be highlighted. In addition, we have applied the agency framework to produce specifications of several single-agent architectures. Through this approach, the systems themselves can be evaluated in some coherent and widely-applicable framework. We claim that by directly applying our theoretical models to build specifications of implemented and theoretical single-agent and multi-agent systems mechanisms, we are relating theory to practice in a very strong and transparent way.

The first of the two multi-agent system case-studies is the contract net protocol, a mechanism for dynamic task allocation in distributed systems. It is one of the most common techniques employed in implemented multi-agent systems and there is a great deal of literature on its use in industrial applications. In addition, it is typically used to demonstrate new systems or theories, indicating a consensus in the agent community regarding its centrality to multi-agent systems. The second case-study covers the social reasoning mechanism based on social dependence networks which provide agents with a means for determining the interdependence of agents to achieve their goals. In contrast with the contract net protocol, the social dependence models are a relatively new theoretical development which arises out of related work on social interaction in sociology and psychology. It is therefore situated at the other end of the spectrum from the contract net protocol in terms of its nature, its age, its origin and its use in deployed systems. The third case study is AgentSpeak(L), which falls within the single-agent category. AgentSpeak(L) is important since it is an attempt to build a theoretical language that captures the essential operation of systems based on the belief-desire-intention model, of which there are many. Indeed this model is seen by some as the best example of deliberative intelligent agent architecture.
10.2.3 Methodology

We wish to be precise about the concepts and structures we introduce. Formal methods satisfy this requirement. However, the advantages of formality are best utilised if the resulting models are widely accessible, since then they can be employed by the wider community. In order to square our requirements for formality with the need to build practical systems, the model-based formal language Z is employed. It is a specification language most widely used to design software. It is a very expressive language, and has allowed us to provide a unified account of system structure, state and operation, enabling our specification to be well-structured, formalising and relating different levels of abstraction.

The incremental development of our work has been facilitated in Z by using schema inclusion. This can help ensure that at each new abstraction level, only the necessary details required to define an agent at that level are introduced and considered. In addition, new information can be formally related to existing information from previous levels.

It was possible to identify errors in AgentSpeak(L) not only through the process of re-specification itself, but also because of properties of the Z language. First, that fact that, in Z, preconditions for operations are given explicitly, enabled us to identify missing cases in the original description. Second, the property of Z being strongly-typed enabled tools to be used to automatically check the Z specification of AgentSpeak(L) for syntax and type correctness [155].

It is a mature language and there is a set of well-developed techniques for refining Z specifications to implementations and for proving implementations correct with respect to a design [77, 114, 140, 163, 168]. Z is exactly right for our needs then, ensuring accessibility because of the great wealth of case studies, documented industrial experience, text books and tool support that is widely available.

10.3 Contributions

There are a number of varied aspects to the work presented in this thesis. We summarise the key contributions below.

10.3.1 Agent Hierarchy

A generally applicable definition of agency has been supplied. This is not an attempt to create yet another agent definition, but one that relates to existing concepts and attempts to encompass them. In such a definition, we set up a sound conceptual base on which to elaborate more sophisticated definitions and architectures, while at the same time giving rise to a framework that can accommodate all types of agent be they non-computational, reflexive, or deliberative.

As set out in our requirements for formal frameworks in general, we have defined a language or vocabulary that provides accessible and precise meanings for common agent terms and concepts. A common conceptual framework is some way ahead, but this vocabulary is a tool that can help in moving towards this goal by, for example, isolating common components in agents, relating analogous abstractions in theories and enabling the comparison of agent designs. A possible side-effect is that it may stem the flow of diverse agent-related definitions that are continually entering the frame and causing unwarranted distraction.

In the view we propose, agents and autonomous agents are distinguished from each other as well as from non-autonomous entities, with clear and precise distinctions being made. Agents are defined as objects that can be ascribed goals, while the definition of autonomous agents as agents that can generate their own goals arises as a consequence. The hierarchy thus constructed is clean and simple, but embodies a sufficient set of concepts to provide a firm grounding for subsequent development and analysis.

The definitions of agency and autonomy are presented in base abstract architectures. Thus, even at the definition stage, our models have a direct link to issues involved in system design since they detail the interdependence of the various aspects of an agent’s data components and activity. This interdependence is described in these architectures using a functional model, which relates actions, goals, motivations (where appropriate) and perceptions. Once an agent is situated in an environment, values for the parameters of functions are available, and they can be evaluated to describe the agent’s...
state at run-time. Then, after the state is described, an operational model is defined that details the interaction of an agent with its environment as changes in that state.

The model supplied by the agency framework is universally applicable since it does not specify any internal architectural, design or implementation biases. All we require of agent systems is a minimal adherence to the set of basic action and perception functionalities that are specified as part of the model.

10.3.2 Agency Relationships

Arguably one of the most important results of the work described here is the way in which the analysis of fundamental inter-agent relationships has arisen neatly and naturally out of the agency framework. This occurs as a result of our definition of agents as objects with an ascribed set of goals, since it necessitates not only the existence of autonomous agents who can generate their own goals but the existence of goal adoption as the mechanism by which non-autonomous entities can become agents. Goal adoption is the key process in multi-agent systems, and once an agent has adopted the goal of another, there is an essential relationship between the two, with one entity acting on behalf of the other. Thus agency necessitates goal adoption, which in turn creates and defines the universal agency relationships of multi-agent systems.

We have a model that allows us to reason about how and why goal adoption takes place, a mechanism previously not investigated as an integral aspect of multi-agent systems. Agency relationships are dependent on the entities involved; whilst autonomous agents enter into relationships voluntarily, non-autonomous agents do not have this choice. Cooperation is therefore distinguished from engagement; whilst autonomous agents cooperate in voluntary relationships, non-autonomous agents are engaged in compulsory ones. We provide an understanding of the origin of these relationships, how and when they are initiated, manipulated, or broken and, further, how any changes may impact on other relationships. (For example, when an agent in an engagement chain is disengaged, this has implications for other agents lower down in the chain.)

Cooperation and engagement relationships can be analysed further to define a taxonomy of the different ways in which two agents can be related. This information is critical, not only since it is possible to understand the precise dependence of any two agents according to goal adoption, but also since it highlights the different interaction possibilities that are available to participating agents.

10.3.3 Agent Architectures

We have defined a set of agent architectural schemes that incorporate various dimensions commonly found in agents operating in multi-agent systems. They specify the requisite dimensions for effective interaction, providing a base architecture for deliberative agent design, a foundation from which to develop theories of social behaviour such as multi-agent planning or communication, for example, and an environment in which different mechanisms for integrating various agent dimensions can be analysed.

We have shown that in order to be generally effective in multi-agent systems agents need to be sociological. In other words, they need to have access to information about the agency relationships identified in this thesis, and recognise them in the system of which they are a part. This is not only so that they can reason about taking advantage of the current social configuration of agents, but also so that they can avoid unnecessarily affecting this structure in any potentially harmful or damaging way. We have provided an abstract architecture for such agents and demonstrated how their increased reasoning capabilities enable them to make more informed social decisions by specifying the multi-agent plans of agents, which contain actions to be performed by others. \textit{En route} to demonstrating the effectiveness of sociological agents, we have established a taxonomy of multi-agent plans, based on the current social obligations of the agents involved, which provides basic information to these agents.
10.4 Limitations and Further Work

The work described in this thesis is limited but provides the foundation for a great deal of subsequent research. Here, we outline the major limitations and the potential for further investigation.

10.4.1 Which Language?

Studies that use Z to specify examples such as communicating state machines [158], reactive systems [49] and concurrent systems [48] demonstrate its expressive power. However, any formal language is limited. Clearly, there are situations where a solely model-based approach is inappropriate as, for example, when conceptualising agent systems as a collection of interacting processes, when other languages such as CSP [81] or temporal logics [59] are more suitable. Thus, the agency framework is limited by its mode of expression. Whilst there are case studies to show how the complementary use of Z and CSP can be adopted [2, 90, 131, 169] we know of only one example that uses a combination of Z and temporal logic [43]. Further investigation, therefore, is needed on the integration of other formal modelling techniques with Z to describe the agency framework in order to increase its scope and applicability.

10.4.2 Agent Mechanisms

Although the focus of the work has been on analysing inter-agent relationships, there are many important agent mechanisms we have not explored such as goal generation, planning and communication. The first of these requires a model of motivations, their interaction, and the means by which they can be used to assess the relative benefits of competing goals. This is clearly a very large area of investigation. There is, for example, some notable work by Norman [125] in this field. Second, whilst stating that plans are necessary for effective behaviour, we have not considered the process of planning itself. However, it would be relatively straightforward to express algorithms for constructing and modifying plans within our work. Third, though we have not considered communication at all, we can now begin to address relevant issues since we have a strong conceptual platform from which to do so. Clearly, more work is required to integrate mechanisms such as these into our models.

10.4.3 Complexity

There has been no complexity analysis of how difficult it is to model and manipulate the agency relationships. Indeed, complexity is an issue that is missing from almost all multi-agent literature. As the field matures, such analyses will provide information regarding the suitability or feasibility of agent architectures for real systems of significant size. We have used our framework to sketch some preliminary results regarding the difficulty of constructing a set of agency relationships in order to satisfy a goal. Even in the most simple case, making severely limiting assumptions about the agents and their environment, our results show that the problem is NP-complete [41]. However, this work is at present immature and more effort is required to understand multi-agent system problems in terms of their complexity as well as designing algorithms to solve them.

10.4.4 Applicability

The specifications we have built by application of our work to existing single-agent and multi-agent theories and systems cover a varied but limited number of examples. Although a limited number does not guarantee the applicability or usefulness of work in general, we do assert that by deliberate selection of examples spanning lower-level, fundamental coordination mechanisms, through to sophisticated, mentalistic languages which integrate theoretical and practical issues and on to recent, ambitious, large-scale theories of social interdependence, even a limited number can demonstrate its utility in many situations. This is just what we have tried to do.

Mapping existing systems and architectures onto our formal agent framework should engender closer links between alternative agent models. The specifications contained in this thesis presents the first step along this path.
Our final aim is to develop a library of agent components, architectures and systems that are not tied to particular implementation platforms. One of the key overall project goals in pursuing work on formal specification of agent systems is to support the principled development of practical systems. We envisage a collection of specifications for both existing systems and distinct agent components from which a selection may be made as appropriate for the problem at hand, and used as the basis of implementation.

10.4.5 Design Methodology

If agent technology is to impact on the design of industrial systems, there needs to be a well-defined set of principles or methodologies for producing agent-based designs and specifications [92]. The structure of our work and the case study applications have suggested how this might be achieved by: analysing the components of a system in terms of the agency framework; determining the control or influence of agents on each other by a way that requires an analysis of agent relationships; and designing the functionality required by agents to act effectively. Some initial work has progressed in this direction, which opens up new areas of development [104].

10.5 Concluding Remarks

The lack of an agreed terminology or structure within which to pursue research in multi-agent systems is set to hamper the further development of the field if efforts are not made to address it. This thesis has described one such effort which has provided a framework that allows the development of diverse definitions, architectures, designs, theories and systems to be related within a single coherent whole. We have provided simple and encompassing definitions of agency and autonomy and explicated the relationship between them. These definitions are encapsulated in abstract general architectures that can be applied to analyse inter-agent relationships and refined to provide more detailed architectures for the deliberative dimensions required by more practical agents. The usefulness of our work has been demonstrated by elaboration and refinement to the point where we have been able to present descriptions, stated in terms of our formal models, of both implemented systems and theoretical models.

There is currently a high financial investment in agent products and it is becoming a mature technology. We must therefore address the need to develop agent-based tools, supported by a strong conceptual foundation, which can be used to design, specify and implement such systems. The work in this thesis is one attempt to build such a tool that spans a range of levels of definition and abstraction, in order to provide a common integrated framework within which different levels of reasoning, behavioural and interaction tasks can be related and considered through design and specification. If the field of multi-agent systems is to progress in a rigorous, disciplined and, perhaps most importantly, accessible way, then efforts such as this, which seek to provide a common unifying foundation for a diverse body of work, in which theory and practice can be strongly related, are essential.
Bibliography


168


179

Appendix A

Generic Z Definitions

Below we provide the standard library of mathematical definitions that are used in this thesis. A complete set of definitions can be found in The Z Handbook [156] on which the definitions below are based.

A.1 Sets

A set, \( S \), is a subset of another set, \( T \), if whenever an element is contained in \( S \), it is also contained in \( T \). \( S \) is a strict subset of \( T \) if \( S \) is a subset of \( T \) but \( S \) is not equal to \( T \).

\[
\forall S, T : \mathbb{P} X \bullet
(S \subseteq T \iff (\forall x : X \bullet x \in S \Rightarrow x \in T)) \wedge
(S \subseteq T \Rightarrow S \subseteq T \wedge S \neq T)
\]

For any set \( X \), \( \mathbb{P}_1 X \) is the set of all subsets of \( X \) which are non-empty.

\[
\mathbb{P}_1 X == \{ S : \mathbb{P} X \mid S \neq \{ \} \}
\]

\( S \cup T \) are those objects which are members of \( S \) or \( T \) or both. \( S \cap T \) are those objects which are members of both \( S \) and \( T \). \( S \setminus T \) are those objects which are members of \( S \) but not of \( T \).

\[
\forall S, T : \mathbb{P} X \bullet
S \cup T = \{ x : X \mid x \in S \lor x \in T \} \wedge
S \cap T = \{ x : X \mid x \in S \land x \in T \} \wedge
S \setminus T = \{ x : X \mid x \in S \land x \notin T \}
\]

The generalised union of a set of sets contains every object that is in at least one of the sets. The generalised intersection of a set of sets contains those objects that are contained in every set.
The functions, \textit{first} and \textit{second} return the first and second element of an ordered pair, respectively.

\begin{itemize}
\item \textit{first} : \(X \times Y \rightarrow X\)
\item \textit{second} : \(X \times Y \rightarrow Y\)
\end{itemize}

\begin{align*}
\forall x : X; y : Y \bullet \\
& \text{first}(x, y) = x \land \\
& \text{second}(x, y) = y
\end{align*}

\section*{A.2 Relations}

The domain of a relation, \(R\), with source set \(X\) and target set \(Y\) is all the elements of \(X\) that are related to at least one element of \(Y\). The range of a relation, \(R\), with source set \(X\) and target set \(Y\) is all the elements of \(Y\) that are related to at least one element of \(X\).

\begin{itemize}
\item \textit{domain} : \((X \leftrightarrow Y) \rightarrow \mathcal{P} X\)
\item \textit{range} : \((X \leftrightarrow Y) \rightarrow \mathcal{P} Y\)
\end{itemize}

\begin{align*}
\forall R : X \leftrightarrow Y \bullet \\
& \text{dom} \ R = \{x : X; y : Y \mid (x, y) \in R \bullet x\} \land \\
& \text{ran} \ R = \{x : X; y : Y \mid (x, y) \in R \bullet y\}
\end{align*}

The anti-domain restriction of a relation, \(R\) with source set \(X\) and target set \(Y\), with respect to a set \(S\) of type \(\mathcal{P} X\), are all the pairs of \(R\) whose first element is not a member of the set \(S\).

\begin{itemize}
\item \(\ll S : \mathcal{P} X; R : X \leftrightarrow Y \bullet \\
& S \ll R = \{x : X; y : Y \mid x \not\in S \land (x, y) \in R \bullet (x, y)\}\)
\end{itemize}

An object \(y\) is related to an object \(x\) by the relation inverse \(R^{-1}\) of \(R\) if and only if \(x\) is related to \(y\) by \(R\).

\begin{itemize}
\item \(\ll^{-1} : (X \leftrightarrow Y) \rightarrow (Y \leftrightarrow X)\)
\end{itemize}

\begin{align*}
\forall R : X \leftrightarrow Y \bullet \\
R^{-1} = \{x : X; y : Y \mid (x, y) \in R \bullet (y, x)\}
\end{align*}
The relational image \( R[\mathcal{S}] \) of a set through a relation \( R \) is the set of all objects \( y \) to which \( R \) relates to some member \( x \) of \( S \).

\[
[X, Y] - \mathcal{R}[\mathcal{S}] = (x \ni y) \times \mathcal{P} X \to \mathcal{P} Y
\]

\[
\forall R : X \leftrightarrow Y; S : \mathcal{P} X \bullet \\
R[\mathcal{S}] = \{ x : X; y : Y \mid x \in S \land (x, y) \in R \bullet y \}
\]

A homogeneous relation is a relation whose source set equals the target set. The transitive closure of a homogeneous relation \( R \) is the smallest relation containing \( R \) that is transitive.

\[
[X] - \mathcal{R}^+ = (x \ni x) \to (x \ni x)
\]

\[
\forall R : X \leftrightarrow X \bullet \\
R^+ = \bigcap\{ Q : X \leftrightarrow X \mid R \subseteq Q \land Q \circ Q \subseteq Q \}
\]

The relation \( Q \oplus R \) relates everything in the domain of \( R \) to the same objects as \( R \) does, and everything else in the domain of \( Q \) to the same objects as \( Q \) does.

\[
[X, Y] - \mathcal{R} \oplus S = (\text{dom } S) \sqsubseteq R \cup S
\]

A.3 Functions

A partial function is a relation such that no element in the domain maps to more than one element in the range. A total function is defined for every element in the source set. A partial/total injection is a partial/total function such that no two elements in the domain map to the same element in the range. A partial/total surjection is a partial/total function which is defined for every element in the target set. Finally, a bijection is a function which is total, injective and surjective.

\[
X \rightarrow Y \equiv \{ R : X \leftrightarrow Y \mid (\forall x : X; y, z : Y \bullet (x, y) \in R \land (x, z) \in R \Rightarrow y = z) \}
\]

\[
X \to Y = \{ f : X \to Y \mid \text{dom } f = X \}
\]

\[
X \leftrightarrow Y = \{ f : X \leftrightarrow Y \mid (\forall x_1, x_2 : X \bullet f(x_1) = f(x_2) \Rightarrow x_1 = x_2) \}
\]

\[
X \Rightarrow Y = (X \to Y) \cap (X \leftrightarrow Y)
\]

\[
X \rightarrow Y = (X \to Y) \cap (X \Rightarrow Y)
\]

\[
X \rightarrow Y = (X \to Y) \cap (X \rightarrow Y)
\]

A.4 Sequences

The expression \( m \ldots n \) defines the set of all numbers from \( m \) to \( n \) inclusive.
A sequence of type $X$ is represented as a partial function from natural numbers to $X$ such that the domain is a set of contiguous numbers from 1 up to the number of elements in the sequence. An injective sequence has no repeated elements. The set of non-empty sequences and non-empty injective sequences are also defined.

\[
\text{seq } X = \{ f : \mathbb{N} \to X \mid \text{dom } f = 1..\#f \}
\]
\[
\text{seq}_1 X = \{ s : \text{seq } X \mid \#s > 0 \}
\]
\[
\text{iseq } X = \text{seq } X \cap (\mathbb{N} \to X)
\]
\[
\text{iseq}_1 X = \text{iseq } X \cap \text{seq}_1 X
\]

The concatenation of two sequences $s$ and $t$ contains the elements of $s$ followed by the elements of $t$.

\[
\text{sequences } X \\
\text{seq } X \times \text{seq } X \to \text{seq } X
\]

\[
\forall s, t : \text{seq } X \\
s \cdot t = s \cup \{ n : 1..\#t \cdot ((n + \#s), t(n)) \}
\]

The head and last of a non-empty sequence are the first and last elements of that sequence, respectively.

\[
\text{sequences } X \\
\text{head, last : seq}_1 X \to X
\]

\[
\forall s : \text{seq}_1 X \\
\text{head } s = s(1) \land \\
\text{last } s = s(\#s)
\]

One sequence is a subsequence of another if the latter contains the former.

\[
\text{sequences } X \\
\text{in : seq } X \leftrightarrow \text{seq } X
\]

\[
\forall s, t : \text{seq } X \\
s \in t \iff (\exists u, v : \text{seq } X \cdot u \cdot s \cdot v = t)
\]
Appendix B

Schema Structure for Specifying the Sociological Agency Model

This appendix provides the diagrams which illustrate the specification structure used to define the Sociological Agency Model in Chapter 6.
Figure B.1: Schema Structure for Specifying Store-Agents

Figure B.2: Schema Structure for Specifying Agent Models

Figure B.3: Schema Structure for Specifying Sociological Agency
Figure B.4: Schema Structure for Specifying Sociological Plan-Agents

Figure B.5: Schema Structure for Specifying Sociological Plan Categories