Agents, Contexts, and Logic.

Thesis submitted in accordance with the requirements of
the University of Liverpool for the degree of Doctor in Philosophy

by

Anthony John Hepple

November 17, 2010

Abstract

There is an emerging computational landscape in which processing is becoming increasingly concurrent due to the number of devices, the interconnection of these devices, and the use of multi-core processors. As the landscape changes, so do its inhabitants. No longer do users see technology as an intrusion into their daily lives, indeed many novel applications for computing technology are being found and areas of our lives are already dependent upon such technology. Since we expect this trend to continue, they must lead to the ubiquity of computing devices.

The aim of this thesis is to tackle one of the many challenges that such a landscape presents — that of programming coherent, reliable software at a high level of abstraction. Such software must be able to cope with the dynamic nature of its environment, adapt to the changing needs of its users, and do so without excessive human intervention.

The view adopted here is to take a principled approach to programming, that is inspired by formal logical methods and agent theory, and involves the direct execution of temporal logic specifications. As part of this work, a robust implementation of METATEM, the agent-oriented programming language based on executable specifications, was developed and applied. This direction, when combined with concepts such as contexts, preferences and constraints, provides a promising means of capturing the essence of context-sensitive software, typical of pervasive and ubiquitous computing scenarios.

This thesis provides a formal semantics for the executable specification language, and its extensions, and describes the implementation of a concrete system. The work demonstrates the flexibility, power, and clarity of this approach and hence justifies its candidacy for programming a future generation of computer systems.

Dedication

To Abigail and Sophie, thank you for the cycle rides and other great distractions, your exuberance and enthusiasm have been invaluable.

Acknowledgements

Firstly, I thank Professor Michael Fisher, for his patient support, considerable knowledge, unerring direction and the many other ways he helped me throughout this project and influenced this work. I am very grateful for the opportunity to work with him and for the time, comittment and dedication he has provided to myself and this project.

There are many others within the Department of Computer Science that I am also grateful to. In particular I would like to thank Dr. Louise Dennis for her input and collaboration, and my internal examiner, Dr. Peter McBurney, for his thorough and constructive criticism of my work. I also thank the members of the Logic and Computation Group, especially my fellow PhD students, for creating a welcoming and supportive environment.

Thanks also go to my external examiner, Professor Simon Dobson (University of St. Andrews), for his careful examination of my thesis and insightful comments during my viva.

Finally, I would like to thank my parents and Deborah for allowing me this indulgence and for providing me with unconditional practical and emotional support. I realise that there have been times during this project when my preoccupation has given them cause for concern and thank them for their unwavering love, support and encouragement.

Contents

List of Figures

Chapter 1

Introduction

So many areas of our lives are enhanced by computers and the software they run. Software is often, and appropriately, referred to as a tool, particularly when it supports its user's activity by making that activity easier in some way, commonly resulting in a time-saving benefit. Possibly the most popular example of software as a tool is the word-processor. In comparison with its mechanical predecessor — the typewriter — word-processing software greatly speeds up the task of preparing documents. However, the word-processor does not only provide time savings, it also adds to the functionality of its predecessor. The ability to rearrange content, change its style and print multiple copies are simple features that give word-processors great advantages over typewriters. Yet the humble word-processor brought other more significant features which were considered to be intelligent when they were first introduced; features such as automated grammar checking and correction, that are taken for granted today, provided a remarkable additional benefit to users. Like all good tools, when used correctly, computers enable users to perform tasks that may not just be more time-consuming without the tool but may not even be possible without them. This increasing level of sophistication and apparent intelligence must influence the way that computing devices are perceived by their users.

Software is available for a wide variety of purposes. For example, a computer can provide help with financial transactions, directing traffic, communication and mathematical modelling. As diverse and varied these applications of computing technology are, it is notable that each software tool remains singular in purpose and that they generally operate in isolation. With the exception of applications with closely related purposes,[1] the integration of applications with others of essentially different purposes is not generally considered during their design, is usually only possible with the use of an API and is therefore a difficult task that requires expert knowledge.

[1] Such as collections of enterprise level applications and suites of personal productivity tools.

1.1 Trends

The introduction of computers into wider society (wider than specific industrial, academic and defense applications) is a recent event, in fact there are some sections of even the most developed societies that do not yet interact directly with computers — as well as Luddite attitudes we can identify a significant number of the older generations in all societies — and yet the scale and mode of interaction that does take place has changed remarkably in just a few decades. The first computers were individually vast, occupying whole buildings but affording little convenience to their sole user. Advances in operating systems enabled mainframe computers to share their processing power around several users, allowing each user to interact in an apparently simultaneous way via individual terminals. Advancements in hardware led to the personal computer and a return to a one-to-one correspondence between devices and users. The present-day sees the number of devices exceeding the number of users. Not only is it common for users to interact with many devices (an enthusiastic adopter of technology can be bristling with pocket size devices and own several larger machines) but it is also common for devices to interact with many users. Although computing devices are more numerous, more portable and less costly than in any time in history, and despite novel and imaginative advancements in human-computer interaction that often make these interactions more intuitive, the interaction of man and machine is still very much deliberate (on man's part at least). Not only is this interaction deliberate but its nature is interventional. Its mode originates from mechanical typewriters and early (albeit pioneering) work carried out by Xerox Laboratories. The keyboard and mouse have evolved since their introduction, they have changed in shape, size and construction, even merging into other hardware (e.g. touch sensitive displays) making our interaction with computer systems more convenient, arguably more efficient, but no less deliberate. Adhering to this form of interaction with conventional systems requires conscious cognitive effort in the form of intent, focus, skills to operate the hardware, and knowledge of the system's purpose. All of this is in addition to any effort induced by the context of use. It is inconceivable therefore that the user is not aware of the system.

If we contrast this with the use of other successful tools, for example a pencil whose use is apparently subconscious, we are unaware of the tool when using it, thus enabling all of one's focus to be applied directly to the activity. As for any tool, the design of software strives for this level of usability.

1.2 Advances

Strong evidence of the potential for software to achieve this level of usability is provided by examples of applications which demonstrate — albeit in isolation — many features associated with context-aware computing. Further still, many of these novel applica-

tions are backed by significant commercial organisations that not only have an interest in following trends but also the power to influence future usage trends.

Latitude[2] is an application for GPS enabled mobile phones that maintains your location and activity on a shared map, enabling a user to see their friends' locations and share their own with them.

Bump[3] allows users of Apple's iPhone to exchange contact details by bumping two iPhones together. (Bump is available now.)

Layar[4] overlays information from the Internet onto a real-time image of its users' surroundings (captured by a smart-phone's built-in camera). Popular sites with user-contributed content, such as encyclopedia and photograph sharing sites, are used to augment your actual view of the world.

ShopSavvy[5] enables shoppers to scan the bar-code of any product using a phone's built-in camera, view the results of an Internet search for the item and, if available, buy the item at a more competitive price.

Barclays[6] bank have, since early 2009, been issuing payment cards that allow contactless transactions. Once a significant number of account holders have a card with this capability, they hope to be able to persuade retailers to invest in the necessary point-of-sale equipment. When contactless payment is established the payment card will no longer be restricted to its current form. It could, for example, be embedded within a mobile phone.

Fuel Prices[7] is another smart phone application that locates the five cheapest petrol stations in the vicinity of its user. It uses the phone's GPS functionality to determine the location of its user and a database of credit-card transactions to determine current prices. (Fuel Prices UK is available now.)

Each of these examples is deployed or in the later stages of development and all appear to have been warmly received by users.[8]

These examples support the argument that the increased intelligence exhibited by software is transforming software from esoteric tools that support well- and pre-defined

[2] `http://www.google.com/latitude`
[3] `http://www.bumptechnologies.com`
[4] `http://layar.com`
[5] `http://www.biggu.com/apps/shopsavvy-android/`
[6] `http://www.barclays.com/contactless`
[7] `http://www.mubaloo.com/pages/mubaloo_fuelprices.php`
[8] Although there have been some concerns from privacy activists about Google's latitude application. `http://www.privacyinternational.org/article.shtml?cmd[347]=x-347-563567`

human activity into more sophisticated, autonomous and multi-purpose systems that enhance more general human activity with fewer deliberate human-computer interactions. Today, computing devices and the software they execute (without which they would be useless) are increasingly referred to, not as tools, but as assistants.

1.3 Ubiquity

In the previous section we discussed computers as tools that support human activity in a passive manner, acting only when explicitly commanded to do so. However, software is expected to be more than a tool, it has become an ever-present assistant. A personal assistant. Many people have their mobile telephones at hand for 24hrs a day. The term 'smart phone' flatters the telephone functionality of these small-form personal computers, as this is no longer the predominant function of the device (as is demonstrated by some of the applications mentioned in Section 1.2).

Imagine that you maintain an electronic shopping list. When travelling home, you notice a shopping centre and decide it would be convenient to buy some items from your list. Whilst walking through the shopping centre, your *personal assistant*, unprompted by you, is negotiating with nearby stores in order to find the items on your list at a competitive price and convenient location. Whenever in the vicinity of stores with appropriate stock, your assistant alerts you and when making a purchase your list is automatically updated as the assistant is also your payment device.

Such a scenario is not the sole domain of science fiction writers; similar scenarios have been attracting the attention of computer scientists since Weiser's visions [128] in 1993. He imagined computer systems *disappearing* into our environment such that their presence and their use, like any ubiquitous object, is taken for granted. Pervasive computing is one realisation of Weiser's foresight. A pervasive system is one that, rather than residing on one clearly defined device (or network of devices) and being constrained to the boundaries of that device, pervades the environment of its user(s), making use of resources available to it in an apparently ad hoc fashion. It is characterised by wireless networked devices forming an open network in which software with a high degree of autonomy and intelligence can sense and adapt to the changing context of its users' actions. Many novel applications of pervasive system technology have been proposed, including smart houses, assisted living, location awareness and calm technologies [130].

The shopping scenario introduced above is used later in this thesis, for illustrative and evaluative purposes. Whilst the author believes this to be a realistic future application of computing technology, it was not their intention to narrow the scope of this work to such an application, nor to propose it as an exemplar. As stated, its purpose was to serve as just one case-study that demonstrates certain characteristics of context-aware systems and exposes some of the difficulties that must be overcome if we are to program reliable systems within a highly distributed computing environment.

1.4 Programming paradigms

Imperative programming languages have been dominant since the birth of the the software industry. Their origins can be traced back to the assembly languages of computers with the dominant von Neumann architecture. Imperative languages have of course evolved, gaining new constructs and syntax, in response to both the needs of developers and the applications they create. Techniques for restricting the scope of data, and improving the modelling of the real-world are exemplified by imperative programming in its latest guise — object-oriented programming. At the same time imperative languages have adopted constructs for splitting application code into multiple threads of execution and dealing with the problems that arise when these threads have shared access to data. However, despite their improved syntax and additional constructs, they remain imperative in nature. The programmer must describe both *what* their code must achieve and *how* it must achieve it. That is, in writing imperative code, the programmer must explicitly give the logic and execution order of all statements. An algorithm is both the goal and a solution, leading to an inevitable confusion of the two.

Since it is usually of no consequence to the user how their software satisfies their requirements, only that it does so, it is desirable to be able express just these requirements in a succinct and precise manner. Indeed, this is often the output of the requirements analysis process of many software engineering methodologies, and is used to verify that software is implemented correctly. Unfortunately, automated methods for verifying that imperative implementations match their specifications, where possible, are not practical for most applications.

In contrast, declarative languages and in particular, logical declarative languages, provide a goal-driven approach to programming which, in most cases, relieves the programmer of the burden of describing the path to a solution but instead allows them to focus on describing properties of the solution. In essence, a program of declarative logic statements is a formal theory where execution of that theory consists of one or more deductions that are consistent with the theory. Such languages tend to provide far greater clarity of programming akin to that of a formal specification of requirements. Furthermore, the lack of side-effects in declarative languages lends themselves to concurrent programming, where side-effects are difficult to manage. But perhaps the greatest motivation for the use of a declarative approach to programming is the potential for automated verification, possible due to the close correspondence between specification and implementation languages.

1.5 Agency

Given the prevailing trends of computing hardware (interconnection, cost and performance), those of programming (greater abstraction and human-orientation), and those

of intelligence and ubiquity associated with the use of computing, the popular concept of agency is a complementary choice for engineering distributed systems [20].

Of the many challenges highlighted by the Ubiquitous Computing Grand Challenge [66] it is the high-level modelling and specification of such systems that this project focused on. Current software engineering techniques and tools do not have useful abstractions for, and are unable to express many of the desired behaviours of, intelligent distributed systems, such as context-awareness. Thus we took an agent-oriented stance towards the modelling of these systems, which was, we believed, sensible due to the autonomous and proactive nature of agents. *Proactive*, *autonomous*, *adaptive* and *fault-tolerant* are adjectives commonly used to describe distributed context-aware systems.

All but the most trivial of agent-based systems are modelled as multi-agent systems where each agent has distinct goals, and therefore purpose, and enjoys individual autonomy. Agents combine, often in a co-operative way to satisfy the end-user requirements of a given application. It is inconceivable that a pervasive computer system for example, that is modelled using an agent-oriented approach, will not comprise many, many agents.

The agent abstraction is often modelled as isolated pairs of agent and environment, with each agent being aware of other agents only by sensing changes in their environment brought about by the actions of those other agents. Direct communication between agents greatly facilitates their ability to cooperate but many in the agent research community believe that an organisational abstraction must be introduced to the multi-agent modelling process if individual agent's efforts are to support a system's global aim. Thus, how agents in a multi-agent system are organised is a key problem that currently holds much attention from the agent research community [9, 98]. In Chapter 4 and, in more detail in [74], reviews of some of the many proposed agent organisation strategies are presented, with the aim of uncovering the key concepts of agent-organisation.

Finally, the agent research community is motivated not only by the potential of this field but also by its successes. Agent-based techniques have been used with great effect in applications as critical as air traffic control [125] and space-craft management [101].

1.6 Logic

Traditional Predicate Calculus [82] forms the basis for most logic-based programming languages. The most wide-spread example of these languages is Prolog, whose programs contain facts and clauses, and are 'queried' rather than executed. Knowledge, in Prolog, bears no direct relationship to time and therefore concurrency can only be modelled by applying meaning to standard predicate symbols, with an inevitable loss of clarity and concision. Pnueli proposed a logical method of reasoning about time-dependent events

that allows the specification and verification of concurrent programs. His temporal logic of programs [102] has inspired much work in this particular aspect of formal methods and has led to the concept of directly executable specifications. In METATEM [47], the language employed to support this thesis, statements of temporal logic are 'executed' by a process of repeated deductions that result in a sequence of logical interpretations. Each interpretation represents a distinct moment in time that is logically consistent with all previous moments. Such a model, σ, is represented by

$$\sigma = \langle s_0, s_1, s_2, s_3, \ldots \rangle$$

where each state, s_i, represents a moment in time and where s_0 represents the starting/initial state. Thus, the model has, from any state, a finite past and an infinite future.

Discrete, linear-time, temporal logic provides several operators for reasoning over the interpretations of temporal states. Three common operators are the '$\bigcirc$' ("at the *next* moment in time"), '$\Box$' ("at *all* future moments in time") and '$\Diamond$' ("at *some* future moment in time") operators. Formal semantics for these operators (in a model σ at a moment i) are given below.

$$\begin{array}{lcll} \langle \sigma, i \rangle & \models & \bigcirc \psi & \text{iff} \quad \langle \sigma, i+1 \rangle \models \psi \\ \langle \sigma, i \rangle & \models & \Diamond \psi & \text{iff} \quad \text{there exists } j \geq i \text{ such that } \langle \sigma, j \rangle \models \psi \\ \langle \sigma, i \rangle & \models & \Box \psi & \text{iff} \quad \text{for all } j \geq i.\ \langle \sigma, j \rangle \models \psi \end{array}$$

Thus, using this logic, the dynamic behaviours of an agent can be expressed intuitively. Statements of future intentions can be written with great clarity and concision, for example

$$\begin{array}{rcl} \mathit{academic} & \Rightarrow & \Box \mathit{writing} \\ \Box \mathit{writing} & \Rightarrow & \Diamond \mathit{published} \\ \mathit{published} & \Rightarrow & \bigcirc \mathit{academic} \end{array}$$

METATEM is a programming language in which programs are temporal logic specifications and multiple programs can be executed asynchronously, with message-passing communication. Using only the modal operators $\Box$, $\Diamond$ and $\bigcirc$ to convey temporal semantics of *always*, *eventually* and *in the next moment* respectively, this language is well suited to capture high-level requirements of a class of systems characterised by continuous concurrent execution [48].

1.7 The future

The changing landscape of computing brings with it many challenges; understanding these challenges is the first step towards solving them. Thus much justified effort has

been directed to characterising the pervasive systems of the future. Research by Coutaz, Dobson *et al.* [25, 40, 39, 41] and the UK's Ubiquitous Computing Grand Challenge [65], each describe requirements for clear, focused and principled design and development of such systems.

Key characteristics that occur repeatedly in these and other authoritative works include:

Context: *The requirement for software to modify its behaviour appropriately when the behaviour of entities in its local environment changes requires rich models of context and tractable methods of reasoning about context.*

Mobility: *An infrastructure comprised of a multitude of wirelessly connected mobile devices with limited resources implies that software must also be mobilised, allowing it to migrate for efficiency, security and reliability reasons. Movement not only produces changes in location, but also in context and possibly in context- or resource-dependent behaviour.*

Responsibility: *Any action made by a system, in a dynamic environment without central authority, is made without full and certain knowledge of the system's state. Actions may produce undesirable environment states or unwanted side-effects which adaptive systems should be able to detect and mitigate the effects of.*

Organisation: *Pervasive/Ubiquitous computing suggests a proliferation of single purpose agents which, with organisation, collectively perform useful tasks. Thus, "interconnection is more important than data" [41]. So, whilst each agent may be described as autonomous it must have a clear structural position if the interaction necessary to ensure security, efficient communication and much more, is to be produced.*

Uncertainty: *In dynamic and unpredictable environments, individual events cannot be reliably used as the triggers for system behaviour. Similarly, in systems composed of multiple autonomous entities, each with the ability to provide information about its environment, we must expect discrepancies and uncertainties to occur.*

Dependability: *Widespread deployment and adoption of such technology will not be achieved until high levels of dependability are reached. Many application areas expected to exploit this technology are of a safety critical nature or carry security and privacy risks which require levels of dependability only possible through formal techniques.*

These are radical changes to the current dominant computing platform, which if realised, will enable software to become less dependent upon user interactions, more adaptable to subtle changes in context of use.

In summary, the author believes that a declarative language based upon temporal logic provides an opportunity for software development to adapt to these trends, harness the potential of the concurrency implied by these systems, with accuracy and concision. All of this, while potentially benefiting from the advantages of formal methods such as true verification in the future. Thus, this thesis evaluates to what extent METATEM can be extended with multi-agent organisation concepts and context notions in order to provide a basis for the principled programming of pervasive systems.

1.8 Aims and achievements

As is to be expected, the general evaluation aim as described above, can be divided into many and varied sub-tasks. Whilst this document aims to present a palatable summary of the project, there have been many other outputs of the project and these sub-tasks. Furthermore, the life of a PhD student is not entirely occupied by their project (despite its domineering influence), he or she is encouraged to participate in Departmental activities in relation to both teaching and research, such as conducting undergraduate seminars and publishing research papers. This final introductory section describes some of the aims, achievements and outputs that are not explicitly mentioned elsewhere in the thesis. Additionally, this project was able to support two final-year undergraduate projects and a vacation project, which were co-supervised by Michael Fisher and the author.

Clearly, the argument for the use of a given technology is supported by both theory and practice, therefore with the aim of evaluating the use of METATEM with respect to pervasive computing, and of adding weight to the arguments of this thesis, a robust, reliable and efficient implementation of an agent interpreter was essential to the success of the project. This realisation put significant focus on the design, coding and documentation of the METATEM interpreter and germinated at least two undergraduate software projects. Whilst this thesis, rightly focuses on the fundamentals of the approach, it perhaps does not convey the amount of effort expended by those involved in its design and implementation, nor the potential future utility of a perspicuous and maintainable code base. Specifically, the project has directly led to the following software outputs.

- A Java interpreter for multiple METATEM agents.
- A standalone Java API for creating, manipulating and evaluating temporal logic formulas.
- A graphical visualisation tool for observing, analysing and debugging multi-agent structures, at run-time.[9]

[9]This tool was developed by final-year student Michael Ceislar when he joined the project with the help of an EPSRC Vacation Bursary.

Finally, a number of research papers have been co-written, published and presented with fellow group members. This invaluable experience has also led to a number of peer-review requests, which have been gratefully accepted. In chronological order, these research activities and outputs were;

- Presented at the 9th European Agent Systems Summer School (EASSS'07).
- Hepple, A., Dennis, L. A. and Fisher, M. A Common Basis for Agent Organisation in BDI Languages. In *Proceedings of 1st International Workshop on Languages, Methodologies and Development Tools for Multi-agent Systems. Lecture Notes in Artificial Intelligence 5118*, pages 177–188, Springer 2008.
  - Chapter 4 contains the key points from this paper.
- Dennis, L. A., Hepple, A. and Fisher, M. Language Constructs for Multi-Agent Programming. In *Proceedings of 8th International Workshop on Computational Logic in Multi-Agent Systems (CLIMA). Lecture Notes in Artificial Intelligence 5056*, pages 137–156, Springer 2008.
  - Chapter 5 corresponds closely to the purpose and content of this paper.
- Presentation of the above at both the LADS workshop and the European Workshop on Multi-Agent Systems (EUMAS) in 2007.
- Peer-review for Journal Knowledge Engineering Review, Cambridge University Press.
- Peer-review for EASSS'09.
- Fisher, M., Dennis, L. A. and Hepple, A. *Modular Multi-Agent Design.* Department of Computer Science, University of Liverpool Technical Report number ULCS–09–002, January 2009.
- Peer-review for International Workshop, Programming Multi-Agent Systems (ProMAS).
- Peer-review for EASSS'10.
- Fisher, M. and Hepple, A. *Executing Logical Agent Specifications.* Chapter in *Multi-Agent Programming: Languages Platforms and Applications, Volume 2*, Springer 2009.
  - This publication has provided material for Chapter 3 and influenced other chapters.
- Fisher, M. and Hepple, A. *Executable Specifications for Pervasive Systems.* Submitted to Journal *ACM Transactions on Autonomous and Adaptive Systems.*

– This paper has been prepared using material from Chapters 6 and 7.

At the time of writing, more information, including copies of papers, downloads and links to external sources, can be found at

`http://www.csc.liv.ac.uk/~anthony` .

To clarify the aims of the thesis and the structure of this document the chapters contained herein can be summarised as follows. Chapters 1 and 2 discuss the underlying trends in computing that led to our adoption of a principled agent-oriented approach to the programming of context-sensitive applications. Chapter 2 continues, by describing some of the ways in which the concept of context has been interpreted, modelled and used, before giving further background information about some agent-oriented programming languages that are comparable to the one we adopt. Chapter 3 provides a detailed description of the METATEM language, its implementation created for this project and the language extensions which it provides, whilst Chapter 4 elaborates on these extensions, providing their formal semantics and demonstrating how they can be used to model concepts of agent organisation. Chapter 5 can be regarded as a complementary addendum to the thesis as it does not address the issue of context directly but argues for a common underlying basis for agent organisation across a certain category of agent programming languages. Chapters 6 and 7 demonstrate and evaluate our approach by undertaking two pervasive computing case-studies. Finally, Chapter 8 draws some conclusions and makes suggestion for the direction of future work.

Chapter 2

Background

2.1 Ubiquity

This chapter describes with more detail the nature of the emerging trend for ubiquity and mobile devices. It outlines the challenges faced by those who wish to develop and deploy novel applications and touches on some of the, already significant, body of work that has been published. Ubiquity and pervasive computing scenarios present many and varied challenges that will no doubt keep many researchers occupied for many years to come. This chapter defines the scope of the author's research within the wider field by clarifying which of these challenges it aims to contribute to.

2.1.1 Trends and drivers

The availability of small, inexpensive and low energy computing devices is one of many premises supporting the argument that we are on the verge of a significant shift in system engineering — if only driven by industry's inevitable desire to find applications for button sized (or smaller) devices [67].

Many novel applications of pervasive system technology have been proposed, including smart houses, assisted living, location awareness and calm technologies [130]. It is not only new applications that pervasive technology will bring, it promises to enhance the way we use conventional applications — reducing the system's dependency on our deliberate intervention and freeing us to concentrate on higher level activities. Groupware will benefit from the automatic entry of appointments, deduced from related activities such as telephone conversations. Point of sale equipment will be aware that to sell alcohol to a minor is illegal, and reject such a transaction. Information will be accessible on all manner of objects, presenting itself in a form and at a time appropriate to the user without prompting — step onto a train platform and a customer information panel seamlessly queries the itinerary held on a discrete device woven into your jacket, and displays the time of the next train to your destination.

2.1.2 Limits

As the distribution of computing devices within our every day environment approaches ubiquity and the scenarios mentioned in Section 1.3 become more common, so the key problems that must be overcome to realise the potential of ubiquitous computing become clearer. With clarity in the characterisation of problems comes an indication of their scale. Indeed, two of the original six Grand Challenges set by the UK Computing Research Committee[1] were directly related to ubiquitous computing. UK-UbiNet is an EPSRC funded project that has declared a manifesto that invites researchers to rise to the challenge [65].

Meeting the Grand Challenge requires a multi-disciplined effort and involves many aspects of computer science. Techniques are being developed to deal with aspects such as low energy wireless networking, sensor technologies and ad hoc networking, but it is two sub-projects of the UK-UbiNet project that this doctoral project complements; *Agent Technologies* and *Model-checking ubiquity* [66].

If, as is expected, developments in hardware provides us with the ability to create reliable low-power ad-hoc networks *and* such systems become socially acceptable then we still need to develop the means of specifying and expressing the behaviour we want (and don't want) from such systems. There are also interesting opportunities for novel forms of human-computer interaction and radically new user experiences, however it is the programming problem that is clearly the main barrier and so became the focus of this research.

Conventional approaches to engineering distributed software systems take a centralised view of the system, typically employing a client/server model to organise the components. In pervasive computing scenarios, as outlined above, a central authority will not be practical — the number of components (clients) is expected to increase beyond practical limits, components will be transient (joining, changing roles within, and leaving, a system in a highly dynamic fashion), continuous communication channels between components and a central authority cannot be relied upon and, crucially, components will require high levels of autonomy.

The programming problem is aggravated by the paradigms, languages and tools currently used. The predominant event driven, object oriented tool sets are not suitable for software that must be pro-active, adaptive, autonomous and context aware.

2.1.3 Challenges

In order to progress this project it was important to understand what characteristics are required of a pervasive systems programming language. This is the very question considered by Dobson and Nixon in [40] and [41], resulting in an informal list of require-

[1] They have since been merged into one. For more information on the grand challenges: `http://www.ukcrc.org.uk/grand_challenges/current/index.cfm`

ments. This section describes these requirements and relates them to the techniques used during the author's research.

Dobson and Nixon's "wish-list" [41] can be summarised as follows.

(a) **Events are too noisy to serve directly as a basis for programming.**
In dynamic, unpredictable environments of the kind that pervasive systems are expected to operate in, individual events can not be reliably used as the triggers for system behaviour. The significance, accuracy and volume of events make them unsuitable for describing system behaviour without considering other properties of the system.

(b) **Don't take anyone's word for anything.**
In systems composed of multiple autonomous entities, each with the ability to provide information (sensed or otherwise generated) about the environment, we must expect discrepancies to occur.

(c) **Interconnection is more important than data.**
Whilst the access to knowledge (sources) is important, it is essential that the reliability and significance of sources are considered. A disconnected component must depend upon its own source (sensor) for information, whilst a highly connected component should have some strategy for exploiting those connections and need not have its own source.

(d) **Any decision needs a mitigation strategy.**
Any action made by a system, in a dynamic environment without central authority, is made without certain knowledge of the system's state. Actions may produce undesirable environment states or unwanted side-effects which adaptive systems should be able to detect and mitigate the effects of.

(e) **Everything interesting comes from composition.**
Any framework for building distributed-intelligence must provide a well-founded basis for the composition of systems from independent components such that their interactions are well understood before deployment.

Let us examine each of these points separately.

Events are too noisy (a)

Individual events, such as the transition of a user from one zone to another, cannot be used in isolation to trigger system behaviour. For instance, in a meeting room scenario, it cannot be assumed a meeting is over if an individual sensor reports that an attendee has left the room — they may have hung their jacket (containing their identifier) on the back of the door or strayed into an area of sensor inaccuracy [41]. Instead, the

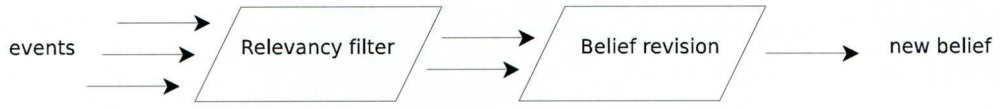


Figure 2.1: Events can be reduced to beliefs.

event must be considered against a broader context before a decision is taken to act. It is contexts and the properties of human reasoning that the agent-oriented paradigm to be introduced in Section 2.3 employs. Rather than being event driven, and risk being overrun by all minutiae of events, agents are driven by more abstract notions. In this abstraction systems behave according to their beliefs and changes in their beliefs, giving a stronger basis for acting than events alone. Beliefs are derived not only from sensor data but also as a result of reasoning with existing beliefs, the general concept is depicted in Figure 2.1. In practice the relevancy check might be performed by an agent acting on behalf of all the agents in a given context, protecting them from the disabling effects of excess communication.

In terms of logic, significant work [30] has been carried out on the extension of epistemic logic with attributes that indicate an agent's degree of certainty that a given belief is true. In this way beliefs arising from less relevant (perhaps more distant or less trusted) sources can be assigned lower probabilities and subsequently used to filter the noisy events.

Don't take anyone's word for anything (b)

This statement refers to the noise level expected in data from environmental sensors — high levels of noise means that atomic environment perceptions cannot always be trusted. Two distinct approaches to this problem exist; the *consensus estimate*, and assigning *trust values* to the sources of perceptions.

An agent-oriented approach is the closest to the latter. If considering a critical system behaviour that is dependent upon a percept which can be sensed by a number of sensors, a multi-agent approach might model the behaviour *and* each of the sensors as an agent. The levels of trust between data consumers and data producers would then be expressed by inter-agent relationships and constraints on inter-agent interactions.

Interconnection is more important than data (c)

Pervasive computing applications will not simply emerge, once our physical environment has been saturated with sensor devices. The data itself, without a model rich in relationships is not sufficient to allow applications to derive the enormous amounts of knowledge required of intelligent behaviour. Many model types have been proposed to, for example, reason about time, communication and security, however any pervasive

computing application is expected to involve many, if not all, of these aspects. Therefore any general pervasive computing platform will need to be flexible enough to allow the construction of a wide variety of models. If the abstract entity used to describe these systems is to be the 'agent' and systems are to be comprised of multiple agents, then the agents must be organised into appropriate structures.

In Chapter 4 the author demonstrates that by providing agents with appropriately flexible constructs, agents themselves can become become groups, collections or networks, indeed whatever structure is appropriate for the application.

Any decision needs a mitigation strategy (d)

In dynamic systems composed of asynchronously executing autonomous components it must be expected that an action made for a given purpose may bring about some unforeseen side-effect or simply not achieve the desired outcome. The ability to cope with such situations is one of the main aims of adaptive behaviour research, an area that agent-oriented techniques have contributed to. Agents have *goals* describing desirable system states and *plans* to achieve the goals. Unlike objects, which react to system states with the same action (method) each time they encounter that state, agents commonly have a number of plans for dealing with the same system state. If an agent's plan fails to achieve a goal it is able to try 'plan b'. Commonly an agent also has failure plans which apply when unable to achieve a goal, the purpose of the failure plan is often to 'revise' the goal and 'undo' any unwanted side-effects of its failed attempts.

Everything interesting comes from composition (e)

This statement may appear to be philosophically inspired but is also supported by the success of simplification techniques such as divide-and-conquer. Computing trends, both hardware and software have been following a component-built model for many years. However, where Dobson's and Nixon's concept of composition differs from the conventional interfacing of software components or the networking of personal computers, is its impromptu, flexible and yet reliable nature. Components are currently designed to fit narrow requirements and mostly require manual adaptation if those requirements change. Components in pervasive environments will be flexible enough to combine with a wider variety of other components, giving the potential for interesting composites. This requirement has many similarities with the goal of open agent societies, in which agents, unknown at compilation time, may enter and interact using open protocols.

2.2 Context

Perhaps not surprisingly, context appears to be a concept with numerous definitions. The dictionary-style definitions encountered during this work are listed here.

1. **The weaving together of words and sentences.**
   This literary meaning of context suggests that context is the collective meaning given by words and sentences such that this meaning is not conveyed by any individual word present.

2. **The connection or coherence between the parts of a discourse.**
   This meaning is more appropriate for the purposes of agent interaction but is perhaps too general to be a useful definition.

3. **The body of information that is presumed to be available to the participants of a speech situation.**
   This definition suggests that context is largely factual in nature and static for a given interaction.

4. **The physical and social situation in which interacting entities find themselves.**
   In contrast to definition 3 above, this definition defines context as the situation, as opposed to the knowledge inherent in the situation.

5. **The discourse that surrounds a language unit and helps to determine its interpretation.**
   From Linguistics, this definition highlights the effect context can have on the meaning of an interaction.

Clearly some of the above definitions are more relevant to this work than others but each add to our understanding of the concept. Naturally, there have been attempts from within the computing community, to define context. We can add the two prominent ones to our list:

6. **Where you are, who you are with and what resources are nearby.**
   Schilit, Adams and Want, in [115], informally describe context this way. It is the author's opinion that this overstates the influence of location and physical surroundings which, although often significant, are not necessarily so. Also, it seems sensible to add *"what you are doing"* to this description.

7. **Any information that can be used to characterize the situation of an entity.**
   This popular definition of context from Dey [36] goes on to describe an entity as *"a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves"*.

It was felt necessary to define context for the scope of this project in order to give clarity to this work. After consideration of the above definitions and the project objectives the following definition was assumed:

Context: The connection or association between the participants of an interaction and any characterising information.

2.2.1 Early experiments

The Xerox PARC laboratory originated the idea of Ubiquitous Computing based upon the belief that by exploiting contextual information such as the spatial arrangement of devices and users, and novel modes of interaction, that future computer systems may be able to provide more valuable services collectively, than they can individually [127]. The PARCTAB, a handheld, tablet shaped, wireless device with touch sensitive screen, formed the centre-piece of the laboratory's pioneering work in the early 1990s. The PARCTAB experiment provided valuable insight into many of the problems that face ubiquitous computing researchers today. Problems that one might expect such as user-interface design and power management, were indeed encountered but were not regarded as critical. Rather, in [127] they state that

> "Creating such an intuitive and distributed system [as envisioned by Weiser in [130]] requires two key ingredients: communication and context."

The PARCTAB project demonstrated an early example of technologies that have since become widespread, such as the incremental search techniques exemplified by the T9™ predictive text feature of mobile telephones.

The architecture of the PARCTAB system was, of course, influenced by hardware limitations at the time. Consequently the hand-held devices, equipped with little processing power or memory in comparison to equivalently sized devices today, were little more than "thin clients" for applications that resided elsewhere on the network. Partly as a consequence of the limited mobile resources but also by design choice, each mobile device was partnered with a software agent. These agents were responsible for forwarding events between the tablet and applications but also for providing applications with information about the context of a tablet and its user. Significantly, the terms *context* and *agent* came together in this very first Ubiquitous Computing experiment. Examples of contextual information managed by the agents included location, the proximity of other devices, the presence of people, time, and the state of the network file system.

While some context-aware applications were developed [115], the use of context was relatively conservative. Context was often limited to modification of menu choices or simply presentation of context information so that the user could make more informed decisions, for example about whether or not to disturb a colleague.

Language choice

The developers of PARCTAB did consider the suitability of available programming languages before embarking on their experiment but did not have the luxury of suitable special-purpose languages. Instead they chose, what at the time, was a state-of-the-art general purpose object-oriented language — Modula-3. A choice which, on evaluation, proved unpopular with developers. Developers, they concluded, preferred a choice of languages with familiar constructs. These conclusions support the argument that any language innovations must provide marked benefits (as opposed to incremental ones) if they are to be adopted.

Overall, the PARCTAB experiment was an important landmark in context-aware computing as it generated interest and ideas in the area, and influenced some commercial products on the market today.

2.2.2 The handling of context

In a branch of linguistics that studies the use of language in social interactions and the semantic contributions that context provides, called pragmatics, philosophers attempt to understand how ambiguities of language are overcome by employing a number of non-linguistic concepts such as presupposed knowledge, environmental circumstances and speaker intent. The study of pragmatics is motivated by the many ambiguities that sentences presented in isolation can contain. For example, consider the sentence: "You are heading in the right direction." Presupposed knowledge held by both the speaker and the spoken-to may include an absolute direction (e.g. north, south, east or west) and a target destination, in the context of spatial directions. Or, in a different context, presupposed knowledge may include a mutually understood goal or solution to a problem, and an activity that the addressee is currently undertaking to achieve that goal. Context then, as defined by those studying pragmatics, is the body of presupposed information that fixes the meaning of discourse. One challenge that is common to both computer scientists and linguists but perhaps more acute for those computer scientists interested in automated reasoning, is the identification of knowledge and its assignation as common-knowledge.[2] Even with respect to information systems the nature of common-knowledge will vary dramatically between applications. For instance, environmental circumstances will differ for an auction-based trading platform and a medical monitoring system.

In [120] the author argues that the linguistic aim of any speech act is an attempt to modify the body of presupposed information. The theory of agency has been strongly influenced by the work of linguists and particularly by the theory of speech acts [116].

[2]Similar to the concept of presupposed-knowledge, the concept of common-knowledge is used in the field of knowledge representation, a sub-field of computer science. For the purposes of comparison, one can think of common-knowledge as explicitly shared presupposed knowledge within a monotonic reasoning framework.

To the extent that the Foundation for Intelligent and Physical Agents (FIPA) have defined a standard Agent Communication Language based upon the theory of speech acts. The relationship between the semantics of agent communication and the handling/modelling of context is clearly a intimate one.

Attempts to formalise context understandably vary in their treatment of and approach to context, based upon a their chosen definitions of context. This section aims to describe the variety of contributions from both philosophers and computer scientists, hence describing the author's view of the state-of-the-art in contextual modelling and reasoning.

A pragmatic view of context

In [119], and concerning the study of speech acts in a social context, the author proposes a model in which context is treated as a first-class concept and combined with Kripke semantics of possible worlds to produce a semantics that interprets propositions from speech-acts and context, and subsequently truth values from possible-worlds. Simplistically, a speech-act combines with a context to give a proposition:

$$speech\_act \times context \mapsto prop$$

and truth values are obtained with an interpretation function $\mathcal{I}$ with Kripke semantics:

$$\mathcal{I}(prop) \mapsto \{true, false\}.$$

Such a treatment of context could be viewed as the extraction of information from $\mathcal{I}$, whereas a simpler system might involve a direct mapping from speech-acts to propositions and a correspondingly expanded model of possible worlds within $\mathcal{I}$. However, the use of contextual knowledge in the derivation of propositions recognises the subtle but real difference between presupposed/contextual and the rest of the knowledge used for any given computation.

2.2.3 Context modelling

Such is the general consensus that context will perform a key part in the long term transformation of software engineering, that a significant amount of effort has been, and is being, channelled towards the modelling of context. Here we refer to the modelling of context in isolation from (but with regard for) software and program code. In [8] Bettini et al identify some requirements from their ideal context modelling technique. It is not surprising that the list of requirements bears striking comparisons with the requirements of an ideal pervasive computing programming framework, as discussed on page 15 of this thesis, and serves to reaffirm context's place in the programming of pervasive systems. Bettini et al require that an ideal context modelling technique be able to:

- deal with **heterogeneity and mobility** with respect to contextual information sources;
- express **relationships and dependencies** between types of context information;
- express the **timeliness** of context information, such that past, present and future context information is identifiable;
- express the quality of contextual information due to the likeliness of **imperfection**;
- support **reasoning** with the modelled data;
- provide **usable modelling formalisms** to facilitate both real-world modelling and manipulation by software applications; and
- support **efficient context provisioning** such that even in the presence of massive amounts of context information, relevant information can be readily identified and accessed.

No single technique claims to satisfy all of these requirements. The Context Toolkit [37] enjoyed early prominence with its concept of observable context 'widgets' and in [26] Coutaz and Rey extended these concepts with their Theory of Contextors, that involved mapping relationships between context-providing 'observables'. Their work makes a deliberate distinction between the concepts of a situation and a context; the former being described as a temporal snapshot of observables and the latter being composed of the changes in observables over time. However, as this field of research has developed three other approaches have emerged, each of which are currently being actively pursued, namely object-role based, spatial models and ontology based. The remainder of this section briefly describes an example of each of these approaches and their respective advantages.

Object-role based

Object-role modelling is a technique that arose from database modelling which allows formal description of concepts via intuitive languages and diagrams. A prominent form of object-based modelling of context is the Context Modelling Language (CML), described in [73]. CML uses an XML based language to describe the detail of contextual concepts in detail but also uses a form of predicate logic to reason about higher-level context abstractions called 'situations'. The formality of CML and its semantics enable the context information to be fed to fact-based reasoning engines. Strengths of object-role modelling include its support for graphical modelling and the fact that it is based upon a mature modelling technique which is familiar to a broad base of developers.

Spatial models

Physical location is an essential feature of context in many context-aware applications. Spatial models often tend to be fact-based models, such as object-role, that include a location fact and in which entities are structured according to location. Location can be symbolic or can be based upon a coordinate system. Spatial models effectively elevate location to the primary context; thus applications whose primary concern is location can benefit from efficient context provisioning due to the complementary way in which data is structured and the potential for pruning the search space based upon location. An example of this type of spatial models is the Augmented World Model [97].

Ontology based models

Ontology based models consider context to be knowledge, supporting context definition number 3 above. Hence the popularity of OWL-DL which, as a carefully chosen subset of OWL and a description logic, offers complete reasoning and is well supported by automated reasoning tools. Furthermore, ontologies are ideally suited for sharing context information in pervasive and non-pervasive computing environments. The SOUPA [18] ontology has been proposed specifically for pervasive computing applications.

In summary, these approaches to handling context consider context as an extraneous information type which is queried and may be held centrally. In contrast, and like the literary definition number 1 above, we consider it to be woven into the system's model.

Other techniques

In addition to the models described above, which directly tackle context, there are other modelling techniques that provide an appealing means of expressing and modelling context even though this is not their primary purpose. Milner's Bigraphical model [83] is a nice example of such a technique. Bigraphs are an attempt to combine a spacial model of entities, with a model of connectivity, in a rigourous algebraic way but also providing an appealing graphical representation. The graphs provide an intuitive way for humans to model and analyse the spacial relationships and communication connections that exist between autonomous entities in a system, whilst the algebra allows the graphs to be transformed in a consistent way. In this approach a graph (or sub-graph) can be viewed as a context for the activity of a node in the graph. As we will see, bigraphs share some similarities to the modelling approach adopted by this thesis. However, bigraphs are necessarily restricted to the semantics of locality and connectivity whereas our approach aims to have more general application.

2.2.4 Programming frameworks

Having reached a consensus that context, as a concept, is vitally useful when describing the behaviour of entities in intelligent and distributed computing environments, an

understandable reaction is to modify existing programming frameworks in order to accommodate context as a first-class entity. Although this approach is not the one taken by this work, the evaluation of and comparison with, such an approach was important to the completion of this project. This section describes, amongst others, the Java Context Awareness Framework, a mature extension of the popular object-oriented programming language.

Extending Java with context

The Context Toolkit [37] was one of the first attempts at a middleware for storing and disseminating context data that allowed context-sensitive applications to built without this overhead. The Context Toolkit provides simple context 'widgets' and an API for distributing and accessing them in a networked environment. Although conceptually useful [96], the Context Toolkit did not employ popular software development standards and currently does not appear to be maintained.

The Java Context Awareness Framework (JCAF) [3] arose from the need for a generic programming framework that supports the programming of context-aware applications with conventional languages. It aims to be a light-weight set of interfaces that allows the expression of context-based events, communications and actions with constructs from the Java programming language and uses well-known patterns such as the event model. JCAF facilitates the creation of distributed service-oriented and event-based applications and does not target any specific application domain. It provides a Run-time Infrastructure that gathers together a number of context information processing services, and an Application Programming Interface (API) for developing the context-consuming clients [3].

JCAF applications are able to make requests for contextual data on a client-server basis or subscribe to relevant context events. Sharing of sensitive data has been considered and basic access control implemented. JCAF upholds the worthy design principle of semantic-free modelling abstraction with respect to context, however this is easily achieved by allowing context data to be any Java object that implements the `ContextEntity` interface. Interestingly, JCAF allows quality measures to be assigned to context information, indicating for example, its age, accuracy or the dependability of its source. A context-providing service maintains contextual information for a distinct environment; when an application's domain contains multiple distinct environments, these services are networked in a peer-to-peer topology and able to query one another, presumably to locate information they have been asked for but do not possess.

Within each context-providing service runs an entity component corresponding to a context-sensitive entity in the application domain and in turn, each entity holds any number of context data items. Each service entity is a Java process that is responsible for monitoring and responding to changes of context in the application entity. This

is achieved with notions of context monitors and context actuators, each of which is represented by a Java interface.

Interestingly, some similarities exist between the context entities of the JCAF runtime and the way in which contextually related agents are structured in the proposals detailed by this thesis. Related entities are held in a container, this container controls the life-cycle of entities it contains. Also, entity components are described as 'working together to achieve their tasks', although it is not suggested that any level of agent-like co-operation is occurring, merely that an object method interface is held by each. Furthermore, a similar recursive relationship exists between context entities and context items, whereby entities are themselves context items. Thus, an entity can comprise of further entities.

During evaluation of this framework a context-aware hospital bed application was investigated, here examples of context entities included patients, places, beds, medicines and monitoring equipment. An entity container corresponds to a hospital and examples of context data items are location (e.g. coordinates), a patient's name, and treatment activity [2]. Other applications investigated include proximity based user authentication and a novel information system for informing collaborating workers of their colleagues' current activities [4].

Whilst JCAF does not attempt to address the problem of finding novel programming constructs appropriate for programming context awareness, nor does it attempt to provide a means for any automated verification of system properties or satisfaction of requirements, it does achieve a credible infrastructure which could feasibly be deployed across an organisation's existing network, upon which a variety of applications could access, share, generate and modify contextual data.

2.2.5 Logical formulations of context

This section discusses context in more depth. In particular, the popularity of context as a concept for supporting reasoning and capturing general intelligence. We summarise important attempts to formalize context, for example by McCarthy [92].

Context in AI

Perhaps the first significant proposal for a logical formalisation of context, whereby context is explicitly expressed in the syntax of a language of logic, came from McCarthy, Guha and Lenat in [92, 71]. Their aim was to enhance AI logic with qualities of human reasoning, such as context-based assumptions and the results were incorporated into the Cyc common-sense database [70]. In attempting this aim they introduced a context object and a relation ist, such that $ist(c, p)$ asserts that proposition p is true in context c, for the purpose of expressing common-sense in the form of context-dependent axioms and thus allowing state (within an explicit context) to be expressed more concisely with

fewer assumptions. This formalisation requires **all** formulas to be asserted within a context, including the *ist* relations, hence a hierarchical relationship between contexts exists which can be harmlessly infinite [92]. The authors realised that time is a frequent attribute of context, but rather than employ modal operators in the language, they chose to describe time and other attributes of context as a term in a function which specialises a context, i.e.

$$ist(specialize\text{-}time(t, c), p)$$

which says that proposition p is true at time t in context c. Clearly the proposed language was not designed for dynamic reasoning, however its authors did consider the acts of entering and leaving context, basing their proposals on the relation $ist(c, p)$ having the meaning $c \Rightarrow p$.

Despite its relative obscurity today, this work provides significant background to this thesis as it highlights some of the difficulties that present themselves when trying to apply deductive techniques to aspects of human reasoning such as context awareness.

Situation calculus

Situation calculus provides a calculus for representing and reasoning about change. It describes the world in terms of *fluents (properties whose values are subject to possible change),* actions that can be performed and *situations* that arise from a sequence of actions. Changes to the world are modelled as a sequence of situations leading to time t, where each situation appends actions to the previous situation and each situation encodes a complete history of actions from time $t = 0$.

The language is based upon second-order predicate logic with three term types: actions, AC, situations, S, and objects, O, where each type can be represented by constant or variable symbols. The special constant `s0` denotes the initial situation and the function do is of the form

$$do : AC \times S \mapsto S$$

hence $do(a, s0)$ denotes the situation resulting from performing the action a in the initial situation s, and $do(b, do(a, s0))$ denotes the subsequent situation following action b. Thus directly encoded within a situation is the sequence of actions that characterise it. Fluents can be predicates or functions with arity of one or more and in each case the final argument is a situation. The value of predicate fluents are specified by domain axioms known as effect axioms that state in which situations and after actions, fluents change their values. Other domain axioms, called "action precondition axioms", specify when (in which situations) an action is valid or invalid

$$Poss : AC \times S \mapsto \{true, false\}.$$

Foundational and domain axioms combined enable the value of fluents to be calculated for any situation. This valuation of fluents is akin to the interpretation of a possible

world in Kripke semantics but the situations within situation calculus are unique for any system and have only one temporal occurrence — a situation cannot be revisited.

Though the situation calculus does not aim to address the context and does not provide an explicit construct for context, it does encodes a verbose form of context by providing, for each situation, a complete history of events that led to that situation.

GOLOG

Clearly the situation calculus in its pure form is impractical for the programming of reactive non-terminating systems with infinite execution time, as this requires situations to have ever increasing, potentially infinite, lengths. Nevertheless, the GOLOG programming language was developed from the theory of situation calculus and, like METATEM, it too is a specification language which aims to capture a high-level of behavioural abstraction.

Based upon the situation calculus but also extending it, GOLOG provides an interpreter for specifications that attempts to generate a sequence of actions which satisfy the specification. Among the initial extensions to situation calculus were procedural declarations and a non-determinism operator [90]. Perhaps the most significant practical difference between this GOLOG interpreter and the METATEM interpreter that was implemented to support this thesis, is the static nature of a GOLOG interpretation. That is, a sequence of actions are determined ahead of execution, whereas the METATEM interpreter is able to react to run-time events and modify its future execution accordingly. The initial GOLOG language lends itself to implementation by Prolog and multiple implementations exist, including an agent-oriented concurrent version called CONGOLOG [21]. Finally, INDIGOLOG [32] introduces the possibility of reactive behaviour by interleaving the execution of actions, allowing 'agents' to sense and react at run-time.

Further descriptions of GOLOG, including its implementation and application, are provided in Section 2.5 where a survey of respected agent languages is provided.

2.3 Agent concepts

The nature of hardware and software platforms is changing rapidly. There are trends of increased distribution, openness and mobility, increasing modes of connectivity and communication, and novel human-computer interactions. This increased sophistication provides the opportunity to develop software for ever more complex scenarios. Scenarios in which software applications are expected to cope with unpredictable environments, act with increased autonomy and adapt to the users' changing context of use with minimal explicit user intervention, and which are themselves distributed over many hardware components.

This work is concerned with the development of a programming framework for applications to be deployed in such complex environments. The popular agent metaphor of autonomous entities acting and sensing in some environment has been adopted due to the ability of agents to act independently, to react to unexpected situations and to co-operate with other agents, making it a natural choice. However, the agent abstraction is often modelled as isolated pairs of agent and environment, with each agent being aware of other agents only by sensing changes in their environment brought about by the actions of those other agents. Complexity is introduced by the presence of other autonomous entities, by restrictions on time and memory resources and by incomplete, heterogeneous or contradictory, information; indeed by anything that introduces environmental dynamics. Hence, many believe that a further abstraction is required to capture the complex relationships between an agent, the environment and the other agents in that environment [98]. Thus, this section covers the background of agency relevant to this work, with a particular emphasis on abstractions for organising multiple agents.

The term 'agent' has many and varied uses, including a category of software application, an extension of an object and an intelligent mobile entity. For the purposes of this work an agent is characterised as an autonomous software component having certain goals and being able to communicate with other agents in order to accomplish these goals [135]. The key reason why an agent-based approach is advantageous for modelling and programming autonomous systems, is that it permits the clear and concise representation, not just of *what* the autonomous components within the system do, but *why* they do it. This allows us to abstract away from low-level control aspects and to concentrate on the key feature of autonomy, namely the goals the component has and the choices it makes towards achieving its goals. Dennett's theory of Intentionality [34] provided the underpinning philosophy for modelling a system in terms of agents, where each agent is ascribed beliefs and goals, which in turn determine the agent's intentions. Such agents then make decisions about what action to perform, given their beliefs and goals/intentions. This kind of approach has been popularised through the influential BDI (Belief-Desire-Intention) model of agent-based systems [110] and a number

of programming languages based upon this model have been developed [15]. Across these languages, some agreement on the core attributes of an individual BDI agent has emerged.

Agents have *beliefs*, with which they can represent their environment and other agents. They are just beliefs, they may not be true now or at any time in the past or present, but are considered to be the agent's best understanding with its available information. In BDI agent languages, beliefs are often formalised with a modal logic of belief [110, 111, 112]. Agents are driven to act autonomously by a motivational aspect that also influences their action when faced with a choice. Commonly this motivation is represented by *goals* which encode a desirable (for the agent) environmental state. To achieve its goals without aimless random behaviour, an agent has *plans* which aim, but do not guarantee, to achieve goal states.

2.4 Agent-oriented design methodologies

Although the scope of this thesis does not extend to methods of agent-oriented system design, a possible avenue of future work (described in Section 8.1) does. For this reason it is worth noting some of the significant design methodologies proposed. The methodologies Prometheus, Gaia and Tropos[3] stand out amongst a small number of proposals, due to the authority of their authorship and/or their popularity of use.

Prometheus [100] has been developed with collaboration from the vendors of JACK (described in the next section) and aims to provide a comprehensive process for the specification, design and implementation of agent-oriented systems. Prometheus is a practical methodology aimed at student and industrial audiences and is well supported by development tools. In contrast Gaia [136], taking its inspiration (and name) from Lovelock's popular view of the earth's eco-system, takes a more theoretical view of multi-agent design. Gaia applies software engineering principles to the design of both the agent and the society of agents. Gaia allows the specification of liveness and safety properties, commonly associated with temporal-logic based specifications. Tropos [16] combines knowledge engineering and software engineering principles, paying particular attention to the analysis of human-agent interactions and thus encouraging a good understanding of the problem domain.

2.5 Agent languages

As mentioned, a number of agent languages have been inspired by the BDI architecture, some of which are also formalised using logical languages. Most have inspired implementations in executable programming languages. This section surveys a number

[3]The Greek origin of each of methodology name is believed to be coincidental.

of respected implementations, indicating their logical credentials, their prominent features and intended applications. The reader can find more detailed information in [15] and [14].

2.5.1 dMARS

The Distributed Multi-agent Reasoning System (dMARS) language is a true successor to the Procedural Reasoning System [63] — the forefather of all BDI-based programming languages. Whilst not having a logical basis these systems are significant because of their close adherence to the BDI architecture and their robust commercially accepted implementations. A dMARS agent has four main data structures: a plan library; a belief base; an event queue; and an intention stack. Deliberative behaviour is described by a number of selection functions;

An **event** selection function, responsible for selecting an event to respond to.

A **plan** selection function, responsible for selecting appropriate plans with respect to selected events, current intentions and the belief base.

An **intention** selection function, responsible for identifying which of the identified intentions to act upon.

Plans comprise of *trigger* event, a *context* which serves to modify the plan to suit the agent's circumstances, and a *body* of actions [112]. Although dMARS has reached the end of its maintenance period, it remains one of the most successful agent systems built, being used for many significant commercial applications such as control systems, supply chain management and air traffic control [64].

2.5.2 GOLOG

GOLOG is an extension of McCarthy's situation calculus (see Section 2.2.5) in which agent programs are specified as situations. Recall that a situation in situation calculus is a list of all actions that have led to it, as opposed to a list of properties which characterise the situation. However, instead of explicitly stating an agent's entire execution, a GOLOG specification may contain *sequencing*(;), *test*, *iteration* and *non-deterministic choice*(|) of actions, and of other GOLOG programs (procedures). Thus, the following program

Proc $serve(n)$ $go\_floor(n)$; $turnoff(n)$; $open$; $close$ endProc.

Proc $go\_floor(n)$ $(current - floor = n)$? | $up(n)$ | $down(n)$ endProc.

Proc $serve\text{-}a\text{-}floor(\pi n)$ $[next\_floor(n)$? ; $serve(n)]$ endProc.

Proc $control$ [**while** $(\exists n)on(n)$ **do** $serve\_a\_floor$ **endWhile**] ; $park$ endProc.

Proc $park$ **if** $current\_floor = 0$ **then** $open$ **else** $down(O)$; $open$ **endIf** endProc.

from [90], in conjunction with the definition of domain fluents and axioms, describes the behaviour of a lift. Execution is a form of theorem proving, akin to that of Prolog, whereby a sequence of primitive actions is attempted and variables instantiated. A successful execution produces a satisfying sequence of (grounded) actions. Concurrent GOLOG (CONGOLOG) is an extension in which the execution of multiple GOLOG processes are interleaved and where fine-grained control of the interleaving is possible [31]. GOLOG has a popular following with robotics enthusiasts and cognitive robotics researchers, following another extension, LEGOLOG [89].

2.5.3 AgentSpeak

AgentSpeak, was proposed by Rao in [109] as a logically sound variant of the BDI architecture popularised by the Procedural Reasoning System (PRS) and dMARS. AgentSpeak was given an operational semantics using a restricted first order modal logic. Retaining concepts of events, actions and a belief-base this language drew significant attention which, in turn, generated several implementations [11, 91, 132]. Of these implementations, the Jason interpreter is the most developed [13]. In addition to the BDI features of AgentSpeak, Jason adds inter-agent communication based upon speech act theory, physical distribution of agents and a framework for designing agent organisations [12]. The syntax used by Jason is reminiscent of Prolog, with horn-clause heads representing goals and events, whilst the body represents a plan. Clauses can be both guarded and annotated, allowing significant flexibility during plan selection and event handling. The following Jason code, provided with the Jason download, illustrates how a 'cleaning' robot might react to the event of new knowledge, i.e. finding garbage.

```
+garbage(r1) : checking(slots)
   <- !stop(check);
      !take(garb,r2);
      !continue(check).
```

If this event is selected, and providing the robot is currently checking for garbage (`checking(slots)` is in its belief-base), then the agent/robot will attempt the body of the plan. In this case involving three sub-goals. The second example below, illustrates a plan for achieving a newly adopted goal.

```
+!stop(check) : true
   <- ?pos(r1,X,Y);
      +pos(back,X,Y);
      -checking(slots).
```

This plan is always relevant as the guard is `true` and the body involves querying the belief-base (`?`) and adding (`+`), and removing (`-`), beliefs to/from the belief-base.

AgentSpeak has neither targeted nor attracted commercial applications. However, it has a growing academic following, encouraged largely by the well-maintained Jason interpreter.

2.5.4 3APL

3APL, pronounced 'triple-a-p-l', also follows the BDI architecture faithfully. It explicitly caters for concepts of *belief, capabilities, goals* and *plans.* Beliefs, goals and (additionally) rules are declared as horn-clauses, whilst capabilities correspond to the fluent axioms of situation calculus. A key feature of 3APL is its programmable deliberation cycle which uses a meta-language to allow revision of goals and custom ordering of goals [27]. The well maintained implementation from researchers at the University of Utrecht allows agent communication and distribution, as well as the ability to execute arbitrary Java code. 3APL does not target any specific application area and has (to the author's knowlege) not been applied industrially, it does however have a wide academic user-base.

2.5.5 JACK

JACK is another language inspired by PRS and dMARS into adopting the BDI architecture. In fact, JACK is a commercial spin-off from the Australian Artificial Intelligence Institute, who devised and developed both PRS and dMARS. As one might expect from a programming language that targets commercial applications, JACK supports the developer with tools such as an IDE and graphical plan development. JACK's syntax is a conservative extension of Java syntax, which not only enables agent code to be translated into pure Java, but also makes the language accessible to existing Java developers. However, it does mean that providing formal semantics for JACK is not possible at this time. Communication between JACK agents is not limited to plain messages, they can also share capabilities by sending 'bundles' of plans. Extensions to JACK support FIPA communication and agent teamwork. Interestingly, the JACK Teams extension adds the concepts of *team* and *role* in a way that models a team as a specialisation of an agent, retaining all the agent concepts of plans, capabilities and beliefs, etc. but allowing a 'team agent' to have *sub-teams* [133]. As we will see later, this is similar to the approach to teamwork taken by METATEM.

2.5.6 Jadex

Jadex is the final agent programming language surveyed to be faithful to the PRS example. One of its motivating aims is to make agent-oriented programming easier and more accessible. To this end, beliefs are given an object-oriented representation,

such that the belief-base consists of sets of $(name, object)$ pairings and can be queried by a set-oriented query language. Goals too, are explicitly stored in a queryable goal-base, a feature not present in most BDI style agent languages without a meta-level reasoning extensions. Agents have a static definition by means of an XML file, whilst plans are defined in pure Java [103]. No formal semantics for Jadex has been defined. The language is well developed and supported via an open-source community project.

2.5.7 JIAC

JIAC (Java Intelligent Agents Componentware) does not have logical foundations and deviates further from the BDI architecture than the other languages mentioned. The JIAC framework places emphasis on satisfying industrial requirements such as complying with software standards, security and scalability. It combines the academic agent-oriented paradigm with industy standard service-oriented techniques and is mentioned here due to its success in recent agent programming competitions [28] and industrial funding [78].

2.6 Agent organisation

This section summarises some of the popular and diverse approaches to agent organisation that have been proposed, as an introduction to the agent-organisation adopted by this work, a more detailed description of which appears in Chapter 4.

Joint intentions With a respected philosophical view on agent co-operation, Cohen and Levesque produced a significant paper 'Teamwork' [24] extending previous work [88, 22, 23]. They persuasively argue that a team of agents should *not* be modelled as an aggregate agent and propose new (logical) concepts of *joint intentions*, *joint commitments* and *joint persistent goals* to ensure that teamwork does not break down due to any divergence of individual team members' beliefs or intentions. The authors' proposals oblige agents working in a team to retain team goals until it is mutually agreed amongst team members that a goal has now been achieved, is no longer relevant, or is impossible. This level of commitment is stronger than an agent's commitment to its individual goals which are dropped the moment it (individually) believes they are satisfied. Joint intentions can be reduced to individual intentions if supplemented with mutual beliefs.

Teams Tidhar [124] introduced the concept of *team-oriented programming* with social structure. Essentially this is an agent-centred approach that defines joint goals and intentions for teams but stops short of forcing individual team members to adopt those goals and intentions. An attempt to clarify the definition of a 'team' and what team formation entails is made using concepts such as 'mind-set synchronisation' and 'role

assignment'. Team behaviour is defined by a temporal ordering of plans which guide (but do not constrain) agent behaviour. A social structure is proposed by the creation of *command* and *control* teams which assign roles, identify sub-teams and permit inter-team relationships. In [17], the authors formalise their ideas of social structure with concepts of commitment expressed using modal logic. This allows the formal expression of commitment between teams, such as

team A intends to achieve task B for the sake of team C.

Pynadath *et al.* [108] describe their interpretation of team-oriented programming that aims to organise groups of heterogeneous agents to achieve team goals. A framework for defining teams is given that provides the following concepts:

Team—an agent without domain abilities;

Team-ready—agents with domain abilities that interface with team agents;

Sub-goal—a goal that contributes to the team goal; and

Task—the allocation of a sub-goal to a team-ready agent.

An implementation of their framework, TEAMCORE, provides organisational functionality such as multicast communication between agents, assigning tasks, maintaining group beliefs and maintaining hierarchies of agents (by role). Heterogeneous agents are accommodated by wrapper agents that act as proxies for the domain agent.

Roles Ferber *et al.* [44] present the case for an organisational-centred approach to the design and engineering of complex multi-agent systems. They cite disadvantages of the predominant agent-centred approaches such as: lack of access rights control; inability to accommodate heterogeneous agents; and inappropriate abstraction for describing organisational scenarios. The authors propose a model for designing language independent multi-agent systems in terms of *agents*, *roles* and *groups*. Agents and groups are proposed as distinct first class entities although it is suggested that an agent ought to be able to transform itself into a group. (We will see later that this is close to our approach.)

In [45], Ferber continues to argue for an organisational-centred approach, advocating the complete omission of mental states at the organisational level, defining an organisation of agents in terms of its capabilities, constraints, roles, group tasks and interaction protocols. Clearly articulated here is a manifesto of design principles.

Hübner *et al.* believed that the agent organisational frameworks proposed prior to their 2002 paper [81] overlooked the significant relationship between structural and functional properties of an organisation. Thus, in [81], they propose a three component approach to the specification of agent organisations that combines independent

structural and functional specifications with a deontic specification, the latter defining among other things the roles (structural) having permission to carry out group tasks (functional). The approach provides a proliferation of constructs for specifying multi-agent systems, including the ability to concisely express many additional aspects, such as

- the ability to specify *compatibility* of group membership, akin to the members of a government expressing a conflict of interest.
- enabling the *cardinality* of group membership to be defined and thus defining a well formed group as a group who's membership is between its specified minimum and maximum size.
- control of the organisation's goal(s), with an ability to specify sequential, branching and parallel execution of sub-goals.
- the ability to express a variance in the agents' permissions over time.

It is argued that such an approach improves the efficiency of multi-agent systems by focusing agents on the organisation's goals. Indeed, we note that of all the proposals discussed in this section this approach provides the developer with the widest vocabulary with which to express agent behaviour when defining the organisation.

Institutions Esteva, Sierra *et al.* have made formal [43] and practical [42, 126] contributions to this method of agent organisation that enjoys much current popularity [98]. An electronic institution aims to provide an *open* framework in which agents can contribute to the goals of *society* without sacrificing its own self-interest; the implication being that an autonomous agent will be motivated to participate in the institution by its desire to satisfy it own goals, but that its participation will be structured by the framework in such a way that institutional goals are achieved. A key concept is that of institutional norms.

In [43], the institution remains independent of agent-architecture by modelling agents as roles, of which there are two types — internal and external (to the institution) — with different rights. A *dialogue* defines valid locutions, a *scene* is a unit of interaction within an institution and a *performative structure* defines an objective as a network of scenes. In an attempt to allow more agent autonomy these ideas were refined and in [126] more concepts were introduced, including *landmarks* that can be used to guide agents through an interaction when a prescriptive dialogue is considered too constraining.

Perhaps the most noteworthy aspect of these proposals is the change of focus from the agents themselves onto the interactions that take place between agents. In recognition that in an open multi-agent system, it may not be possible to verify the internal computation of an individual agent, only its interactions with other agents.

Summary

It should be noted that none of the above organisational approaches can comprehensively model all forms of co-operative multi-agent systems. Rather they represent attempts to discover practical and beneficial ways of specifying distributed computational systems, and facilitating the focus of computation on a system's main purpose whilst not compromising the autonomy of the system's components. In achieving this aim it may be convenient to categorise groups of agents in terms of cohesion and co-operation. For instance, a *group* of agents may be individually autonomous, existing as a group solely due to their proximity to one another rather than their co-operation. In contrast, the word *team*, implies a high degree of co-operation and adhesion with an *organisation* fitting somewhere in between. As Cohen stated in [24]

> "teamwork is more than co-ordinated individual behaviour".

Thus, the more expressive proposals reviewed here enable the specification of more cohesive groups but often at significant cost to the agents involved.

Chapter 3

METATEM

This chapter describes the foundational theory of a formal temporal specification and execution language, and the subsequent incremental additions to the theory that this work and others has made. The language in question is inspired by automated formal verification techniques, temporal logics and concepts of agency. Coined METATEM, due to its aim of providing a meta-level programming facility and its use of temporal logic, this allows high-level temporal specifications of a system to be directly executed [6]. Thus, conventional error-prone techniques for the translation of specification to lower-level executable code can be avoided. Latterly known as CONCURRENT METATEM, due to the support of multiple asynchronously-executing specifications, it has evolved over a number of years of research and experimental implementations [49, 51, 60, 48, 57, 52].

The chapter begins by putting forward the case for a declarative approach to programming in general, but particularly for the applications considered by this project. It introduces the fundamentals of the temporal logic utilised, then goes on to describe the basic temporal semantics of METATEM. The implementation developed in support of this thesis is then described, including illustrative examples of its syntax and use, before a larger complete example brings the chapter to a close.

3.1 Declarative agents

Many programming languages or frameworks based on the idea of describing how an agent behaves in different situations[1] have been proposed (some of which are discussed in Section 2.5) but these languages/frameworks have typically involved descriptions of the form

if in *Situation1* **then** do...

if in *Situation2* **then** do...

... and so on ...

[1]The term situation here is used in a general sense to refer to the combination of an agents internal attributes (beliefs, goals, etc.) and its external influences such as perceptions of its environment.

These explicit, ahead of time descriptions illustrate the close relationship between behaviour and their situation but are problematic for a number of reasons. For example:

- when these situations are not mutually exclusive an agent might be forced into an inconsistent state if it has prescriptive behaviour based on several situations;
- errors in perception or inconsistencies caused by conflicting behaviours in overlapping situations may lead to errors in the agent, reducing its fault-tolerance;
- it assumes that the set of situations affecting the agent can be identified (fully and correctly) in advance — in an open system this is, of course, impossible; and
- assuming all situations can be correctly identified in advance and that the agent's behaviour in all possible combinations *can* be described, doing so is only practical for a small number of situations.

Declarative programming provides a solution to these problems. It has been well established in academic communities for decades but has also enjoyed commercial prominence, most notably for the success of expert systems [114]. The most popularly declarative approach to programming is that of logic programming, exemplified by Prolog [86], but other approaches are also popular. For example, functional and constraint-based languages.

Logic programming languages such as Prolog essentially contain a set of rules and a set of facts, each corresponding to different types of formula within predicate logic. An execution involves querying this 'knowledge base' by presenting a predicate of unknown value and attempting to reduce it to the terms of known value. Some reductions fail, and so backtracking is employed to force alternative reductions. The goal of execution is to demonstrate that the query is satisfiable and to provide grounded terms for any variables that appear in the query. Whilst later implementations of Prolog provide enhancements such as the programmatic modification of the knowledge base and the possibility to interact with backtracking, Prolog is typically used as a 'plugged-in' reasoning engine for applications developed with more conventional imperative languages, where the Prolog engine is used to make isolated queries based upon a static knowledge base.

Declarative agent programming requires more sophistication due to the dynamic and reactive nature of agents. An agent does not maintain a static knowledge base of facts. It reasons with changing beliefs and attempts multiple goals concurrently. However, logic programming does have a valuable characteristic that is well suited to agent programming. Goals, in logic programs, are not satisfied by an explicit sequence of rules and/or facts. Likewise, by merely describing what the agent wishes to achieve (its 'goals'), and by discretely providing plans with the ability to affect/modify the

agent's behaviour on the way to these goals, some of the problems identified above can be alleviated.

Efforts to develop parallel logic programming languages (such as Parlog [68], P-Prolog [137] and Concurrent Proglog [117]) have been made. Practical problems included inefficiency of memory management and scheduling overhead. Whilst reasoning complexity restricts them to either and- or or-parallelism [104].

3.2 Specifying programs with temporal logic

The benefits of adequate, unambiguous and precise specifications in any engineering activity is well established. Such a specification formalises the requirements of a project's output, providing a reference point against which the product can be assessed. Any software that claims to satisfy its specification has likely undergone a process of verification during which it is scrutinised with respect to each aspect of its specification. When this scrutiny is performed manually or when the specification is incomplete, confidence in the efficacy of verification is lost.

In addition, certain categories of software require a high level of confidence, or even formal proof, that the output satisfies the specification. In these situations practitioners look for automated techniques and formal specification languages to achieve high levels of assurance. This section describes how temporal logic is used to create an agent specification language that allows an agent specification to be executed directly. This technique not only makes the human interpretation of specifications redundant but also reduces the often error-introducing processes of design and implementation. Due to its strict logical foundations, this technique results in agent executions that are guaranteed to satisfy their specification, providing of course that the specification *is* satisfiable to begin with.

3.2.1 Specifying agent behaviour

As discussed in Section 1.6, the temporal logic employed, PTL, is a variant of classical logic which has been extended with operators having temporal semantics. This section formally introduces this logic, that we will call `PTL`.

Syntax

Being based upon propositional logic, the well formed formulas of `PTL` contain a signature of propositional constants, $\langle \mathcal{P} \rangle$, the propositional symbols **true** and **false**, and the usual connectives $\neg$, $\vee$, $\wedge$ and $\Rightarrow$. To these we add a special symbol, **start**, and the temporal operators $\bigcirc$ (next), $\Diamond$ (sometime), $\Box$ (always), $\mathcal{U}$ (until) and $\mathcal{W}$ (unless).

A well formed formula wff, is defined inductively in the familiar way. Let α be a formula:

if $\alpha \in \mathcal{P}$ then α is a wff,

the symbols **true**,**false** and **start** are wff,

if α is a wff then $\neg\alpha$ is a wff,

if α and β are wff then $\alpha \vee \beta$ is a wff,

if α and β are wff then $\alpha \wedge \beta$ is a wff,

if α is a wff then $\bigcirc\alpha$ is a wff,

if α is a wff then $\Box\alpha$ is a wff,

if α is a wff then $\Diamond\alpha$ is a wff,

if α and β are wff then $\alpha \mathcal{U} \beta$ is a wff,

if α and β are wff then $\alpha \mathcal{W} \beta$ is a wff, and finally

if α and β are wff then $\alpha \Rightarrow \beta$ is a wff.

Semantics

Kripke models are used to provide semantics to modal logics, the intuition they use is that of possible worlds. Thus, if we define our language of temporal logic, PTL, with syntax as described above, a Kripke model, M, is defined by $M = \langle W, R, \pi \rangle$ where

- W is a set of worlds
- R is a binary relations such that $R \subseteq W \times W$
- π is an interpretation function such that $\pi : W \times P \mapsto \{\textbf{true}, \textbf{false}\}$

However, since we are concerned only with linear models of time and hence R is a serial relation, we can represent the set of all possible worlds by the natural numbers, $\mathbb{N}$, and use its semantics of accessibility and ordering. Hence, the model for our language becomes $M = \langle \mathbb{N}, \pi \rangle$, where

- each member of $\mathbb{N}$ is a discrete temporal world
- π is an interpretation function such that $\pi : \mathbb{N} \times P \mapsto \{\textbf{true}, \textbf{false}\}$

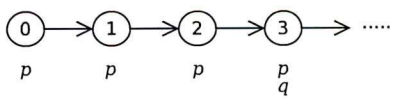
and we use the expression $\langle M, i \rangle \vDash \psi$ to denote that formula ψ is satisfied in temporal world i of model M. In this way, the semantics of formulas in our language PTL are

described formally as follows:

$$
\begin{array}{lll}
\langle M, i\rangle \vDash \textbf{true} & & \\
\langle M, i\rangle \nvDash \textbf{false} & & \\
\langle M, i\rangle \vDash \textbf{start} & \text{iff} & i = 0 \\
\langle M, i\rangle \vDash \psi & \text{iff} & \psi \in P \text{ and } \pi(i, \psi) = \textbf{true} \\
\langle M, i\rangle \vDash \neg\psi & \text{iff} & \psi \in P \text{ and } \pi(i, \psi) = \textbf{false} \\
\langle M, i\rangle \vDash \psi \wedge \phi & \text{iff} & \langle M, i\rangle \vDash \psi \text{ and } \langle M, i\rangle \vDash \phi \\
\langle M, i\rangle \vDash \psi \vee \phi & \text{iff} & \langle M, i\rangle \vDash \psi \text{ or } \langle M, i\rangle \vDash \phi \\
\langle M, i\rangle \vDash \bigcirc\psi & \text{iff} & \langle M, i+1\rangle \vDash \psi \\
\langle M, i\rangle \vDash \Diamond\psi & \text{iff} & \text{there exists } j \in \mathbb{N} \text{ such that } \{j \geq i \text{ and } \langle M, j\rangle \vDash \psi\} \\
\langle M, i\rangle \vDash \Box\psi & \text{iff} & \text{for all } j \in \mathbb{N} \ \{\text{if } j \geq i \text{ then } \langle M, j\rangle \vDash \psi\} \\
\langle M, i\rangle \vDash \psi\,\mathcal{U}\,\phi & \text{iff} & \text{there exists } j \in \mathbb{N} \text{ such that } \{j \geq i \text{ and} \langle M, j\rangle \vDash \phi\} \text{ and,} \\
 & & \text{for all } k \in \mathbb{N} \ \{\text{if } i \leq k < j \text{ then } \langle M, k\rangle \vDash \psi\} \\
\langle M, i\rangle \vDash \psi\,\mathcal{W}\,\phi & \text{iff} & \langle M, i\rangle \vDash \psi\ \mathcal{U}\ \phi \quad \text{or} \quad \langle M, i\rangle \vDash \Box\psi
\end{array}
$$

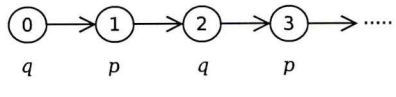
Intuitively, the formula $\bigcirc\psi$ is interpreted as 'in the *next* moment in time, ψ is true'. Similarly, the formula $\Diamond\psi$ is taken to mean '*eventually*, at some future moment, ψ will be true', $\Box\psi$ means 'from this moment onwards, ψ is *always* true', $\psi\,\mathcal{U}\,\phi$ means that 'at some future moment ϕ will be true, and *until* then ψ will true', and finally, $\psi\,\mathcal{W}\,\phi$ means that 'ψ will be true in all future moments *unless* and until such time that ϕ is satisfied'.

Hence, for an arbitrary linear path through a Kripke model, a number of temporal formulas are satisfied. For example, given the path of Kripke worlds depicted here



one can say that the formulas p, $\Box p$, $\Diamond p$ and $\Diamond q$ are satisfied with respect to each of the states, that $\bigcirc q$ is satisfied with respect to state 2 and q is satisfied in state 3. Note, that from this point onwards the term *state* will be used to describe what is analogous to a Kripke world, as this term is in keeping with the use of the term when defining the semantics of programming languages.

If we were to interpret positive predicates in such a model as the actions of an agent, then the path



corresponds to an agent performing actions q, and action p alternately. Let us assume that this is desirable behaviour and that we wish to specify it. I.e. we want to write a temporal formula with respect to the first state, which exactly specifies this behaviour. Such a formula might be

$$q \wedge \bigcirc p \wedge \bigcirc\bigcirc q \wedge \bigcirc\bigcirc\bigcirc p.$$

There are a number of problems with this as a specification and in particular an agent specification;

1. A complete model of behaviour is rarely available.
2. It does not allow for alternative valid executions (e.g. other paths from a Kripke model).
3. It cannot capture the infinitely alternating nature of q and p.

Consider instead, the specification

$$q \wedge \Box(q \Rightarrow \bigcirc p) \wedge \Box(p \Rightarrow \bigcirc q),$$

it is satisfied by the above four states yet it describes a longer sequence of states (infinitely longer) and it captures the consequential (and consecutive) relationship between actions p and q in a way in which the previous specification did not. Concurrent METATEM is an agent specification language that allows agents to be specified by an implied conjunction of temporal formulas similar to this last example. As can be shown, by defining a normal form which includes a reference to a starting state, these formulas can be used to generate a sequence of states that, if possible, satisfies the formulas.

Given a formula φ, of logic $\mathcal{L}$, we construct a model $\mathcal{M}$, for φ such that

$$\mathcal{M} \vDash_{\mathcal{L}} \varphi .$$

Typically, many different models that satisfy φ may exist but by defining a normal form and applyling constraints and heuristics, a valid model can be generated if φ is satisfiable.

3.2.2 Separated Normal Form

An irreducible set of formula types called Separated Normal Form (SNF) was first proposed in [46] and further developed in [50]. Any PTL/TL formula can be translated to SNF formulas such that the SNF formulas are equivalent to the original but, of course, in a different form. SNF employs only three temporal operators, $\bigcirc$, $\Diamond$ and $\Box$, having *strong next*, *reflexive sometime* and *always* respectively, with $\Box$ used only once

to range over a conjunction of all sub-formulas. To this we added a non-temporal rule form to help reduce the size of a specification. Our standard form is defined as

$$\Box \bigwedge_{a=1}^{n} R_a$$

where each R_a is a 'rule' of one of the following forms

a **start rule**	$start \Rightarrow \bigvee_{b=1}^{m} l_b$
a **next rule**	$\bigwedge_{c=1}^{p} k_c \Rightarrow \bigcirc \left[\bigvee_{d=1}^{q} l_d \right]$
a **sometime rule**	$\bigwedge_{e=1}^{r} k_e \Rightarrow \Diamond l$
or a **non-temporal rule**	$\bigwedge_{f=1}^{s} k_f \Rightarrow \bigvee_{g=1}^{t} l_g$

and where l_b, k_c, l_d, k_e, l_g and l are literals. Thus, start rules allow a number of alternative interpretations for a given execution's first state. Next rules provide choice points during state transition such that, if in any state all of $k_1, \ldots, k_p$ are true, then in the next state at least one of $l_1, \ldots, l_q$ must be satisfied. Sometime rules provide goals that direct decision making at these choice points, by providing a constraint on a future state such that, if in any state all of $k_1, \ldots, k_r$ are true then a future choice must satisfy l. Finally, non-temporal rules provide a way of expanding a choice with non-temporal aspects. In summary, the normal form requires that all negations apply only to literals, that all temporal operators other than $\bigcirc$ and $\Diamond$ are removed, and that all occurrences of the $\Diamond$ operator apply only to literals. An example specification, with an implied *always* conjunction omitted, is given below.

$$\begin{aligned}
start &\Rightarrow seek \\
start &\Rightarrow (move \vee turn) \\
seek &\Rightarrow \Diamond found \\
(clear \wedge \neg found) &\Rightarrow \bigcirc(move \vee turn) \\
(\neg clear \wedge \neg found) &\Rightarrow \bigcirc \neg move
\end{aligned}$$

Removal of $\mathcal{U}$ and $\mathcal{W}$

It should be noted that although SNF uses a restricted set of temporal operators, it does not restrict expressivity as the 'until' and 'unless' operators can be re-written in terms of 'next' and 'sometime' before subsequent transformation into normal form.

Each re-write rule involves introduction of a new proposition, named x in the examples below.

$$a \mathcal{W} b \equiv x \wedge (x \Rightarrow (b \vee (a \wedge \bigcirc x)))$$
$$a \mathcal{U} b \equiv x \wedge (x \Rightarrow (b \vee (a \wedge \bigcirc x))) \wedge (x \Rightarrow \Diamond b)$$

3.2.3 From specification to execution

Given a specification, the idea is to *execute* it by building a concrete model for it, hence model-building for temporal formulae was developed using a principle referred to as the *imperative future* by the authors of [62, 6], essentially comprising forward chaining from initial conditions and building the future, state by state.

Basic execution

Recall that we consider an agent execution to be a linear sequence of states starting from an initial state that has no predecessor and which satisfies the special symbol, **start**. Given a temporal description, using the above language, execution takes the following approach:

- ensure that all formulas conform to SNF and perform any transformations as necessary [50];
- from the initial constraints, as determined by the 'start rules', *forward chain* through the set of temporal rules constraining the *next* state of the agent; and
- constrain the execution by attempting to satisfy eventualities (aka goals), such as $\Diamond g$ (i.e. g eventually becomes true). (This, in turn, involves some strategy for choosing between such eventualities, where necessary.)

Later in this section we describe the execution algorithm more formally but first let us look at some simple examples of its execution.

Examples of basic execution

Several basic examples are now considered, in order to describe how execution of simple specifications occur.

Example 1

Consider a machine capable of converting raw material into useful 'widgets', that has a hopper for its raw material feed which, when empty, prevents the machine from producing widgets. A simple specification for an agent controlling such a machine, presented in the normal form described above, is as follows (each rule is followed by an informal description of its meaning):

start $\Rightarrow$ *hopper_empty*

The hopper is initially empty.

$true \Rightarrow power$

The machine has uninterrupted power.

$hopper\_empty \Rightarrow \bigcirc fill\_hopper$

If the hopper is empty, then it must be refilled in the next moment in time.

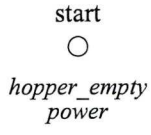
$fill\_hopper \Rightarrow \bigcirc(material \vee hopper\_empty)$

Filling the hopper is *not* always successful.

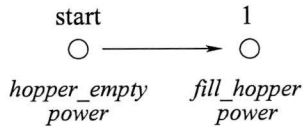
$(material \wedge power) \Rightarrow \bigcirc widget$

If the machine has power and raw material then, in the next moment in time a widget will be produced.

Execution begins with the construction of an initial state which is constrained by the start rules and any present-time rules. Thus, in the *start* state our machine has an empty hopper and power:



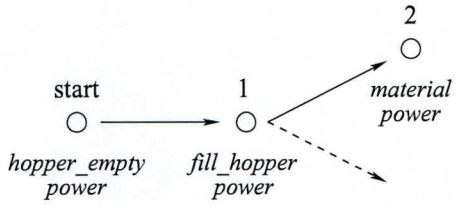
The interpretation of each state is used to derive constraints on the next state. Applying the above rules to this initial state produces the constraint $fill\_hopper$, which must be true in any successor state. The METATEM execution algorithm now attempts to build a state that satisfies this constraint and is logically consistent with the agent's present-time rules. In this example we have only one present-time rule, which does not contradict our constraints but does introduce another constraint, hence state 1 is built:



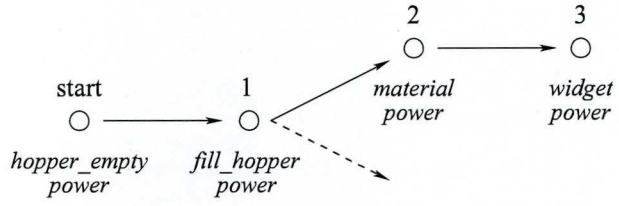
State 1 provides the METATEM agent with its first choice point. Evaluation of the agent's rules constrains the next state to satisfy the disjunction

$$((material \wedge power) \vee (hopper\_empty \wedge power)).$$

Without any preferences or goals to guide its decision, the METATEM agent is able to choose either alternative and makes a non-deterministic choice between disjuncts. For this example we will assume that *material* is made true in state 2:



In this state, our machine has both the power and material necessary to produce a widget in the next state:



Note. Without explicit rules, be they temporal or non-temporal, the machine no longer believes it has its raw material. Hence, evaluation of the agent's temporal rules with the interpretation of state 3 produces no constraints and the agent will produce no further states.

Example 2

This example illustrates the backtracking nature of the METATEM algorithm when it encounters a state that has no logically consistent future. Staying with our widget machine, we modify its non-temporal rule and provide an additional rule (which is obviously valid, but is useful for explanatory purposes):

$true \Rightarrow (power \vee \neg power)$

Power can now be switched 'on' or 'off'.

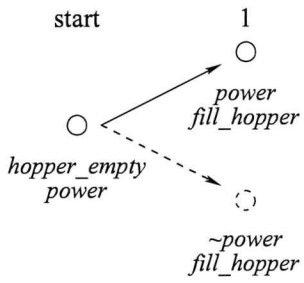
$(fill\_hopper \wedge power) \Rightarrow \bigcirc false$

Filling the hopper with the power switched on causes irrecoverable problems in the next state!

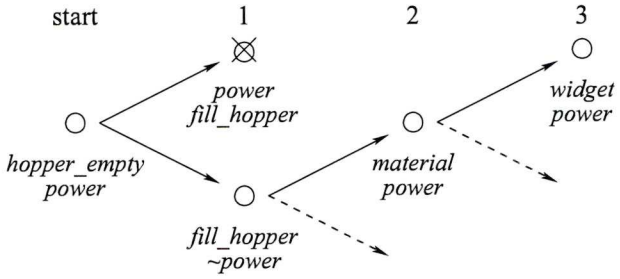
Execution now begins in one of two states,

$$(hopper\_empty \wedge power) \text{ or } (hopper\_empty \wedge \neg power)$$

due to the conjunction introduced by the modified present-time rule. Let us assume that the former is chosen, though it is inconsequential to our example. Again our agent has a choice when constructing the next state, it can fill the hopper with the power on or with the power off. Each of these choices has a consistent present but only one has a consistent future! Let us assume that the 'wrong' choice is made and the 'correct' choice is retained for future exploration;



Now, evaluation of state 1's interpretation constrains all future states to include $false$— this state has no future. It is at this point, when no consistent choices remain, that METATEM backtracks to a previous state in order to explore any remaining choices. Note that, in this example, the agent's ability to fill its hopper is considered to be *reversible*. However, as will be discussed later, this METATEM implementation distinguishes between reversible actions and those that cannot be reversed and hence prevent backtracking from states in which they hold true. This is an example of how the semantics of concurrency and agency have influenced the purely logical origins of Concurrent METATEM. The sending of messages is an important example of an irreversible action. Execution then completes in much the same way as the previous example:



At this point it should be emphasised that the above executions are, in each case, only one of many possible models that satisfy the given temporal specification. Indeed, many models exist that produce no widgets at all. To ensure the productivity of our widget machine we must introduce a goal in the form of an eventuality. For the next example we return to our conversational agent to demonstrate the use of temporal eventualities.

Example 3

For this example, a simple protocol for a successful conversation between two courteous agents is specified:

$true \Rightarrow \bigcirc(speak \vee listen)$

Attentive agents are always speaking or listening...

$speak \Rightarrow \neg listen$
$listen \Rightarrow \neg speak$

...but never at the same time.

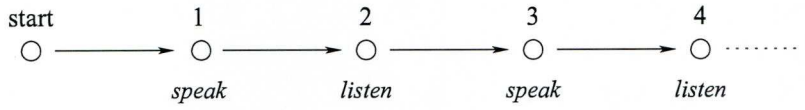
$listen \Rightarrow \Diamond speak$

Will speak after listening...

$speak \Rightarrow \bigcirc listen$

and always pause to listen, after speaking.

The model resulting from execution of this specification is one which alternates between listening and speaking in successive states;



Although intuitively we may expect to see multiple listening states between each speaking state, the METATEM algorithm endeavours to satisfy outstanding eventualities (goals) *at the earliest opportunity.* That is, providing it is logically consistent to do so, an eventuality (such as "$\Diamond speak$") will be made true without being explicitly stated in the consequents of a *next* rule. There are no conflicting commitments and therefore there is no need to delay its achievement.

Execution algorithm

Having described the intuition of the forward-chaining execution of a METATEM agent, we now describe it formally. Recall that an agent specification is a PTL formula, let us call it φ, and that execution means constructing a sequence of states σ, that satisfies

$$\langle \sigma, 0 \rangle \models \varphi \ .$$

That is, we are attempting to construct a model for the formula that corresponds to the set of rules. If we subdivide φ into the sets Initial, Next, and Sometime. Where Initial contains rules of the form "$start \Rightarrow \ldots$", Next contains rules of the form "$\ldots \Rightarrow \bigcirc \ldots$", Sometime contains rules of the form "$\ldots \Rightarrow \Diamond \ldots$", and of course

$$\varphi \Leftrightarrow \mathsf{Initial} \wedge \mathsf{Next} \wedge \mathsf{Sometime}$$

and identify two lists; E, the outstanding eventualities for each state (such that E_i are the eventualities outstanding at state i) and S, the Boolean assignments of propositions for each state (S_i for state i). Then the execution proceeds according to the following algorithm [54].[2]

1. Make a (consistent) choice of Boolean assignments for propositions as described by the Initial set of rules; label this as S_0 and let $E_0 =< >$.

[2]For simplicity, we will assume that all the *sometime* rules within Sometime are of the form $A \Rightarrow \Diamond^+ B$, where '$\Diamond^+$' is the non-reflexive version meaning "at some point from the next moment in time onwards"; clearly, $\Diamond p \Leftrightarrow (p \vee \Diamond^+ p)$.

2. Given S_i and E_i, proceed to construct S_{i+1} and E_{i+1} as follows:

   (a) Let $C = \{F \mid (P \Rightarrow \bigcirc F) \in \mathsf{Next} \text{ and } S_i \vDash P\}$

   i.e., C represents the constraints on S_{i+1} derived from the Next clauses.

   (b) Let $E_{i+1} = E_i \frown \langle G \mid (Q \Rightarrow \Diamond^+ G) \in \mathsf{Sometime} \text{ and } S_i \vDash Q \rangle$

   i.e., E_{i+1} is the previous list of outstanding eventualities, with all the newly generated eventualities appended.

   (c) For each $V \in E_{i+1}$, starting at the head of the list,

   if $(V \wedge C)$ is consistent,

   then update C to $(C \wedge V)$ and remove V from E_{i+1}.

   else leave V within E_{i+1}.

   (d) Choose an assignment consistent with C and label this S_{i+1}. If there are no consistent assignments then backtrack to a previous choice point.

   (e) Loop check:

   if V has occurred continuously in all of E_i, E_{i-1}, ..., E_{i-N}

   then fail and backtrack to a previous choice point

   else go to (2)

The core METATEM execution mechanism, as defined above, with the strategy involving attempting the oldest outstanding eventualities first, at the choice in step 2b, and forcing backtracking at step 2e if the same outstanding eventualities occur for N states continuously (where N is related to the bound on the size of a finite model for the logic), is *complete*, i.e., an execution (a model) will be produced if, and only if, the original formula is satisfiable. The proof of this theorem for basic METATEM can be found in [5, 6].

Strategies and deliberation

Where there are multiple outstanding eventualities, and where only a subset of these can be satisfied at the same time, then some strategy for deciding which eventualities to satisfy now, and which to hold over until future states, is required. As shown in the previous section, the specification precludes both *speak* and *listen* being true at the same point in time. Thus, if it is specified that both $\Diamond$*listen* and $\Diamond$*speak* are to be made true many times, then when to make *speak* true, when to make *listen* true, and when to make neither true must be determined.

The basic strategy for deciding between conflicting eventualities is provided directly by the original METATEM execution algorithm. This is to choose to satisfy the eventuality that has been outstanding (i.e. needing to be satisfied, but as yet unsatisfied) the longest. This has the important benefit that it ensures that no eventuality remains

outstanding forever, unless it is the case that the specification is unsatisfiable [6], and ensures the correctness of the basic execution.

There are, however, a number of other more sophisticated mechanisms for handling such strategies that have been developed. The most general is that described in [51] which maintains the outstanding eventualities at any moment in time as a list. The eventualities will then be attempted in list-order. Thus, in the basic METATEM case the list has a natural ordering based on the age of the eventualities. When an eventuality is satisfied, it is removed from the list; when a new eventuality is generated, it is added to the end of the list, effectively forming a queue of eventualities. An alternative strategy is to attempt all eventualities at the first opportunity, regardless of whether or not any other eventualities remain unsatisfied. This strategy prioritises newly instantiated eventualities with the aim of satisfying them before they become *outstanding*; this strategy can be likened to a human strategy of tackling the 'here-and-now'. These strategies are general strategies and are applied prior to execution. A more advantageous strategy should be flexible enough to allow an agent to modify its deliberation strategy at runtime. With this aim, a 'prefer' directive has been proposed to provide a convenient way to express a priority ordering for all predicates (actions, commitments, eventualities, etc.) [77]. Section 3.3 of this chapter describes how this project's METATEM implementation enables agents to modify this ordering dynamically, using the 'prefer' and other directives, allowing them to adapt their behaviour and/or strategy according to the context of their activity.

3.3 Implementation algorithm

The algorithm presented in Section 3.2.3 is the theoretical basis of METATEM. This section presents its practical implementation, via the platform-independent programming language, Java. In addition to the underlying theory, this implementation provides some enhancements, also described here. There have been a number of prototype implementations of Concurrent METATEM before, but none that contain all the features of this project's implementation or written for long-term evaluation and maintenance. Specifically, this section presents the syntax, semantics and execution algorithm of the implementation and how it differs from the language described in Section 3.2.

3.3.1 Syntax and semantics

The implemented version of METATEM differs in certain details from the fundamental language described earlier. It allows more formula types, set term types with appropriate semantics and provides some agent-specific constructs to facilitate the specification of multi-agent systems. As it allows first-order predicates and implied quantification of variables, so the specific normal form used is more complex. The implemented version

of each rule from the Separated Normal Form is given below along with illustrative examples in first-order temporal logic and, with the implied quantification, an equivalent example in the syntax required by the implementation.

Start rule:

General $\quad start \Rightarrow \exists \overline{x}. \bigvee_{i=1}^{a} p_i(\overline{x})$

Example $\quad start \Rightarrow \exists x.[p(x) \vee q(x)]$

Code

```
start => p(X) | q(X);
```

Step rule:

General $\quad \forall \overline{x}. \Big[\Big[\bigwedge_{i=1}^{a} p_i(\overline{x}) \wedge \exists \overline{y}. \bigwedge_{j=0}^{b} q_j(\overline{y}, \overline{x}) \Big] \Rightarrow \bigcirc \exists \overline{z}. \bigvee_{k=1}^{c} r_k(\overline{z}, \overline{x}) \Big]$

Example $\quad \forall x. \Big[[p(x) \wedge \exists y.q(y,x)] \Rightarrow \bigcirc \exists z.[r(z,x) \vee s(z,x)] \Big]$

Code

```
p(X) & q(Y,X) => NEXT r(Z,X) | s(Z,X);
```

Sometime rule:

General $\quad \forall \overline{x}. \Big[\Big[\bigwedge_{i=1}^{a} p_i(\overline{x}) \wedge \exists \overline{y}. \bigwedge_{j=0}^{b} q_j(\overline{y}, \overline{x}) \Big] \Rightarrow \Diamond \exists \overline{z}.r(\overline{z}, \overline{x}) \Big]$

Example $\quad \forall x. \Big[[p(x) \wedge \exists y.q(y,x)] \Rightarrow \Diamond \exists z.r(z,x) \Big]$

Code

```
p(X) & q(Y,X) => SOMETIME r(Z,X);
```

Note that the *sometime* rule here has the reflexive semantics introduced earlier.[3] In addition to the rule types common to SNF, the implementation allows the use of non-temporal rules and a further two temporal operators: *until* and *unless*. These new operators may only be used on the right-hand side of rules and preclude the use of other temporal operators in the same rule. These additional rules are now presented in the same manner as the above rules.

[3]Recall that the algorithm presented used the non-reflexive $\Diamond^+$.

Non-temporal (present-time) rule:

General $$\forall \overline{x}.\Big[\Big[\bigwedge_{i=1}^{a} p_i(\overline{x}) \wedge \exists \overline{y}.\bigwedge_{j=0}^{b} q_j(\overline{y},\overline{x})\Big] \Rightarrow \bigvee_{k=1}^{c} r_k(\overline{x})\Big]$$

Example $$\forall x.\Big[\big[p(x) \wedge \exists y.q(y)\big] \Rightarrow \big[r(x) \vee s(x)\big]\Big]$$

Code

```
p(X) & q(Y) => r(X) | s(X);
```

Until rule: (and similarly for unless)

General $$\forall \overline{x}.\Big[\Big[\bigwedge_{i=1}^{a} p_i(\overline{x}) \wedge \exists \overline{y}.\bigwedge_{j=0}^{b} q_j(\overline{x},\overline{y})\Big] \Rightarrow \Big[r(\overline{x})\ \mathcal{U}\ s(\overline{x})\Big]\Big]$$

Example $$\forall x.\Big[\big[p(x) \wedge \exists y.q(x,y)\big] \Rightarrow \big[r(x)\ \mathcal{U}\ s(x)\big]\Big]$$

Code

```
p(X) & q(X,Y) => r(X) UNTIL s(X);
```

Note the we are able to omit explicit quantification symbols from the program code by applying the following assumptions.

- Any variable appearing positively in the antecedents, and either positively or negatively in the consequents, of a rule is universally quantified.
- Any variable that appears either only in the antecedents or only in the consequents, is treated as existentially quantified.
- Any variables whose appearances in the antecedents are all negative are treated as existentially quantified, and all negated literals containing the variable are ignored.
- Any variable appearing only in the consequents of a rule (including those that have an ignored negative counterpart in the antecedents) are treated as existentially quantified.

The adoption of these assumptions along with the algorithm used for execution means we can enforce a number of restrictions on, for example, the valid appearances of existential variables. These restrictions are now listed along with some explanation of their purpose. Each of these restrictions is enforced by the implementation by validation checks and subsequent parse errors, prior to execution.

1. **Non-temporal rules must not contain negations in their antecedents.**
   These rules types are fired recursively, expanding the predicate constraints on a state after each recursive call. The presence of negations in the antecedents can cause difficulties when a rule containing such a negation is fired prematurely and so preventing the complete generation of choices.

2. ***Sometime, until* and *unless* rules must not contain negations in their antecedents.**
   Due to the fact that these rules are re-written in terms of $\Diamond$, $\bigcirc$ and non-temporal rules, prior to execution, they too are subject to restriction number 1.

3. **No negated existential variables may appear in the consequents of any rule.**
   As $\exists x.\neg p(x)$ can be trivially satisfied according to the open-world principle, such predicates are prohibited to prevent misleading code.

4. **Non-temporal rules must not contain existential variables in their consequents.**
   Existential variables in the consequents of rules are grounded by Skolemisation. This restriction prevents the infinite generation of terms by an endless loop of term generation and rule firing.

Recall that if the consequent of a fired sometime rule cannot be satisfied immediately, it is appended to a list of outstanding eventualities, where it remains until satisfied. Thus this list of 'goals' is naturally ordered by their ages. With this list view, our basic strategy for deciding which eventualities to satisfy next is an 'oldest-first' strategy based on this natural order. Thus, if the agent can *re-order* this list between states then it can have a quite sophisticated strategy for *deliberation*, i.e. for dynamically choosing what to tackle next. This approach is discussed further in [51, 53] but, unless we put some constraints on the re-ordering we might apply, then there is a strong danger that the completeness of the execution mechanism will be lost [53].

In the current implementation, rather than using this quite strong, but dangerous, approach we adopt simpler, and more easily analysable, mechanisms for controlling (or at least influencing) the choice of eventuality to satisfy. These mechanisms are characterised by the predicates/directives `atLeast`, `atMost` and `prefer`.

The `atLeast` predicate places a minimum constraint on the number of instances of positive predicates, whilst `atMost` places a maximum constraint on the number of instances of positive predicates in a given temporal state, in the style of the capacity constraints described by [38]. Besides providing the developer with the ability to influence an agent's reasoning, when applied judiciously `atMost` and `atLeast` can simplify the fragment of the logic considered and hence can increase the execution performance of a METATEM agent.

As an example of the use of predicate constraints we provide some code snippets from an example included with the METATEM download, which specifies the behaviour of a lift. The lift responds to calls from floors above and below it and, when more than one call remains outstanding, it must decide which call to serve first, changing direction if necessary. Each discrete moment in time of our temporal model denotes the lift's arrival at a floor and the transition between temporal states is analogous to the lift's transition between floors. The following rules specify that the lift starts at the ground floor and must satisfy all calls before it can achieve the `waiting` state:

```
start => atFloor(0);
true => SOMETIME waiting;
call(X) => ~waiting;
```

Clearly, it is desirable that the lift visits a floor in each state of our model. This behaviour could be specified by the rule

```
true => NEXT atFloor(X);
```

which states that there must exist an `X` such that `atFloor(X)` is satisfied in each moment in time. However, our lift must visit *one and only one* of a limited number of *valid* floors. The above rule is logically too general as it allows multiple `X`'s in any moment in time and implies an infinite domain of `X`.[4] Therefore our lift specification does not use the rule given immediately above, but instead employs predicate constraints. These ensure that the lift visits one and only one floor at each moment, without introducing an existential variable. The following declarations in an agent description file achieve this.

```
at_most 1 atFloor true;
at_least 1 atFloor true;
```

These are examples of 'at most' and 'at least' directives which constraint the number of positive predicates with a specified symbol in a state. These constraints can be given a *satisfying condition* that determines when the constraint is applied. In both examples shown, the satisfying condition is `true`, hence these constraints apply in all moments. Thus, due to the two directives, exactly 1 predicate with the symbol `atFloor` must appear in all states.

3.3.2 How deliberation is implemented

The construction of each temporal state during the execution of a METATEM specification generates a logical interpretation that is used to evaluate the antecedents of each

[4]Indeed, the current implementation considers existential variables on the right-hand side of future rules on an open-world principle, implementing a form of Skolemisation by, when necessary, creating new terms. In this example our lift could disappear to an imaginary floor!

temporal rule. The consequents of all the rules that fire are conjoined (and transformed into disjunctive normal form) to represent the agent's choices for the next temporal state, each conjunction being a distinct choice, one of which is chosen and becomes the interpretation of the next temporal state, from which the next set of choices are derived. This process is repeated, and conjunctions that are not chosen are retained as alternative choices to be taken in the event of backtracking. As mentioned earlier, a number of fundamental properties of the formulae in each conjunction affect the choice made. For example, an agent will always satisfy a commitment if it is consistent to do so, and will avoid introducing commitments (temporal 'sometime' formulas) if able to, by making a choice containing only literal predicates. These preferences are built-in to METATEM, however the `prefer` construct allows the developer to modify the outcome of METATEM's choice procedure by re-ordering the list of choices according to a declared pair of predicates (e.g. `prefer(win,lose)`) after the fundamental ordering has been applied. We refer to the `prefer` construct as a deliberation meta-predicate and the architecture of the current METATEM allows the implementation of further deliberation meta-predicates as 'plug-ins'.

Each of these constructs can be declared as applicable in all circumstances or as context dependent, that is, only applicable when a given arbitrary formula evaluates positively. Furthermore, each preference is assigned an integer weighting, within the (arbitrary) range of 1–99, which allows a fine-grained ordering of preferences. [5]

For example, the following snippets are two alternative applications of the `prefer` construct to the lift example described above, to encourage the lift to continue moving in the same direction when appropriate:

```
prefer downTo to upTo when moving(down) weight 50;
prefer upTo to downTo when moving(up) weight 50;
```

Alternatively:

```
prefer("downTo","upTo","moving(down)",50)
prefer("upTo","downTo","moving(up)",50)
```

The first two directives above are examples of those that appear in the preamble of an agent definition file, these preferences apply from time, $t = 0$. The latter two directives are examples of meta-predicates that, when appearing in the consequents of a temporal NEXT rule, will provide the agent with the declared preference from the temporal state following the state in which the rule was fired. The former type is simply a syntactic convenience for a rule of the type

```
start => prefer("downTo","upTo","moving(down)",50)
```

Once a preference is applied it is upheld for all future states and there is no mechanism

[5] We reserve the weighting values 0 and 100 for built-in preferences.

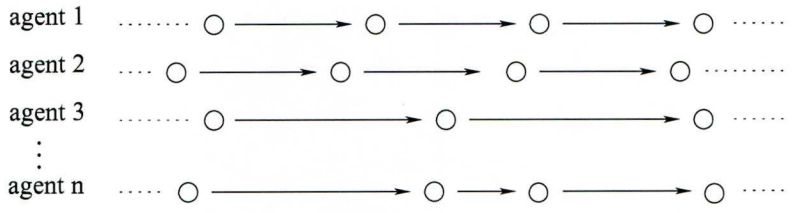


Figure 3.1: Typical asynchronous agent execution.

for explicitly deleting it, instead preferences can be *overridden* by an otherwise identical preference which declares a higher priority value or *counteracted* by an opposing preference. However, the use of context dependent preferences is encouraged as leaving a context provides the effect of deleting a preference but with the benefit that the preference will be reinstated upon entering the relevant context. We believe this is a natural interpretation of preferences.

3.3.3 Multiple agents

METATEM supports the asynchronous, concurrent execution of multiple agents which are able to send one another messages that are guaranteed to arrive at some future moment in time. Each agent has its own concept of time and the duration of each time step for an individual agent is neither fixed nor constant throughout execution. Conceptually then, the transition of multiple agents between successive temporal states is as depicted in Figure 3.1.

Note. The form of asynchronous execution seen in Figure 3.1 is a little problematic for propositional temporal logic to represent straight-forwardly. However, as described in [49] a temporal logic based on the *Real Numbers* rather than the Natural Numbers, provides an appropriate semantic basis. Importantly, the propositional fragment of such a *Temporal Logic of the Reals* required still remains decidable [84].

An agent sends a message to another agent by firing a rule whose consequents contain the action predicate $send(Recipient, Message)$ true in one of its own states. This guarantees that at some future moment the predicate $receive(From, Message)$ will be true in at least one state of the recipient agent (where $Recipient$, $From$ and $Message$ are all terms and are substituted by the recipient agent's name, the sending agent's name and the message content, respectively). The *send* predicate is an example of a special 'action' predicate which, when made true, prevents subsequent backtracking over the state in which it holds. For this reason, the use of a deliberate–act style of programming is encouraged in which an agent explores multiple execution paths, performing only retractable internal actions, backtracking when necessary, before taking a non-retractable action.

Although METATEM agents exist as individual threads within a single Java virtual machine there are no other predefined agent containers or agent spaces that maintain a centralised structuring of multiple agents. Instead METATEM follows an agent-centred approach to multi-agent development with the only implemented interactions between autonomous agents being message passing. Support for the abstract structuring of agent societies is provided by internal (to each agent) constructs and is discussed in detail in the next chapter.

3.4 Implementation architecture

This section gives a detailed overview of how METATEM has been implemented. The most significant aspects of the Java implementation are described along with rationale for their design. The purpose of this section is to support any future development of the code, whether its aim is maintenance or advancement. It aims to outline the responsibilities of the most important Java classes, their functionality and their dependencies. As such, it is aimed at a reader with the intent not only to program agents but also to program METATEM and with knowledge of object-oriented programming with Java.

3.4.1 Java packages

The implementation is fully modularised to reduce maintenance effort but also to increase the potential for use of each module in other projects. This section lists each Java package, giving for each, an overview of its purpose and describing any significant programming strategies or patterns that have been used. Complete documentation of each package and the classes they comprise can be accessed at

`http://www.csc.liv.ac.uk/~anthony/metatem/javadoc/index.html`

`metatem`

This is the parent package for all packages and classes, it contains the executable class `metatem.Main`, two interfaces with project-wide scope `MultiAgentSystem` and `MutiAgentView` corresponding to the model and view of the MVC pattern, and an implementation of a model, `BasicMAS`.

`metatem.agent`

The classes and sub-packages contained in `metatem.agent` describe the core agent behaviours. Significant members of this class are the abstract `metatem.agent.Agent`, the concrete `metatem.agent.BasicAgent` and `metatem.agent.AgentSpecification`. `Agent` implements both `metatem.temporal.Term` and `java.lang.Runnable`, naturally reflecting its status and behaviour in the abstract language. Throughout the implementation but particularly with the `Agent` class, care has been taken to provide thorough

encapsulation of data, by providing appropriate access modifiers to all class members. With this in mind, the only public method that `Agent` exposes which modifies its state, is `Agent.send(Message)`. In order to provide agent observers with useful information, it was necessary to provide them with an interface of public methods, but these have been implemented to return deep copies or immutable values of an agent's state.

Finally, the class `metatem.agent.SNFRule` uses Java's reflection to perform validation on rules found whilst parsing the input files. Creating an additional validation check requires only the addition of a method with the signature

```
static void <method_name> (Formula, Formula)
    throws InvalidRuleException;
```

that throws an appropriate exception if the two supplied formulas do not form the head and tail of a valid SNF rule.

`metatem.agent.ability`

Contains interfaces, abstract classes and some concrete classes that provide METATEM agents with the ability to act in their environment. All agent abilities must implement exactly one of the interfaces

```
metatem.agent.ability.InternalAbility
metatem.agent.ability.ExternalAbility
```

the first of these provides an `undo` method, allowing an agent to take reparative action in the event of backtracking over a state in which the action was performed. The latter simply identifies the action as non-retractable and prevents backtracking.

`metatem.agent.communication`

A collection of classes and exception that enable agents to communicate by message passing and facilitate the maintenance of an 'inbox'.

`metatem.parser`

A number of classes generated by the parser generator Javacc,[6] from the parser description file `src/metatem/parser/Parser.jj`. Javacc is a combined lexer and parser tool for compiling pure Java compilers.

`metatem.temporal`

An API for temporal logic that has been designed and built for METATEM but is entirely independent of the other packages. This package effectively encapsulates the logical aspects of a METATEM agent, as well as term matching, arithmetic and set

[6] `http://javacc.dev.java.net/`

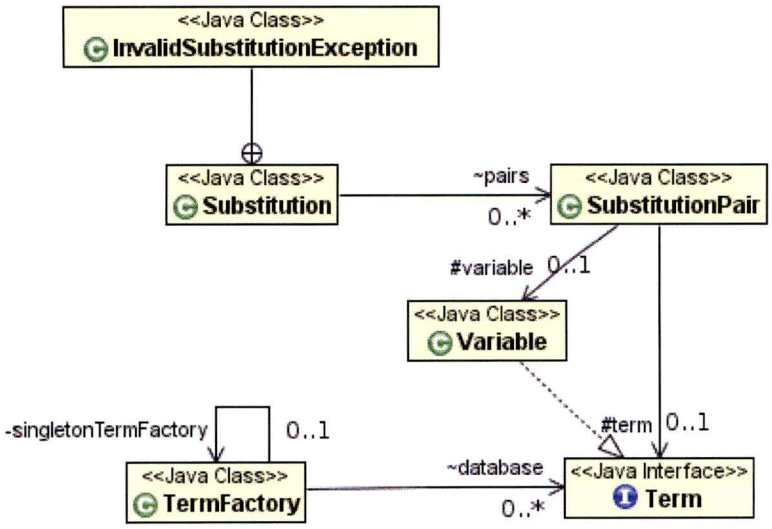


Figure 3.2: A database of unique terms are maintained by the `TermFactory`.

manipulation. The package strongly adheres to the 'program to interfaces' maxim and also provides many custom exceptions that serve to improve the robustness and readability of code. Generation of unique terms is guaranteed by use of the Singleton and Factory patterns, see Figure 3.2, and `MatchingEngine` provides a pluggable engine for an agent to delegate all of its logic operations. Note that agent-centric behaviours such as ordering of choices, remain the responsibility of the `Agent` object. A degree of confidence in the correctness of these packages is achieved through strict enforcement of interfaces, the Adapter Pattern and the immutability of *all* formulas. Figure 3.3 illustrates this clearly.

`metatem.tools`

A package for classes that provide support for the development of METATEM programs. This package contains the agent visualiser classes.

These packages, along with a number of resource files, are packaged as as `metatem.jar` which itself comprises the files `agent.jar`, `temporal.jar` and `tools.jar`, corresponding roughly to the Java packages described above. Together with a Java runtime environment, they form the METATEM runtime environment. An agent system is declared in a number of text files and any custom agent abilities are provided by additional Java classes. Figure 3.4 illustrates then, the essential components required to form a multi-agent system and the dependency relationship between them.

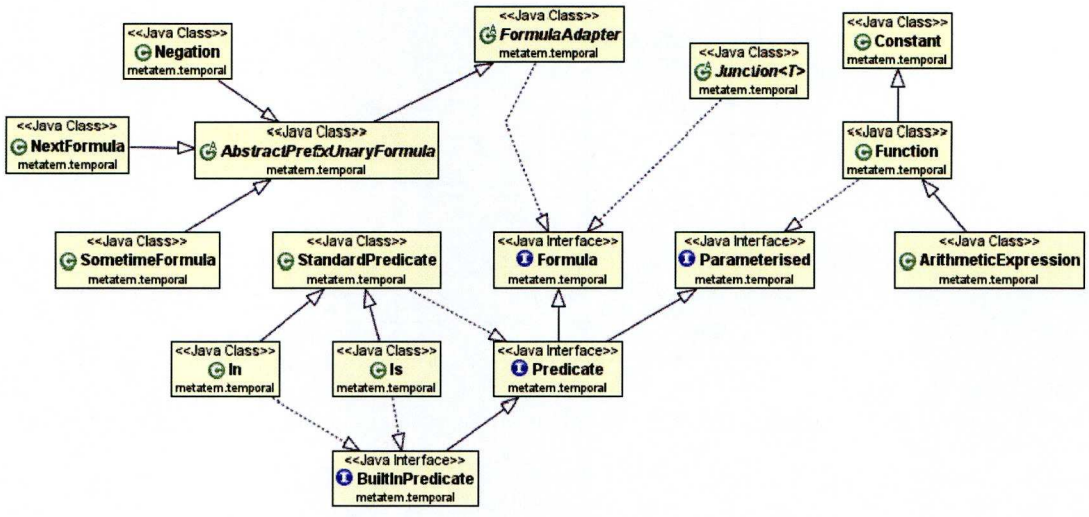


Figure 3.3: Interfaces and abstract classes encourage immutable implementations.

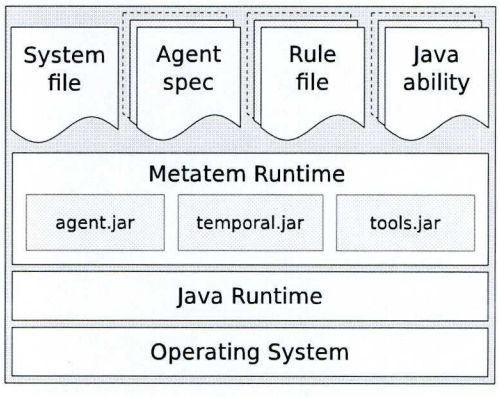


Figure 3.4: The components of a typical multi-agent system.

3.5 Execution example

As has been shown in Sections 3.3 and 3.4, the Java METATEM interpreter resulting from this project faithfully provides semantics for many of the language constructs first developed by Barringer *et al.* in [6]. This section contains a step-by-step execution of a simple multi-agent example as an illustration of these semantics.

Polite conversation

A simple agent specification which describes the art of polite conversation might be:

```
// Start a conversation.
start => speak;

// Pay attention, always speak or listen
true => speak | listen;

// but do not try to do both at the same time,
speak & listen => false;

// eventually speak after listening, and
listen => SOMETIME speak;

// always listen after speaking.
speak => NEXT listen;
```

On execution of this specification, as a single-agent system, the choice for the first state is clear. Logging output begins:

```
[speaker] state 0: [speak]
```

The first `NEXT` rule specifies that either `speak` or `listen` must be satisfied in the next moment, providing a choice for the execution algorithm. However, having just spoken in state 0, the second `NEXT` fires, specifying that, the agent must listen in the next moment in time. Thus, the execution is constrained by this rule and continues as follows:

```
[speaker] state 0: [speak]
[speaker] state 1: [listen]
```

Now, the same choice is encountered for the next moment in time but this time no `NEXT` rules are fired by state 1 and both `speak` and `listen` are logically consistent choices for state 2. However, the `SOMETIME` rule is fired, creating a commitment to satisfy `speak`. So, with no competing commitments or constraints, the execution algorithm satisfies it immediately

```
[speaker] state 0: [speak]
[speaker] state 1: [listen]
[speaker] state 2: [speak]
...
```

and execution continues, alternating between speaking and listening, for ever.

3.6 Extensions

During the interpreter's design and implementation a number of modifications, deviations and elaborations to the theoretical METATEM language were made. Such extensions were often made for convenience or pragmatic reasons but also out of perceived necessity and include diverse features such as the restriction of predicates in non-temporal rules and the provision of programmer-defined add-on sets.

This section aims to describe a number of these extensions in order that

- their syntactic use is clear,
- their semantics can be effectively understood, and
- their implementation is documented where necessary.

Note, that formal semantics for some of these extensions are given in Section 4.3

3.6.1 Sets

The implementation has provided agents with the ability to create, maintain, manipulate and query sets of terms. Terms, of course, include agents themselves. So, in addition to the pre-defined sets `content`, `context` and `known` (the use of which will be explained later), an agent's specification can declare further sets using the syntax demonstrated by this example

```
set term1 : { term2,term3,term4 }
```

where `term1` is the term used as a reference to the implemented set (referred to as the 'set name' from here onwards) and { `term2,term3,term4` } is the set's initial contents and is omitted when an empty set is needed. Any number of additional sets can be declared, providing a unique term is used to refer to each set.

Having declared a set, the set name can be used anywhere it is valid for a constant term to appear, indeed, the set name is a constant term and will be matched against variables and other terms in the usual way when appearing in standard predicates. There are however, a number of non-standard predicates called 'built-in' or 'ability' predicates where the set name is used to refer to the actual set. They are identified here, in Prolog fashion, by their predicate symbol and the number of terms they require.

`add/2` and `remove/2`

This predicate, when appearing on the right hand side of rules, behaves as an internal ability, allowing the addition of a term to a set. When appearing on the left hand side of rules it behaves as a standard predicate. So, assuming that `friends` is a set name, the following rule can be read as "if david has been added to our set of friends then add nick to the set in the next moment".

```
add(david, friends) => NEXT add(nick, friends);
```

Crucially, this predicate may be used to add another agent to the pre-defined sets `content`, `context` and `known` but unlike the external abilities `addToContent/1` and `enterContext/1` which have the same effect on the executing agents, the `add/2` predicate does not have a reciprocal effect on agent being added. The predicate `remove/2` has the expected converse effect on sets.

`in/2`

The `in/2` predicate is an example of a built-in predicate that allows the querying of both explicit sets and referenced sets. It may only appear on the left hand sides of next rules and has two distinct purposes, determined by whether the first argument is a constant or a variable.

1. To evaluate set membership.
2. To retrieve an element of a set.

When a constant term is provided this predicate is used to evaluate set membership, such that `in(david,friends)` evaluates to true if and only if `friends` is a set name *and* `david` is a member of the actual set that `friends` refers to. Alternatively, if after matching, the first argument remains a free variable, this predicate's boolean value is dependent upon whether or not the set is empty and is used to provide a substitution containing all possible variable-constant pairings. For example:

`in(t,{a,b,c})` is `false`,

`in(t,s)` is `true` iff `t` is a member of a set with name `s`, and

`in(X,friends)` is `true` and generates the substitution `[X\david, X\nick]`.

`size/2`

Similar to the `in/2` predicate this predicate provides boolean queries of the size of a set and also generates a substitution pair when the first argument is not grounded:

`size(2,{a,b,c})` is `false`, and

`size(X,friends)` is true and generates the substitution `[X\2]`.

3.6.2 Meta-predicates

The implementation provides a number of predicates that, rather than having domain-level meaning, they operate on or refer to, an agent's execution. These are termed meta-predicates and include the following examples.

`addGoal/1` and `addRule/1`

As their symbols suggests, these predicates provides a mechanism for dynamically adding goals and rules during execution. As neither goals nor rules are constant terms, the argument to `addGoal` and `addRule` must be a quoted term. This term is dis-quoted and parsed using the system parser. Thus the following two rules have similar outcomes

```
p => SOMETIME q;
p => NEXT addGoal("q");
```

but the utility of `addGoal/1` and `addRule/1` is only realised when sending goals and plans to other agents.

```
[sender]      send(receiver,sharedGoal("q"))
[receiver]    receive(From,sharedGoal(G)) => NEXT addGoal(G);
[sender]      send(receiver,sharedPlan("in(sender,context) => jump"))
[receiver]    receive(From,sharedPlan(P)) => NEXT addRule(P);
```

`atMost/3` and `atLeast/3`

These predicates allow the dynamic addition of predicate constraints, as discussed on page 53, when appearing on the right hand side of next rules. Its three arguments are the integer constraint, the symbol to be constrained and a quoted formula. This quoted formula is parsed and evaluated as a contextual condition that determines when the constraint applies.

```
overdrawn => NEXT atMost(1,spend,"prudent");
```

`prefer/4`

Preferences too, can be added dynamically. This meta-ability ensures that the deliberation preference that it describes applies to the the deliberation cycle immediately following its satisfaction. Then, the agent to which the following predicate belongs

```
overdrawn => prefer(spend,save,"impulsive",60);
```

prefers spending over saving after going overdrawn and when feeling impulsive, with a weight of 60.

3.6.3 Abilities

Abilities are provided which allow an agent to print to standard out, send messages to other agents and set a delay timer. Furthermore, a Java interface is provided to allow developers to define their own agent abilities with arbitrary Java code. Indeed, this is to be encouraged for all but the simplest of agent-systems. However, only one ability is mentioned here, essential for all successful societies—the ability to reproduce.

`createAgent/3`

This predicate provides an agent with the ability to create a new agent. Arguments to this predicate define the new agent's name, its temporal specification and its initial relationship to the agent that creates it. It accepts up to three arguments:

```
createAgent(<name_prefix>, content|context|known, <spec>)
```

where the third argument is optional and, if present, must be a quoted string. Some examples and their purpose are given here:

```
createAgent(group, content, "app/subgroup.agent")
```

Creates a new agent whose name will have the prefix 'group' and whose specification is defined by the file subgroup.agent. The newly created agent will, in at least its first state, reside in the content of the creating agent.

```
createAgent(self, context)
```

Creates a new agent that is a 'clone' of the creating agent. The newly created agent has the same initial specification as its creator. The name of the newly created agent will be prefixed with the name of its creator[7] and it will, in at least its first state, reside in the context of the creating agent.

```
createAgent(clone, known)
```

Creates an agent which has the same initial specification as the agent that creates it. The name of the newly created agent will be prefixed by the string 'clone' and it will, in at least its first state, have content:{}, context:{} and known:{<creator>}.

In all cases, the newly created agent will be added to its creators known set and vice-versa. Once the new agent is created and its execution has been started, the creating agent receives a message of the form:

```
receive(self, newAgent(<agent>))
```

[7]Note that the term `self` is a keyword that is replaced at runtime with the executing agent's name which, in this case, is the name of the creator.

3.6.4 Arithmetic

The implementation also supports arithmetic. Using the `is/2` built-in predicate, integer arithmetic expressions containing the operators $*, +, \backslash$ and $-$, are evaluated in the same manner as the Prolog functor with the same name.

3.7 Blockworld

The popular blockworld scenario provides a suitable illustration of the implementation's backtracking and of its efficiency. A single-agent example, requiring a significant number of rules, this example generates a non-trivial number of choices at each state. By performing classical forward-chaining and back-tracking, an agent typically solves a four-block puzzle in about one minute. Part of the specification is listed here.

```
at_most 1 move true;
at_least 1 move ~solved;

//if a block is not moved it remains on the same block
on(X,Y) & ~moving(X) => NEXT on(X,Y) ;

//if a block is clear and is not covered it remains clear
clear(X) &  move(Z,Y) &  X=\=Y => NEXT clear(X) ;

//a block can't be in two places at once
on(X,Y) & on(X,Z) & Y=\=Z => false ;

//if a block is clear then it is either moved or not moved
block(X) & clear(X) & clear(Y) & X=\=Y => move(X,Y) | ~move(X,Y) ;

//if a block is moved and the destination clear
//then the block has a new location...
move(X,Y) &  clear(X) & clear(Y) => NEXT on(X,Y) ;

//...and the previously covered block is now clear
on(X,Y) & move(X,Z) & clear(X) & clear(Z) => NEXT clear(Y) ;

begin => SOMETIME solved ;
```

Finally, logging output showing the final states of the agent `robot`, is listed below. Note that these are the states that lead to a solution of the blockworld problem, following a series of failed attempts and subsequent backtracking. For brevity, only the final backtracking logging statement is included here.

```
[robot] state 1 has no consistent choices remaining, backtracking...
[robot] state 1: [moved(c,table) ^ clear(c) ^ clear(b) ^ on(d,a) ^
        on(c,table) ^ block(b) ^ block(a) ^ clear(d) ^ block(d) ^
        block(c) ^ on(b,table) ^ on(a,table) ^ clear(table) ^
        move(d,table) ^ moving(d) ^ movement]
[robot] state 2: [clear(b) ^ clear(a) ^ moved(d,table) ^ clear(c) ^
        on(a,table) ^ moved(c,table) ^ clear(d) ^ on(c,table) ^
        on(d,table) ^ block(b) ^ block(a) ^ block(d) ^ block(c) ^
        on(b,table) ^ clear(table) ^ move(b,a) ^ moving(b) ^ movement]
[robot] state 3: [on(d,table) ^ clear(table) ^ clear(d) ^
        on(a,table) ^ moved(c,table) ^ on(c,table) ^ clear(b) ^
        moved(b,a) ^ on(b,a) ^ clear(c) ^ moved(d,table) ^
        block(b) ^ block(a) ^ block(d) ^ block(c) ^ move(c,b) ^
        moving(c) ^ movement]
[robot] state 4: [on(d,table) ^ clear(c) ^ moved(c,b) ^ moved(b,a) ^
        on(b,a) ^ clear(d) ^ on(a,table) ^ moved(c,table) ^
        clear(table) ^ moved(d,table) ^ block(b) ^ block(a) ^
        block(d) ^ block(c) ^ on(c,b) ^ move(d,c) ^ moving(d) ^
        movement]
[robot] state 5: [solved ^ clear(d) ^ moved(d,c) ^ clear(table) ^
        moved(b,a) ^ moved(c,b) ^ moved(d,table) ^ on(c,b) ^
        block(b) ^ on(b,a) ^ block(a) ^ block(d) ^ block(c) ^
        on(d,c) ^ on(a,table) ^ moved(c,table)]
```

3.8 Current status

At the time of writing, the implementation is considered to be complete and stable, with respect to the basic features of the language and the additional features described above. It has been released for general consumption and is available for download from

`http://www.csc.liv.ac.uk/~anthony/metatem.html` .

The only system requirement is a Java run-time environment supporting a byte-code version of 1.6 or greater. The download includes

- `metatem.jar`, a package of files containing all the METATEM runtime classes, enabling convenient integration with other Java projects,
- an **examples** directory containing specification examples from this thesis and various other publications,
- comprehensive **documentation of the API** that allows developers to extend an agent or execute arbitrary Java code,

- a **syntax specification** in BNF of the three METATEM input file types,
- an experimental **visualisation tool**, and
- the **source code**.

The implementation is described by a chapter in the second volume of *Multi-Agent Programming* [59] and is used to complement a forthcoming text book from Fisher [54] on formal methods for temporal logic. The system has also been used to implement a virtual cow-herding team, to evaluate its potential use as an entry into the annual Multi-agent programming contest [19]. As a result, we have METATEM users and development is expected to continue, both to support its users and provide enhancements.

While describing the Concurrent METATEM implementation in this chapter, some aspects concerning context have been deliberately omitted. These will be addressed in the next chapter.

Chapter 4

Agents and Contexts

Beginning within the context of executable temporal logics [6], Fisher *et al.* produced a series of papers [60, 58, 52] that developed the METATEM language into a generalised approach for expressing dynamic distributed computations. The language itself is covered in detail in Chapter 3; this chapter describes its agent-organisation features. These features were first proposed by Fisher and Kakoudakis in [60] and subsequently supported by [56, 58, 57, 77, 76] and [74]. As the aim of this project was to implement, extend, apply and evaluate these features, this chapter pays them appropriately high attention.

4.1 Context

As described earlier, we need something more than individual agent specifications if we are to model multi-agent behaviour. Our approach employs the notion of *context* and employs it as a first-class entity in the language, such that context has the potential to trigger changes in behaviour, and provide a means by which an agent can evaluate the relative merits of its choices with respect to its personal attributes such as goals, preferences and beliefs. In this approach a context is not prescriptive, it does not change basic abilities but aims to enhance an agent's effectiveness and appropriateness for the contexts in which it finds itself, at run-time. Agents are not halted by inconsistent combinations of context, are able to respond to unexpected changes in context and are practical to specify.

With the agent metaphor in mind, contexts might naturally influence an agent's behaviour when attempting to achieve its goals. Consider an agent trying to achieve some tasks. It might be in a context that provides additional capabilities, for example it has access to resources provided by other agents or by sources within the environment which are intrinsic to that context. Alternatively, the agent might be within a context that restricts its behaviour, for example through regulatory or resource restrictions, or through norms/roles of behaviour within that context. In this case the agent's behaviour is again modified. Importantly, the agent can be in *both* these contexts at

the same time. Thus the agent's behaviour can be affected in quite complex ways by being within such multiple contexts, making any imperative expression of behaviour problematic.

4.1.1 Example

We can see an analogy with practical human reasoning where contexts can both 'enhance' and 'prune' a human's preferred choices. Humans rarely make entirely rational decisions but have a vital ability to assess the contextual relevance of their choices and modify their rational behaviour when appropriate to do so. Importantly, such behaviour modification does not necessarily come from within. Consider, for example, someone who is a member of a golf club. By being within the golf club context, the player might well get improved facilities/capabilities/skills in relation to hitting the golf ball. Thus, being in the *golf club resources* context can enhance the player's choices. However, being in the golf club also restricts aspects, such as dress code and even gender. Hence, being in the *golf club etiquette* context imposes various norms/rules and regulations of behaviour on the player. This may well lead to the player having behaviour that is not rational in other circumstances, but is perfectly explainable by the context in which the player resides. For example, consider the behaviour of a golfer who adheres to the dress code by wearing argyle patterned jumpers and plus fours. For an example that is more analogous to inter-agent interactions, consider the difference in demeanour, language and veracity, between two humans inside and outside a court of law. Behaviour in both individuals will be affected by the wearing of a wig[1] by either individual.

4.2 Organisation by context

Given the basic principles of an agent's internal architecture and the popularity of multi-agent organisation abstractions any language constructs provided for the purposes of agent organisation should ideally allow the modelling of the all the organisational abstractions reviewed in Section 2.6, as well as be adaptable to future trends.

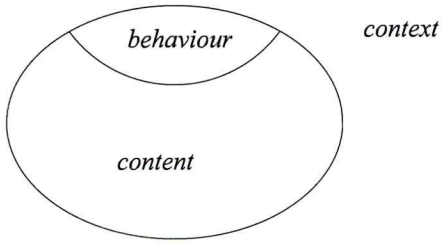
The notion of *context*, as that part of an agent's environment that provides the greatest meaning to its actions and hence should have a significant influence on its deliberation process, is the fundamental notion underlying METATEM's agent organisation strategy. Specifically, it provides an agent with two sets, named *context* and *content*, of agents. Structurally, an agent's *content* set describes those agents that it contains and its *context* set describes those agents that it is contained by. However, it is useful to assign other, non-structural, meanings to the relationships between an agent and the agents in its *context* and *content* sets. For example, agents that have a subordinate relationship may form the *content* set while an agent's *context* set may

[1] In Britain, wigs are worn by judges and others significant court officials, as a symbol of their office.

represent those agents it is submissive to. Alternatively, an agent's *context* may represent agents with whom it wishes to co-operate, whilst its *content* are those agents who wish to co-operate with it. Most generally, we say that an agent's *context* is the set of agents that have some influence on the agent's behaviour, and its *content* is the set of agents over which it has some influence. The formal definition of an agent for our purposes is [60]

$$\begin{array}{lllll} Agent & ::= & behaviour: & specification \\ & & content: & \mathcal{P}(Agent) \\ & & context: & \mathcal{P}(Agent) \end{array}$$

where $\mathcal{P}(Agent)$ are sets of agents and *specification* is the individual agent's behaviour described by our language of temporal logic. Crucially, the membership of these sets changes over time and an agent has access to the sets. Thus, an agent is able to adapt its behaviour at any moment in time, according to the membership of these sets. Graphically, we depict an individual agent as residing in a context and enclosing a content; thus the agent is the oval in:



The addition of *content* and *context* sets to each agent provides significant flexibility for agent organisation. Agent teams, groups or organisations, which might alternatively be seen as separate entities, are now just agents with non-empty *content*. This allows these organisations to be hierarchical and dynamic, and provides possibilities for a multitude of other co-ordinated behaviours. Similarly, agents can have several agents within their *context*. Not only does this allow agents to be part of several organisational structures simultaneously, but it allows the agent to benefit from *context* representing diverse attributes/behaviours. So an agent might be in a context related to its physical locality (i.e. agents in that set are 'close' to each other), yet also might be in a context that provides certain roles or abilities. Intriguingly, agents can be within many, overlapping and diverse, contexts. This gives the ability to produce complex organisations, in a way similar to multiple inheritance in traditional object/concept systems. For example, see Figure 4.1 for sample configurations.

An important aspect is that this whole structure is very dynamic. To reflect the nature of contemporary applications, agents must be able to move in and out of *content* and *context* sets and new agents (and, hence, organisations) should be easily created and/or discarded. Accordingly, no restrictions on set membership are enforced. An

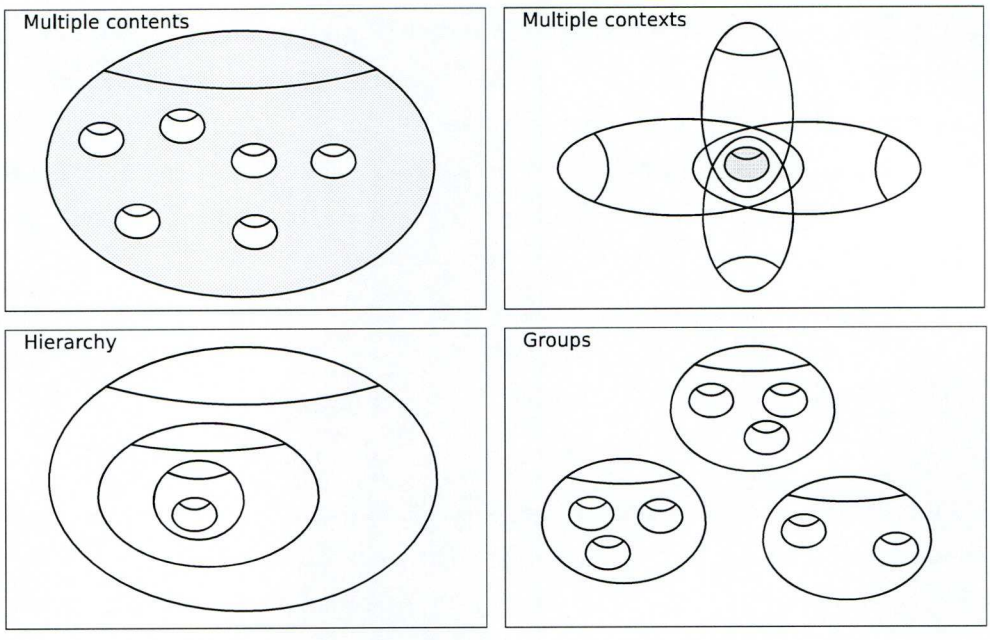


Figure 4.1: A selection of possible organisation structures.

agent may be a member of another agent's *content* and *context* sets simultaneously and cyclical relationships are not prohibited. This allows for a range of structures, from the *transient* to the *permanent*. From the above it is clear that there is no enforced distinction between an agent and an agent organisation. All are agents; all may be treated similarly.

Finally, it is essential that the agent's internal behaviour, be it a program or a specification, have direct access to both the *content* and *context* sets. As we will see below, this allows each agent to become more than just a 'dumb' container. It can restructure, share information and behaviour with, and control access to its *content*.

Intuition

This approach is simple: everything is an agent and every agent has the potential to contain other agents. Yet it can also be comprehensive in its ability to accommodate multi-agent concepts. For example, we can think of several basic varieties of agent [60, 52] (where $\emptyset$ represents the empty set):

- A simple agent: $contents = \emptyset$.
- A simple 'container' context: $behaviour = \emptyset$.
- An independent agent: $context = \emptyset$.
- A more complex context: $contents \neq \emptyset$ and $behaviour \neq \emptyset$.

In this final variety we have the possibility for the agent/context to move beyond simply being a container and to exhibit behaviour/control in its own right. As we will see in the examples below, this is very useful for a variety of complex scenarios. In particular, an agent can only directly communicate with members of its *contents* or *contexts*; it cannot cooperate with arbitrary agents outside these.[2] Since communication must pass through a context, and since contexts are themselves agents potentially with behaviour, then communications can be modified, blocked, duplicated, re-directed, etc., if required by the context's specification.

While it may seem counter-intuitive for an organisation to have beliefs and goals, many of the models surveyed in Section 2.6 required team constructs such as tasks or goals that can naturally be viewed as belonging to a team/group agent. Some also required *control agents* to manage role assignment and communication which in this framework can be handled by the containing agent itself if so desired. On the other hand it is possible to distinguish between agents (with empty *content*) and organisations (with non-empty *content*) and for a programmer to exclude certain constructs from organisations in order to allow an organisation-centred approach, if required.

This model of contexts as agents and agents as contexts has been developed elsewhere (e.g. [60]) and a correspondence with some aspects of Milner's bigraphs [93, 94] has been established [76]. The remainder of this chapter outlines the intended semantics of this approach and demonstrates its ability to represent a wide range of group, team and organisational structures from the multi-agent systems domain.

Semantics

Access to the *context* and *content* sets is given by means of intuitive predicates that allow a number of interesting behaviours to be specified. Those predicates are:

`addToContent/1`	`enterContext/1`	`entered/2`	`in/2`
`removeFromContent/1`	`leaveContext/1`	`left/2`	

In addition to point-to-point messages, agents are able to send a number of multicast messages to the members of their *context* and *content* sets, including what is effectively a broadcast to all agents who have the same context. These three forms of multiple-recipient messaging are fundamental to this agent-organisation abstraction and are depicted in Figure 4.2. This section deals with the formal semantics of the above constructs and messaging.

Message passing is modelled in METATEM by the action predicate `send/2` which, when applied as `send(R,M)`, is true at the moment in time that message `M` is sent to

[2] A third set, *known* has been implemented as a 'list of contacts', which contains all agents that have ever been members of *content* or *context*. There is no specific intuition intended for members of the *known* set, it is intended simply as a programming convenience.

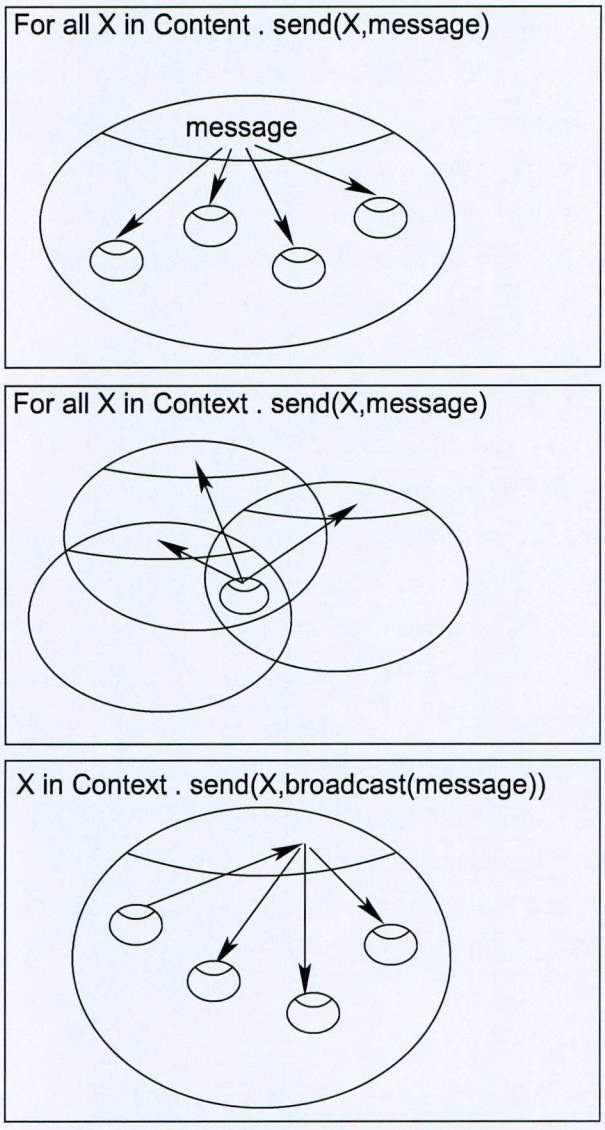


Figure 4.2: The fundamental forms of multicast messaging.

recipient `R`. The complementary `receive/2` predicate becomes true in the recipient agent's state at some time in the future. That is, `receive(F,M)`, indicates receipt of message `M` from agent `F`. The first two diagrams in Figure 4.2 illustrate an agent communicating directly with their *content* and *context* respectively, but more interesting multicasts can be achieved with modified message terms and cooperative members of the *content* or *context* sets. For example (and as depicted by the third diagram in Figure 4.2) a member of a group is able to 'broadcast' a message to all other agents within the group without holding a reference to the group members and retaining its anonymity. Program code for this example and others is given in Section 4.4.

4.3 Operational semantics

For clarity then, this section provides operational semantics for some of the key constructs discussed in this, and the previous, chapters. The semantics describe modifications to the state of a virtual agent that result from a number of operations. The temporal aspects of an agent's operations are intentionally disregarded, not because their semantics have been described in Section 1.6, but because all of the operations described here occur between two contiguous time states, therefore the temporal character of the operations are equivalent.

The purpose of these semantics is to give unambiguous meaning to language constructs. The constructs for which semantics are given affect the sets of terms that an agent maintains in its state, and also the logical evaluation of an agent's temporal states. For these reasons, a notation was chosen that describes the way each construct manipulates the sets (and affects the logical evaluation of temporal states) belonging to a virtual agent.

4.3.1 Notation

The semantics about to be presented uses a state-transition notation, where a state belonging to our virtual agent is denoted by $\mathcal{S}$. However, we differentiate between two types of state; those that correspond directly to an agent's temporal state, $\mathcal{S}$ and $\mathcal{S}^1$, and states that are notional intermediate states occurring between temporal states and denoted by $\mathcal{S}', \mathcal{S}'', \mathcal{S}''', \ldots$. As an agent conceptually executes one or more actions simultaneously between temporal states, no sequence should be inferred from $\mathcal{S}', \mathcal{S}'', \mathcal{S}''', \ldots$. To better illustrate the notation, let us imagine that an agent executes three instructions, $inst^1$, $inst^2$ and $inst^3$, in temporal state S, then apply the semantics of each instruction to acquire three intermediate states

$$\begin{aligned} \mathcal{S} &\xrightarrow{inst^1} \mathcal{S}' \\ \mathcal{S} &\xrightarrow{inst^2} \mathcal{S}'' \\ \mathcal{S} &\xrightarrow{inst^3} \mathcal{S}''' \end{aligned}$$

these states are then combined by a function, $Compose$, which checks that $\mathcal{S}' \wedge \mathcal{S}'' \wedge \mathcal{S}'''$ is consistent, and if so, forms state $\mathcal{S}^1$.

$$\mathcal{S}^1 \quad = \quad Compose(\mathcal{S}', \mathcal{S}'', \mathcal{S}''')$$

In addition to the its logical state, an agent contains a number of sets of terms and a function, $set : T \mapsto S$, that maps set names to these sets, and set of messages, $inbox$, where each message is a pair of terms denoted (ag, m) and corresponding to an agent and message respectively. We denote intermediate state changes to these in a similar way, i.e. set' and $inbox'$.

4.3.2 add/2 and remove/2

Addition and removal of terms from a set requires that the set name be in the domain of *set*

$$\mathcal{S} \xrightarrow[setname \in (\text{dom } \mathcal{S}.set),\ \mathcal{S}'.set = \mathcal{S}.set \dagger [setname \rightarrow \mathcal{S}.set(setname) \cup \{t\}]]{add(t,setname)} \mathcal{S}'$$

$$\mathcal{S} \xrightarrow[setname \in (\text{dom } \mathcal{S}.set),\ \mathcal{S}''.set = \mathcal{S}.set \dagger [setname \rightarrow \mathcal{S}.set(setname) \setminus \{t\}]]{remove(t,setname)} \mathcal{S}'' \quad .$$

4.3.3 addToContent/1 and enterContext/1

Unlike the above operations which affect the state of a single agent, these operations change the state of two agents; the executor and one other. Here, the executing agent is denoted by the term ex and the other by ag, the state of agent ag is denoted by $\mathcal{T}$ and $\mathcal{T}'$ and the state of the executing agent, ex is again denoted by $\mathcal{S}$ and $\mathcal{S}'$. These operations perform a synchronised modification of the *content*, *context* and *known* sets of two agents, such that when the executor agent adds the other to its *content* set, a reciprocal addition to the other's *context* set is made.

$$\mathcal{S} \xrightarrow[\mathcal{S}'.set = \mathcal{S}.set \dagger [content \rightarrow \mathcal{S}.set(content) \cup \{ag\},\ known \rightarrow \mathcal{S}.set(known) \cup \{ag\}]]{ex.addToContent(ag)} \mathcal{S}'$$

$$\mathcal{T} \xrightarrow[\mathcal{T}'.set = \mathcal{T}.set \dagger [context \rightarrow \mathcal{T}.set(context) \cup \{ex\},\ known \rightarrow \mathcal{T}.set(known) \cup \{ex\}]]{ex.addToContent(ag)} \mathcal{T}'$$

$$\mathcal{S} \parallel \mathcal{T} \xrightarrow{ex.addToContent(ag)} \mathcal{S}' \parallel \mathcal{T}'$$

Similarly for entering a context:

$$\mathcal{S} \xrightarrow[\mathcal{S}'.set = \mathcal{S}.set \dagger [context \rightarrow \mathcal{S}.set(context) \cup \{ag\},\ known \rightarrow \mathcal{S}.set(known) \cup \{ag\}]]{ex.enterContext(ag)} \mathcal{S}'$$

$$\mathcal{T} \xrightarrow[\mathcal{T}'.set = \mathcal{T}.set \dagger [content \rightarrow \mathcal{T}.set(content) \cup \{ex\},\ known \rightarrow \mathcal{T}.set(known) \cup \{ex\}]]{ex.enterContext(ag)} \mathcal{T}'$$

$$\mathcal{S} \parallel \mathcal{T} \xrightarrow{ex.enterContext(ag)} \mathcal{S}' \parallel \mathcal{T}'$$

4.3.4 in/2

Satisfaction of $in(ag, content)$ requires that ag be a member of *content* in state $\mathcal{S}$, and does not change the agent's state.

$$\mathcal{S} \xrightarrow[setname \in \text{dom } \mathcal{S}.set,\ t \in \mathcal{S}.set(setname)]{\mathcal{S} \models in(t,setname)} \mathcal{S}' \quad \text{where} \quad \mathcal{S} = \mathcal{S}'$$

4.3.5 `removeFromContent/1` and `leaveContext/1`

By default, either agent has the ability to remove a structural relationship. These operations are similar to their counterparts for creating the relationships, only no modification of the *known* set is made.

$$\frac{\mathcal{S} \xrightarrow{ex.removeFromContent(ag)} \mathcal{S}'}{\mathcal{S}'.set = \mathcal{S}.set \dagger [content \rightarrow \mathcal{S}.set(content) \backslash \{ag\}]}$$

$$\frac{\mathcal{T} \xrightarrow{ex.removeFromContent(ag)} \mathcal{T}'}{\mathcal{T}'.set = \mathcal{T}.set \dagger [context \rightarrow \mathcal{T}.set(context) \backslash \{ag\}]}$$

$$\mathcal{S} \parallel \mathcal{T} \xrightarrow{ex.removeFromContent(ag)} \mathcal{S}' \parallel \mathcal{T}'$$

And similarly for leaving a context:

$$\frac{\mathcal{S} \xrightarrow{ex.leaveContext(ag)} \mathcal{S}'}{\mathcal{S}'.set = \mathcal{S}.set \dagger [context \rightarrow \mathcal{S}.set(context) \backslash \{ag\}]}$$

$$\frac{\mathcal{T} \xrightarrow{ex.leaveContext(ag)} \mathcal{T}'}{\mathcal{T}'.set = \mathcal{T}.set \dagger [content \rightarrow \mathcal{T}.set(content) \backslash \{ag\}]}$$

$$\mathcal{S} \parallel \mathcal{T} \xrightarrow{ex.leaveContext(ag)} \mathcal{S}' \parallel \mathcal{T}'$$

4.3.6 Message passing

When an agent sends a message, m, the recipient agent's in-box is updated before its next reasoning cycle. Again, the agent executing the *send* predicate is denoted by ex, and the receiver as ag. Their states are denoted by $\mathcal{S}$ and $\mathcal{T}$ respectively.

$$\frac{\mathcal{S} \xrightarrow{ex.send(ag,m)} \mathcal{S}'}{ag \in \mathcal{S}.set(known)}, \quad \mathcal{S} = \mathcal{S}'$$

$$\frac{\mathcal{T} \xrightarrow{ex.send(ag,m)} \mathcal{T}'}{(ag,m) \in \mathcal{T}'.inbox}$$

$$\mathcal{S} \parallel \mathcal{T} \xrightarrow{ex.send(ag,m)} \mathcal{S}' \parallel \mathcal{T}'$$

For each message received a corresponding predicate is satisfied in the next state,

$$\frac{\mathcal{T} \xrightarrow{\forall (Ag,M) \in \mathcal{T}.inbox} \mathcal{T}'}{\mathcal{T}' \models receive(Ag,M)}$$

and the in-box is emptied in each temporal state.

4.3.7 Negated built-in predicates

The evaluation of standard predicates, grounded, not-grounded, positive or negative, follows the open-world intuition. That is, any negative non-ground predicates are considered trivially true. Built-in predicates however are treated differently. For instance,

an agent can access its entire in-box, it is, in effect, a closed-world. Therefore, the predicate $\neg receive(Ag, M)$ is not considered trivially true (based upon the notion that somewhere in the world there is an agent that has not sent a message), but instead is only considered true if no agent has sent a message (the in-box is empty).

$$\mathcal{S} \xrightarrow[\mathcal{S}.inbox = \emptyset]{\mathcal{S} \models \neg receive(Ag,M)} \mathcal{S}', \quad \mathcal{S} \xrightarrow[\mathcal{S}.inbox \neq \emptyset]{\mathcal{S} \not\models \neg receive(Ag,M)} \mathcal{S}'$$

$$\text{and } \mathcal{S} = \mathcal{S}' .$$

This is the case for all built-in predicates.

4.4 Representing organisations

In this section we aim to demonstrate how the abstractions found in many of the approaches to agent organisation surveyed in Section 2.6, can be represented appropriately and intuitively using the METATEM language and the *content/context* extensions we have described. Table 4.1 lists these abstractions.

	Joint intentions	Roles	Joint beliefs	Joint commitments	Global goals	Communication	Organisation/Team abstraction	Mutual beliefs	Social structure	inter-[team|group] relationships	Sub-teams	Social commitment	Groups	Interactions/scenes	Task structure	Organisational centred	Agent centred
Cavedon	✔	✔		✔	✔			✔			✔						✔
Cohen & Levesque	✔			✔		✔		✔									✔
Esteva et al		✔			✔	✔	✔		✔			✔		✔	✔	✔	
Ferber		✔				✔	✔		✔				✔		✔	✔	
Hübner		✔			✔	✔	✔		✔	✔	✔	✔	✔		✔	✔	
Pynadath & Tambe		✔	✔			✔	✔		✔						✔	✔	
Tidhar	✔	✔			✔		✔		✔	✔	✔	✔				✔	

Table 4.1: Multi-agent organisation concepts.

4.4.1 Sharing information

Shared beliefs

Being a member of all but the least cohesive groups requires that some shared beliefs exist between its members. Making the contentious assumption that all agents are honest and that joining the group is both individual rational and group rational, let

agent i hold a belief set BS_i. When an agent joins a group[3] j it receives beliefs BS_j from the group and adds them to its own belief base (invoking its own belief revision mechanism in case of conflicting beliefs). The agent in receipt of the new beliefs may or may not disseminate them to the agents in its content, depending on the nature and purpose of the group. Once held, beliefs are retained until revised.

Joint beliefs

Joint beliefs are stronger than shared beliefs. To maintain the levels of cohesion found in teams each member must not only believe a joint belief but must also believe that its team members also believe it. Let us assume the agent is capable of internal actions such as $addBelief(Belief, RelevantTo)$ adding *Belief* to its belief base, and recording the context that *Belief* is relevant to, and $removeBeliefs(Context)$. Upon joining a group, an agent is supplied the beliefs relevant to that context, which it stores in its belief base along with the context in which they hold. This behaviour is captured in the rule below.

```
receive(From, membershipConfirmation(BeliefSet))  &
~size(BeliefSet,0), &
(Belief in BeliefSet)
  => NEXT addBelief(Belief,From);
```

The presence of such *context* meta-information can be used to specify boundaries on agent deliberation, thus mitigating the complexity caused by introducing another variable. When leaving a *context* an agent might either choose to drop the beliefs relevant to that *context* or to retain them.

Note METATEM does not have a dedicated belief revision process other than the inherent prevention of logical inconsistencies. However, by the creation of a custom ability, for example, a developer can provide an agent with arbitrary logical or non-logical belief revision functions. In fact, this was the mechanism used to create the `addBelief` action from the example above.

Shared capabilities

Let agent Ag_i have a goal G, for which a plan P exists. However, Ag_i does not have plan P and therefore must find an agent that does. Two options available to Ag_i are to find an agent Ag_j, who has P, and either: request that Ag_j carries out the plan; or request that Ag_j sends P to Ag_i so that Ag_i can carry out the plan itself. The first possibility suggests a closer degree of co-operation between agents i and j, perhaps

[3]Let us refer to such an agent as a *group* to distinguish it from the agent within its *content* .

even the sub-ordination of agent j by agent i. Whereas, in the second possibility, agent i benefits from information supplied by j.

In the first scenario we might envisage a group in which a member (or the group agent itself) asks another member to execute the plan. In the second case, we can envisage agents i and j *sharing* a plan. This second scenario is typical if groups are to capture certain capabilities — agents who join the *content* of such a group agent are sent (or at least can request) plans shared amongst the group. Either scenario can be modelled using our approach.

4.4.2 Joint intentions

An agent acting in an independent self-interested way need not inform any other entity of its beliefs, or changes to them. On the other hand, an agent who is working, as part of a team, towards a goal shared by itself and all other members of the team has both an obligation and a rational interest in sharing relevant beliefs with the other team members [24]. This gives an agent a *persistent* goal with respect to a team. Such that the agent must intend the goal whilst it is the team's mutual belief that the goal is valid (not yet achieved, achievable and relevant) — it must not give up on a goal nor assume the goal has been achieved, independently. The implications of this impact on agent's individual behaviour when it learns, from sources external to the group, that the goal is no longer valid. In such a situation the team/group agent maintains its commitment to the invalid goal but informs its team members of the antecedent(s) that lead it to believe the goal is invalid. Only when the agent receives confirmation that the entire team share its belief does it drop its commitment.

Extension of an agent's attributes with an intention that reflects the strength of relationship between team members may be a natural way to implement this concept but is likely to involve undesirable modification of an agent's internal architecture. Also, it is far from clear that strong notions of co-operation such as joint intentions can be practically implemented without an agent relinquishing some autonomy to an external entity of some kind, whatever it may be called; agent, team or context. We believe our grouping approach is sufficient to implement joint intentions, mutual beliefs, common goals and other strongly co-operative concepts by strengthening the semantics of the `in/2` predicate. By ensuring that all agents' context and content sets are always consistent, that is for two agents a and b, agent b is a member of a's content set if and only if agent a is a member of agent b's context set;[4]

$$in_a(b, content) \iff in_b(a, context) \tag{4.1}$$

Applying this restriction to our extended METATEM allows us to specify strongly co-operative behaviour. For example, consider the scenario given in Figure 4.3, consisting

[4]Clearly this is only possible for a system of homogeneous agents or for one in which the developer is able to specify the behaviour of all agents.

of a team agent T and member agent B. When the new member A joins the team, it accepts goal JI and confirms its receipt of (and commitment to) the joint intention JI. During membership, and until the team agent informs its members that JI should be dropped, the members have a responsibility to maintain the intention and act rationally with respect to it. This may mean informing the team of any information relevant to the jointly held intention;

```
belief(B) &
in(X,context) &
relevantTo(B,X)
  => NEXT send(X,inform(B));
```

Informing the team if it discovers, independently, that the goal has been achieved;

```
achieved(G) &
in(X,context) &
relevantTo(G,X)
  => NEXT send(X,achieved(G));
```

But dogmatically maintaining the intention regardless of any internal beliefs;

```
goal(G) &
in(X,context) &
relevantTo(G,X) &
~receive(X,drop(G))
  => NEXT goal(G);
```

Thus, an agent is obliged to inform its team of beliefs relevant to jointly held intentions and will maintain a goal whilst it remains contextually relevant.

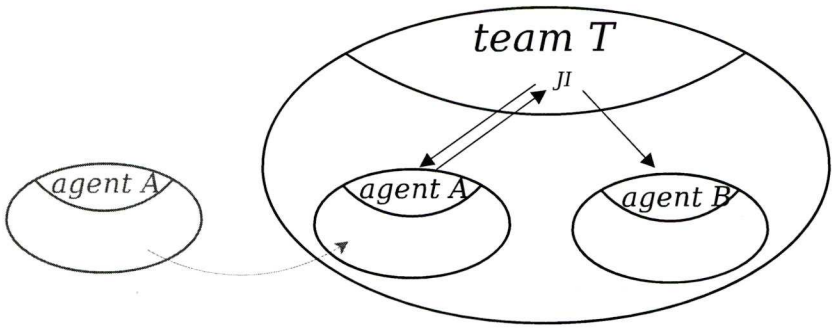


Figure 4.3: Communicating joint intentions upon joining a team.

Agent T. Evaluates group beliefs and communicates both the adoption, and dropping, of intentions when mutual agreement is established. Since T has details of the agents in its *content* and can send messages to interrogate them, it can maintain knowledge of *common* information and behaviours, and reason with this knowledge.

4.4.3 Roles

The concept of a role is a common abstraction used by many authors for a variety of purposes [81, 45, 126], including:

- to define the collective abilities necessary to achieve a global goal;
- to constrain or modify agent behaviour for conformance with team norms; and
- to describe a hierarchy of authority in an organisation of agents and hence create a permissions structure.

Roles are most obviously integrated into our framework as further agents whose *content* is those agents fulfilling the role and whose *context* is the organisation to which the role belongs. However in some cases, in particular strict hierarchies, it may be possible to associate roles directly with the organisational agent. Below we examine a variety of such role types and consider in more detail how each could fit into our model.

Ability roles

Let plan P be a complex plan that requires abilities x,y and z if it is to be fulfilled. An agent A is created (without any domain abilities of its own) to gather together agents that have the necessary abilities. Agent A might generate a new agent in its *content* for each of the abilities required to fulfil plan P.

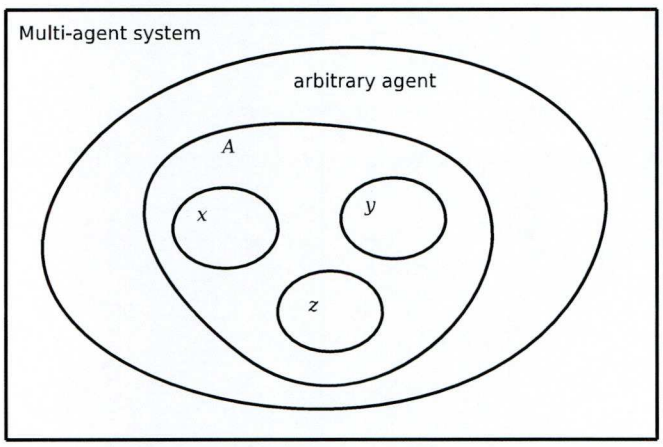


Figure 4.4: Roles according to abilities.

When agents with abilities x, y or z join the *content* of agent A, A adds them to the *content* of the appropriate group (agent), analogous to assigning roles.

A talented agent might become a member of several ability sets. The ability set, itself an agent, may be a simple container or could exhibit complex behaviour of its own. One basic behaviour might be to periodically request (of the agents in its *content*) the execution of its designated ability. Note that, in the case of an ability that is hard to carry out, it may be provident to include many agents with that ability. Similarly, the desired ability might be a complex ability that must be subjected to further planning, resulting in a number of nested abilities.

Roles in society

Joining an institution, organisation or team of agents commonly involves the adoption of the norms of that institution, organisation or team. Whether these norms are expressed as beliefs, goals, preferences or communication protocols, our approach allows them to be transmitted between group members, particularly at the time of joining. For example, if team membership requires that members acknowledge receipt of messages then each new member of a group might be given the new rule (behaviour)

```
receive(From,Message)
  => NEXT send(From, acknowledge(Message));
```

A stronger constraint might require an agent to believe all messages received from its `context`:

```
receive(From,Message) &
in(From,context)
  => addBelief(Message);
```

Without the strength of (4.1), agents cannot be certain that another agent will keep within given constraints or comply with norms of the society, the most it can do is demand formal acknowledgment of its request and a commitment to do so. Group membership can be denied if an agent fails to satisfy the *entry criteria.*

Authority roles

None of the structures discussed usefully reflect hierarchies of authority. Each allow almost arbitrary group membership, with transitive and cyclic structures possible making them unsuitable for expressing a hierarchy of authority, which by its nature must be acyclic with exactly one root.

A common use for such a hierarchy is for creating channels of communication. Our approach to grouping enables communication restrictions for free, as agents may only communicate with their immediate superiors (context), or their direct subordinates (content). Communication to peers (by multicast) can only be achieved by sending a

single *broadcast* message to the agent common to the contexts of the intended recipients. The receiving [superior] agent will, if it deems it appropriate, forward the message to the other agents in its content.

4.4.4 Teams

The team abstraction aims to provide an intuitive approach to the specification of highly coordinated multi-agent behaviour. As such, any concept of a team must make more detailed reference to (required) team and agent abilities than other coordination abstractions. Any team entity would be expected to define

- the roles required to make a well-formed team,
- permitted/restricted communications between team members,
- team resources—knowledge,beliefs,abilities, and
- team goals and plans.

However, we don't advocate the use of a team entity or construct, instead we view an agent and team as equals. Each having identifiable behaviour, a degree of autonomy and a single voice for communication. In METATEM, a team is declared as an agent with a non-empty content set, a number of rules that coordinate the team members, and possibly some behaviour of its own. Consider the example of a professional football team, it plays for a club, against an opponent and aims to win matches in order to maximise profits for the club's shareholders. The team is composed of three sub-teams (defence, midfield and forwards), a coach and a group of substitutes. METATEM encourages the developer to treat each of these entities as agents, consequently the team is declared as follows;

```
football_team {
  context :  club, opponent, shareholders;
  content :  defence, midfield, forwards, manager, substitutes;
}
```

Here, the members of the content set are analogous to the concept of roles. Note that we don't explicitly declare them as roles, nor do we declare that this initial *context*/*content* sets should be fixed throughout the team's life, however it is possible to apply such constraints in rule form. For example, if the a defence is deemed necessary for a functioning team then we would like to specify that $\Box inContent(defense)$ holds. This translates to

```
~in(defence,content) => false;
```

in our normal form.

Defining the team's actions is optional, if team behaviour is considered to be the collective behaviour of its individual members then the agent need not declare any actions, its temporal specification will describe only the internal behaviour of the team such as membership protocols and communication constraints. In our example, we consider the team to be capable of entering tournaments, and reasoning that once entered, it will eventually play a match;

```
action enterTournament : examples.football.EnterTournament ;

enterTournament(X)
  => SOMETIME play_in(X);
```

The team agent can declare (and therefore) disseminate team knowledge in the form of a set of beliefs, that can be given to all members as described in section 4.4.1;

```
set teamBeliefs : {team_name(athletico),league(one),... }
```

We believe that many benefits follow from treating the team and agent as one and the same entity. For instance, any language that support dynamic creation of agents during run-time, will also support the creation of teams formed 'spontaneously'. Also, the ability to substitute agents with (sub)teams in a role, and vice versa, greatly increases the flexibility of design and could be viewed as a way of increasing the scale of a system.

In this chapter, constructs for modelling context-dependent behaviour with the METATEM agent programming language have been described. We have given clear semantics and illustrated their appeal as a flexible approach to a variety of agent organisation abstractions. These organisation abstractions have been captured with a few basic constructs of the METATEM language. None of the organisation concepts or abstractions are built into the language and METATEM does not enforce a particular definition of any of them. Instead, the general notions of content and context are used, along with appropriate and flexible constructs, to capture a variety of agent organisation concepts. In the remaining chapters we aim to demonstrate that this approach is appropriate for the principled specification of pervasive and ubiquitous computing systems, due to its simplicity, flexibility and logical foundations.

Chapter 5

A Common Semantics of Organisation

This chapter concerns the proposal of implementing simple, yet flexible, constructs for extending multi-agent programming languages based on the BDI model. It forms an argument for using the content and context components described earlier in the thesis as a general abstraction to facilitate agent-organisation in many logic-based BDI languages. We argue that the two sets, along with a *constraint* construct, provide sufficient expressive power to allow us to represent, simply and with semantic clarity, a wide range of organisational structures for multi-agent systems. It should be noted, that this chapter represents an addendum to the project's thesis and is derived from a collaboration with Michael Fisher and Louise Dennis.

The chapter begins by outlining the motivation for these proposals. It then informally introduces the approach and provides its formal semantics, through modification of an operational semantics based on the core of AgentSpeak, 3APL and METATEM. In addition, we provide illustrative examples by simulating both constraints and content/context sets within the Jason interpreter for AgentSpeak.

5.1 Motivation

When researchers and developers experimented with agent-oriented languages and used them for a wider variety of applications it became clear that *open* multi-agent systems did not scale well without a further abstraction to capture the working relationships between agents, groups of agents and their environment [61, 105]. Furthermore, only a cursory study of human societies is needed to realise that increased levels of productivity and efficiency are realised by societies with effective frameworks that encourage cooperative behaviour amongst their populations. The study of agent interaction, cooperation and organisation is therefore of current interest in the agent research community [98, 69] but, although a wide variety of BDI languages have been developed [15], few have strong and flexible mechanisms for *organising* multiple agents, and those that do

provide no agreement on their organisational mechanisms. Thus, while BDI languages have converged to a common core relating to the activity of individual agents [35], no such convergence is apparent in terms of multi-agent structuring and organisation.

5.1.1 Proposal

Before looking at the detail of our proposals let us recall the agent organisation techniques, reviewed in Chapter 2, that we aim to support.

Cohen and Levesque [24] argue that a team of agents should be modelled with new (logical) concepts of *joint intentions*, *joint commitments* and *joint persistent goals* to increase the cohesion of team members. Tidhar [124] introduced the concept of *team-oriented programming* that employs a weaker notion (weaker than Cohen and Levesque's) of joint intentions and joint goals. Ferber *et al.* [44] present a model for designing multi-agent systems in terms of *agents*, *roles* and *groups*, where agents and groups are proposed as distinct first class entities. Sierra *et al.* [43, 126] formalised the *institutions* abstraction founded on institutional norms.

In this chapter we consider extending basic BDI languages with simple, yet powerful, constructs that allow the development of a wide range of organisational structures. Thus, in the following section, we introduce the concepts behind the new constructs, in particular showing how they relate to typical BDI language semantics. To clarify this further, in Section 5.3, we provide the core semantics of a subset of AgentSpeak [109, 12] incorporating the new concepts; we call this language AGENTSPEAK$^-$. To show how these concepts can be used, in Section 5.4, we outline how a variety of organisational structures can be expressed using these simple constructs, present several case studies, and even provide some implementations within AgentSpeak.

We begin by introducing the concepts; we do this by first considering the core operational aspects of BDI languages, describe some agent-organisational abstractions and then show how our new concepts affect agent operation.

5.2 Introducing the concepts

Although all BDI languages have a family resemblance, their syntax and semantics can vary immensely. We therefore use a loose unifying framework for our discussion into which we believe most BDI languages will fit,[1] though not always elegantly.

Our semantic framework assumes that a BDI language specifies the behaviour of an agent in terms of the agent's current state, $\mathcal{S}$, which changes over time and a fixed specification, $\mathcal{SP}$, which does not. Thus, an agent is viewed as a tuple $< \mathcal{S}, \mathcal{SP} >$. $\mathcal{S}$ consists, amongst other things, of a set of beliefs, *Bel*. The BDI programming language

[1] Indeed, in [35] such a framework was used to provide a common semantic basis for 3APL, AgentSpeak, METATEM, etc.

then has a process for determining whether a given belief b follows from the current state which we will write as $\mathcal{S} \models b$, since these are often logical mechanisms.

The BDI programming language has a specific operation, **select_instruction**, which acts on the state according to the specification in order to determine the next instruction to be executed and another, **modify**, which modifies the state according to the specification and the selected instruction. The execution of an agent can therefore be viewed as repeated application of the transition rule

$$< \mathcal{S}, \mathcal{SP} > \rightarrow < \textbf{modify}(\mathcal{SP}, \mathcal{S}, \textbf{select\_instruction}(\mathcal{SP}, \mathcal{S})), \mathcal{SP} > \quad . \tag{5.1}$$

We assume that both $\mathcal{S}$ and $\mathcal{SP}$ are made up of a number of sets or stacks (e.g., of beliefs) and use the notation $\mathcal{S}[S_1 \setminus S_2]$ to indicate the state $\mathcal{S}$ in which the set S_1 has been replaced by S_2.

Note

This framework should not be interpreted as assuming that a given BDI language has *explicit* constructs for **select_instruction** and **modify**, but that most BDI languages can be expressed in terms of the operation of appropriate versions of these functions. Indeed, the fact that languages implement their own interpretations of the BDI paradigm's concepts (such as plan selection, intention choice and belief revision), requires that our semantics abstract away the semantics of these operations. This is the reason for our choice of semantic framework, which includes a number of abstract functions to denote language-specific operations. Clearly, if a given language cannot be expressed in terms of these functions then the semantics that follow cannot be applied to that language.

We also assume that a BDI language contains a set of plans (or rules), *Plans*, which are used by the **select_instruction** operation. These plans may either be a part of $\mathcal{S}$ or $\mathcal{SP}$. We assume such plans are *triggered* in some fashion by $\mathcal{S}$. In some cases they are triggered by the composition of the beliefs (e.g., METATEM [52]), in some by the goals (e.g., 3APL [75, 29]) and in some by explicit trigger events (e.g., Jason [12] interpreter for AgentSpeak [109]).

To simplify matters, we use an abstraction of a plan, describing it using a horn-clause notation, as

$$t \leftarrow \{g\} b \quad .$$

Thus, plans comprise; a *trigger*, t; a *guard* (checked against the agent's beliefs), g; and a *body*, b, which specifies an instruction (or sequence of instructions) to be executed. In languages where only beliefs are used to trigger plans this can be written as

$$\top \leftarrow \{g\} b \quad .$$

In order to trigger plans, the language requires some component of the current state $\mathcal{S}$ which activates the trigger. We treat this as a set, T, and write the triggering process as $T \models_t t$.

Finally, we will use the notation $Ag \models_a p$ to indicate that a plan, p, is applicable for an agent, Ag. The semantics of this for a basic[2] BDI agent is

$$\frac{\textbf{app\_cond}(t \leftarrow \{g\}b, Ag)}{Ag \models_a t \leftarrow \{g\}b} \tag{5.2}$$

where **app_cond** are the agent language's applicability conditions. In most languages

$$\textbf{app\_cond}(t \leftarrow \{g\}b, Ag) = ((T \models_t t) \wedge (\mathcal{S} \models g)) \ .$$

Notes

Again we do not necessarily expect these operations associated with plans to be explicit in the languages, for example T may be a stack of goals and $T \models_t g$ may be the process of matching the head (or prefix) of this stack. There may also be other applicability checking processes within the language, for example, the applicability of actions — we represent all of these within $Ag \models_a$. Application of a plan results in an instruction to modify the state either directly (when $+b$ appears in the body of the plan and is an instruction to add b to *Bel*, for example) or indirectly, when the body of the plan is integrated into an *intention* or other part of the state which is subsequently used for further planning or to govern subsequent actions and changes of belief.

This is the case for METATEM, in which plans do not take the form $t \leftarrow \{g\}b$ and plan selection does not select only one plan. Thus, to provide an operational semantics of METATEM in the style of rule-base BDI languages, the general SNF rule form of *antecedent* $\Rightarrow$ *consequent* accommodates trigger events and beliefs in the antecedents, as follows

$$(t \ \wedge \ g) \Rightarrow b \quad ,$$

and, to accommodate METATEM's synchronous execution of multiple plans we provide semantics that makes one state transition per 'rule', then compose two or more of these to form a single temporal transition.

Given the above, we below consider the two aspects we wish to introduce to general BDI programming languages. The only restrictions it puts on any underlying language is that, as in most BDI-based languages (and as described above), there are logical mechanisms for explicitly describing *beliefs* and *goals*, and possibly *plans* and *intentions*. Of course, a form of message-passing between agents is also required. These features are standard in most agent languages.

[2]I.e., a BDI agent whose semantics has not been modified with the constructs we describe later.

5.2.1 Content and Context Sets

Assuming that the underlying language can describe the behaviour of an agent as above, we now extend the concept of agent with the sets described in Chapters 3 and 4, named `Content` and `Context`.[3] The concept and intuition behind these sets have been covered in Chapter 4 but to recap, and to give them a distintly organisational flavouring, we might view the members of an agent's `Content` as those agents that it has recruited, and the members of its `Context` as those agents it has been recruited by. Alternatively, an agent might be used to represent a location and the members of its `Content` the agents at that location.

The proposals prohibit cyclical structures and require that all structural changes occur with the consent of those agents whose `Content` or `Context` sets are affected.

Semantics

The simplicity of the above approach allows us to provide a few general operational rules for managing the content and context sets. We extend the agent's state, $\mathcal{S}$, with a content set, (Cn), and a context set, (Cx), and add four new instructions into the language $+ag^{Cn}$ (add ag to the content set), $-ag^{Cn}$ (remove ag from the content set) and $+ag^{Cx}, -ag^{Cx}$ for adding and removing agents from the context set. We also add four new constructs into the trigger component, T:

$$entered\_content(ag)$$
$$entered\_context(ag)$$
$$left\_content(ag)$$
$$left\_context(ag)$$

Add two new constructs into our language of guards:

$$in\_content(ag)$$
$$in\_context(ag)$$

We then extend the **modify** operation with the rules:

$$\textbf{modify}(\mathcal{SP}, \mathcal{S}, +ag^{Cn}) = \mathcal{S}[Cn \setminus Cn \cup \{ag\}, T \setminus T \cup entered\_content(ag)] \quad (5.3)$$

$$\textbf{modify}(\mathcal{SP}, \mathcal{S}, -ag^{Cn}) = \mathcal{S}[Cn \setminus Cn - \{ag\}, T \setminus T \cup left\_content(ag)] \quad (5.4)$$

and two analogous ones for the context. These rules extend both the state's content/context and the trigger set, T. This allows plans to be triggered by changes in these sets. (e.g., plans of the form

$$entered\_content(Ag) \leftarrow \{in\_content(Ag)\} send(self, Ag, plan)$$

[3]The third set, `Known`, is omitted for brevity as it is not essential to the requirements of the proposal. Its inclusion would, of course, provide convenience, as does its inclusion in Concurrent METATEM.

may be written which are triggered by the addition of a new agent Ag to the content set, into sending that agent a plan).

We also extend the belief inference process to include checking membership of Cn and Cx:

$$\frac{ag \in Cn}{\mathcal{S} \models in\_content(ag)} \tag{5.5}$$

$$\frac{ag \in Cx}{\mathcal{S} \models in\_context(ag)} \ . \tag{5.6}$$

It should be noted that in many languages it may be possible to streamline these extensions (e.g., by merging the triggering of plans and the update of content/context sets – see Section 5.3).

5.2.2 Constraints

The second basic component we suggest is necessary for many meaningful multi-agent structures is that of *constraints*. A constraint consists of additional guards that may be appended to plans/rules and actions and is typically provided by an agent's context. This, for example, allows permissions to be modelled.

Semantics

As with groups we extend the agent's state, $\mathcal{S}$, with a constraint set, C. C is treated as a set of pairs of a trigger and a guard, written $[t \Rightarrow g]$. Depending on the language, it may be desirable to add other pairs to this set, for instance if actions may have guards and there is an applicability process for actions then action/guard pairs may also be useful within constraints. Again, we add new instructions into the language $+new\_constraint^C$ (add $new\_constraint$ to C) and $-new\_constraint^C$ (remove $new\_constraint$ from C), which are analogous to the previous add/remove operators. We then extend our applicability checking process, $Ag \models_a$ to

$$\frac{\forall [t \Rightarrow g'] \in C.\ \mathcal{S} \models g' \quad \textbf{app\_cond}(t \leftarrow \{g\}b, Ag)}{Ag \models_a t \leftarrow \{g\}b} \ . \tag{5.7}$$

So, in many languages, this becomes

$$\frac{\forall [t \Rightarrow g'] \in C.\ \mathcal{S} \models g' \quad T \models_t t \quad \mathcal{S} \models g}{Ag \models_a t \leftarrow \{g\}b} \ . \tag{5.8}$$

Similar modifications can be made to the operational semantics of action applicability (internal or external) and any other relevant components of $\mathcal{S}$ and $\mathcal{SP}$.

It should be noted that constraints make relatively little sense in a single agent environment (where guards on plans and actions are sufficient) it is only in a multi-agent environment where a member of `Context` may wish to provide guards to a pre-existing plan or action that such constraints become useful.

Before going on to providing the semantics of a more comprehensive language (in Section 5.3), we first consider the properties of such semantic extensions.

5.2.3 Properties of groups and constraints

In addition to the generic operational semantics for groups and constraints we present here some properties that ideally any system implementing them should obey. We discuss when these hold in a system that implements these concepts using our suggested rules.

Firstly one agent should believe that another is in its `Content/Context` if, and only if, that agent is *actually* in its `Content/Context`. We express this as:

$$CONTAINS(ag) \Rightarrow BEL(in\_content(ag)) \tag{5.9}$$

$$CONTAINED\_BY(ag) \Rightarrow BEL(in\_context(ag)) \tag{5.10}$$

$$BEL(in\_content(ag)) \Rightarrow CONTAINS(ag) \tag{5.11}$$

$$BEL(in\_context(ag)) \Rightarrow CONTAINED\_BY(ag)\ . \tag{5.12}$$

For the operational semantics presented above we interpret $CONTAINS(ag)$ as $ag \in Cn$, $CONTAINED\_BY(ag)$ as $ag \in Cx$ and $BEL(\phi)$ as $\mathcal{S} \models \phi$.

Let us assume that the the formulae $in\_content(ag)$ and $in\_context(ag)$ are "reserved" in an implementation, i.e., such formulae can not appear in the belief base either when an agent is initialised or through any belief revision process and that there is no way they can be inferred through belief inference except by the use of (5.5) and (5.6). (Many BDI languages have mechanisms for reserving key-words which could be extended for this purpose.) If this is the case then (5.9–5.12) follow directly from rules (5.5) and (5.6). If it is not possible to restrict the formulae that an agent might believe (e.g., it will accept any formula as a belief if sent it by a trusted external agent) then any system adopting our operational semantics only satisfies (5.9 and 5.10), unless additional safeguards are implemented.

Turning to constraints, we would expect any well-behaved system implementing constraints to satisfy

$$(Ag \models_a P) \Rightarrow ((Ag \models_a P) \wedge (C = \emptyset)) \tag{5.13}$$

i.e., if a plan is applicable given some constraints, then it is also applicable if there are no constraints. In our operational semantics this follows from (5.7) if we observe that when $C = \emptyset$ the condition $\forall[t \Rightarrow g'] \in C.\mathcal{S} \models g'$ reduces to $\top$ and that C is not referred to elsewhere in the rule.

Rao and Georgeff [113] state a number of interesting properties they suggest BDI languages might wish to satisfy and it would be tempting to examine some of these in relation to groups (in particular those relating intentions and beliefs (with $INTEND(\phi)$ interpreted as $\phi \in T$)). However this work assumes that intentions are expressed as temporal formulae and that belief inference includes temporal and causal reasoning. Our triggers are *not* expressed in this way and in fact may include formulae (such as

entered_content(ag)) which refer to events that have occurred rather than states of the world the agent wishes to bring about.

As mentioned previously, an agent's internal behaviour, be it a program or a specification, must have direct access to both the `Content` and `Context`, allowing each agent to become more than just a 'dumb' container. An agent can then provide access to, provide services for, and share information or behaviours with, its `Content`, as is demonstrated by Fig. 5.1; here agent j moves into the separate context of agents i and k (perhaps i represents an auctioneer agent who provides j with the bidding rules, whilst k is the agent on whose behalf j is bidding). Our proposals encourage the sharing of plans, beliefs and constraints as structural changes take place but also allow the dissemination of new knowledge. Indeed we can state the following, very general, result.

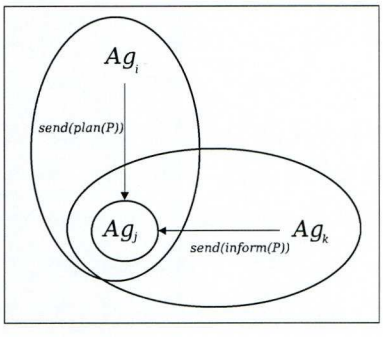


Figure 5.1: Sharing plans and information.

Theorem 1 *If agent A moves into a new context C and*

- *the context agent C, is willing to send plans/beliefs/constraints/etc to A, and*
- *agent A incorporates these plans/beliefs/constraints/etc sent from its new context,*

then A has the new plans/beliefs/constraints/etc provided by its new context.

Aside. There is an obvious counterpart of Theorem 1 whereby A can *ask* its context for information (plans/beliefs/etc). Once it moves into a new context then A has access to the new information/capabilities provided by its context.

Theorem 1 above has many caveats! However, these mainly cover situations where agents choose not to cooperate. In a cooperative scenario, where an agent provides plans/beliefs/constraints/etc to any new members of its content, and where agents accept those items from their new context, then Theorem 1 says that an agent effectively has the information and capabilities provided by its context (in addition to its own).

Importantly, this is seamless. The particular example of *constraints* is informative. Constraints effectively prohibit certain planning choices. Thus, through Theorem 1 we know that an agent with certain choices (e.g. of how to achieve a goal) will inherit the constraints (restrictions) from its context. If the agent is in multiple contexts, the agent must make choices satisfying *all* the constraints received from its contexts. Effectively, the agent is constrained by the union of all its contexts and so its behaviour must follow the intersection of behaviours allowed by each context.

This aspect is exhibited in the *cookery* example in Section 5.4.3, but is also closely linked to organisational aspects such as *norms* in that the agent's choices are modified by the contexts (organisations) in which it finds itself.

5.3 A simple BDI language: AgentSpeak$^-$

We will conclude our discussion of formal semantics with a simple example showing how our framework provides a practical methodology for extending existing BDI languages. Let us consider an extremely simple agent programming language based on AgentSpeak [109, 12]; we will call this language AgentSpeak$^-$.

Syntax

Our language uses ground first-order formulae for beliefs, actions and goals. A plan is a triple of a goal, a guard and a stack of instructions (called here *deeds* following AIL [35]). An Agent is a triple of a set of beliefs, a stack of deeds and a set of plans. This is shown in Fig. 5.2.

$$
\begin{aligned}
belief &:= \text{Ground first-order formula} \\
action &:= \text{Ground first-order formula} \\
goal &:= \text{Ground first-order formula} \\
plan &:= goal : set(belief) \leftarrow stack(deed) \\
agent &:= < set(belief), stack(deed), set(plan) > \\
deed &:= action \mid +belief \mid -belief \mid +!goal
\end{aligned}
$$

Figure 5.2: Syntax of AgentSpeak$^-$.

Operational Semantics

An operational semantics for AgentSpeak$^-$ is provided in the form of the four transition rules in Fig. 5.3. In these semantics $do(a)$ is an operation in an agent's interface that causes it to perform the action, a, and then returns a set of messages in the form of deeds, $+!received(sender, \phi)$, which instruct the agent to handle the message ϕ from

agent *sender*. In this language, therefore, perception has to be handled by an explicit *perception* action which then returns messages from the environment as if from another agent. Finally, ';' represents the *cons* function on stacks, '@' represents the join function, and '*random*' indicates random selection of an element from a set.

$$\frac{do(a) = msg}{< B, a; D, P > \rightarrow < B, msg@D, P >} \tag{5.14}$$

$$\frac{}{< B, +b; D, P > \rightarrow < B \cup \{b\}, D, P >} \tag{5.15}$$

$$\frac{}{< B, -b; D, P > \rightarrow < B - \{b\}, D, P >} \tag{5.16}$$

$$\frac{body = random(\{b \mid g : G \leftarrow b \in P \wedge G \subseteq B\})}{< B, +!g; D, P > \rightarrow < B, body@D, P >} \tag{5.17}$$

Figure 5.3: Operational Semantics of AGENTSPEAK$^-$.

Note

This is not intended as a practical example of a BDI language. For a start the language is entirely grounded and makes no use of unification. Secondly the rather crude use of the deed stack to organise both planning and message handling/perception is likely to cause quite strange behaviour in any real agent setting, as no distinction (and therefore priority) is made between beliefs, perceptions or messages.

Extension to the Simple BDI Language

Fig. 5.4 shows how this language fits into our earlier framework. Modifying these semantics according to our content/context and constraints framework now gives us the language semantics shown in Fig. 5.5

In fact this extension can be improved upon based on the details of our languages. For instance we can omit the *entered_content*() and *left_content*() and use $+ag^{Cn}$ and $-ag^{Cn}$ as plan triggers if we like, changing (5.24) to

$$\frac{body = random(\{b| + ag^{Cn} : G \leftarrow b \in P \wedge G \subseteq B \wedge \forall[+ag^{Cn} \Rightarrow G'] \in C.\, G' \subseteq B\}}{< B, +ag^{Cn}; D, Cn, Cx, C, P > \rightarrow < B, body@D, Cn \cup \{ag\}, Cx, C, P >} \tag{5.28}$$

5.4 Using the concepts

We will briefly discuss some examples of the use of constraints and content/context sets (sometimes termed *groups*) in organisational and multi-agent settings. We begin

Framework	AgentSpeak$^-$
$\mathcal{SP}$	P
$\mathcal{S}$	$< B, D >$
T	D
$\mathcal{S} \models b$	$b \subseteq B$
$T \models_t t$	$t = hd(D)$
app_cond$(gl : g \leftarrow b)$	$g \subseteq B$
modify$((B, D), P, +b)$	$(B \cup \{b\}, D)$
modify$((B, D), P, -b)$	$(B - \{b\}, D)$
modify$((B, D), P, ds)$	$(B, ds@D)$
select_instruction$((B, a; D), P)$	$do(a)$
select_instruction$((B, +b; D), P)$	$+b$
select_instruction$((B, -b; D), P)$	$-b$
select_instruction$((B, +!g; D), P)$	$random(\{b \mid p \in P \wedge Ag \models_a p\})$

Figure 5.4: Mapping our Framework to AgentSpeak$^-$.

$$\frac{do(a) = msg \quad \forall[a \Rightarrow G] \in C.\, G \subseteq B}{< B, a; D, Cn, Cx, C, P > \rightarrow < B, msg@D, Cn, Cx, C, P >} \tag{5.18}$$

$$\frac{}{< B, +b; D, Cn, Cx, C, P > \rightarrow < B \cup \{b\}, D, Cn, Cx, C, P >} \tag{5.19}$$

$$\frac{}{< B, -b; D, Cn, Cx, C, P > \rightarrow < B - \{b\}, D, Cn, Cx, C, P >} \tag{5.20}$$

$$\frac{}{< B, +c^C; D, Cn, Cx, C, P > \rightarrow < B, D, Cn, Cx, C \cup \{c\}, P >} \tag{5.21}$$

$$\frac{}{< B, -c^C; D, Cn, Cx, C, P > \rightarrow < B, D, Cn, Cx, C - \{c\}, P >} \tag{5.22}$$

$$\frac{body = random(\{b \mid g : G \leftarrow b \in P \wedge G \subseteq B \wedge \forall[g \Rightarrow G'] \in C.\, G' \subseteq B\})}{< B, +!g; D, Cn, Cx, C, P > \rightarrow < B, body@D, Cn, Cx, C, P >} \tag{5.23}$$

$$\frac{}{< B, +ag^{Cn}; D, Cn, Cx, C, P > \rightarrow < B, +!entered\_content(ag); D, Cn \cup \{ag\}, Cx, C, P >} \tag{5.24}$$

$$\frac{}{< B, -ag^{Cn}; D, Cn, Cx, C, P > \rightarrow < B, +!left\_content(ag); D, Cn - \{ag\}, Cx, C, P >} \tag{5.25}$$

$$\frac{}{< B, +ag^{Cx}; D, Cn, Cx, C, P > \rightarrow < B, +!entered\_context(ag); D, Cn, Cx \cup \{ag\}, C, P >} \tag{5.26}$$

$$\frac{}{< B, -ag^{Cx}; D, Cn, Cx, C, P > \rightarrow < B, +!left\_context(ag); D, Cn, Cx - \{ag\}, C, P >} \tag{5.27}$$

Figure 5.5: AGENTSPEAK$^-$ extended to multi-agents.

by considering a few common aspects of agent organisations, and then examine two case studies in more detail.

The purpose of these examples and case-studies is to both illustrate the capability of our proposals for implementing agent organisation concepts and demonstrate that the proposals can be practically adopted by an existing BDI programming language. This is achieved by demonstrating the concepts of shared belief, permissions and obligations (in Sections 5.4.1 and 5.4.2), by combining these and other concepts into two more elaborate case-studies (in Sections 5.4.3 and 5.4.4) and by using code for the Jason interpreter for AgentSpeak, throughout. We believe that these examples demonstrate that content/context sets can provide a powerful, flexible and intuitive way of handing agent organisation at a level of abstraction and concision that is appropriate for a common semantics of organisations. Note that a more comprehensive review of how many agent organisational approaches can be modelled using our constructs is provided in [74].

5.4.1 Shared beliefs

Being a member of all but the least cohesive groups/organisations requires that some shared beliefs exist between the members. Making the (contentious) assumption that all agents are honest and that joining a group is both individual rational and group rational, let agent i hold a belief set BS_i and assume the programming language contains the instruction *addBelief*(*Beliefs*) with the semantics

$$\textbf{modify}(\mathcal{SP}, \mathcal{S}, addBelief(Bs)) = \mathcal{S}[Bel \setminus Bel \cup Bs].$$

Suppose a (group) agent i has the plan:

$$entered\_content(Ag) \leftarrow \{in\_content(Ag)\}send(i, Ag, inform(BS_i))$$

and agent j has the plan:

$$received(Ag, j, inform(BS_i)) \leftarrow \{in\_context(Ag)\}addBelief(BS_i)$$

taken together these plans mean that if j joins the `Content` of i it gets sent the beliefs BS_i which it adds to its own belief base. This allows shared beliefs to be established.

The agent in receipt of the new beliefs may or may not disseminate them to the agents in its `Content`, depending on the nature and purpose of the group structure. Once held, beliefs are retained until contradicted or revised (for example, on leaving the group). It is worth noting here that these behaviours are merely suggestions of how our proposals can be used to implement shared beliefs, providing the developer has authorship of all agents.

5.4.2 Permissions and obligations

A number of multi-agent proposals include concepts of permissions and obligations [15]. An agent within a group setting may or may not have the permission to perform a particular action or communicate in a particular fashion. This can be easily represented using constraints: for instance if agents in group, G, may not perform action a then the constraint $[a \Rightarrow \bot]$ can be communicated to them when they join G's `Content`.

It should be noted that in order for such a message to be converted into an actual constraint, the receiving agent would also need the plan:

$$received(Ag, i, constrain([a \Rightarrow g])) \leftarrow \{in\_context(Ag)\} + [a \Rightarrow g]^C \ .$$

This design deliberately allows varying degrees of autonomy among agents to be handled by the programmer.

Obligations are where a group member is *obliged* to behave in a particular fashion. This can be modelled if plans are treated as modifiable by the underlying agent language. Obligations can then be communicated as new plans.

```
/*---------------- initial beliefs ---------------- */
cooperative.

/*---------------- rules -------------------------- */
check_constraint(Plan, Arg)
  :- not constraint_fails(Plan,Arg).

/* --------------- basic plans -------------------- */

/* How an agent responds to a group membership invitation */
+!join(Group)[source(Group)] : cooperative
   <- .my_name(Me);
      +context(Group);
      .println("I believe I have the context of ", Group);
      .send(Group, achieve, accept(Me, Group)).
```

Figure 5.6: A simple cooperative agent defined in AgentSpeak.

5.4.3 Case study 1: Cookery agents

We now describe an implementation case study in which we demonstrate the concepts using AgentSpeak and Jason. It concerns a simple *cook* agent who is provided with a number of plans by a chef agent, each for cooking a different meal. The cook's choice of plan is constrained by the `Context` in which it cooks.

Scenario

The chef of a restaurant hires a cook and provides a list of dishes from which the cook is free to choose when asked to prepare a meal. As diners arrive, their preferences are noted and the cook endeavours to choose a meal that satisfies all of the diners. Our cook was implemented as a simple, cooperative agent with the ability to enter the `Context` of other agents but without any domain abilities — it can't cook — see Fig. 5.6.

When hired, the cook agent receives plans for making risotto, steak and pizza. AgentSpeak code defining this behaviour is shown below.

```
+content(Agent)[source(self)]
   <- .print("Sending ", Agent, " plans...");
      .send(Agent, tellHow, "+!cook(risotto)
                              : check_constraint(cook,risotto)
                                <- make(risotto).");
      .send(Agent, tellHow, "+!cook(steak)
                              : check_constraint(cook,steak)
                                <- make(steak).");
      .send(Agent, tellHow, "+!cook(pizza)
                              : check_constraint(cook,pizza)
                                <- make(pizza).").
```

(Note that this sending of plans is triggered by the cook entering its `Content`.) When asked to prepare a meal without the constraints of any diners it prepares risotto; see Fig. 5.7(b). A meat eating diner then imposes their dislike for risotto by providing the cook with the constraint

```
constraint_fails(cook,risotto).
```

Now acting in the context of this meat eater, rather than making risotto, the chef prepares steak; see Fig. 5.7(c). Finally, a vegetarian diner invites the chef to join its `Content` and imposes the constraint `constraint_fails(cook,steak)`, see Fig. 5.7(d). The agent, now a member of three contexts, must decide an appropriate course of action within the supplied constraints — it must not commit to cooking risotto or steak! Thus it is constrained to choose to prepare pizza; see below.

```
+content(Agent)[source(self)]
   <- .print("Sending ", Agent, " my constraints");
      .send(Agent, tell, constraint_fails(cook,steak)).
```

Full execution output for this example is given below.

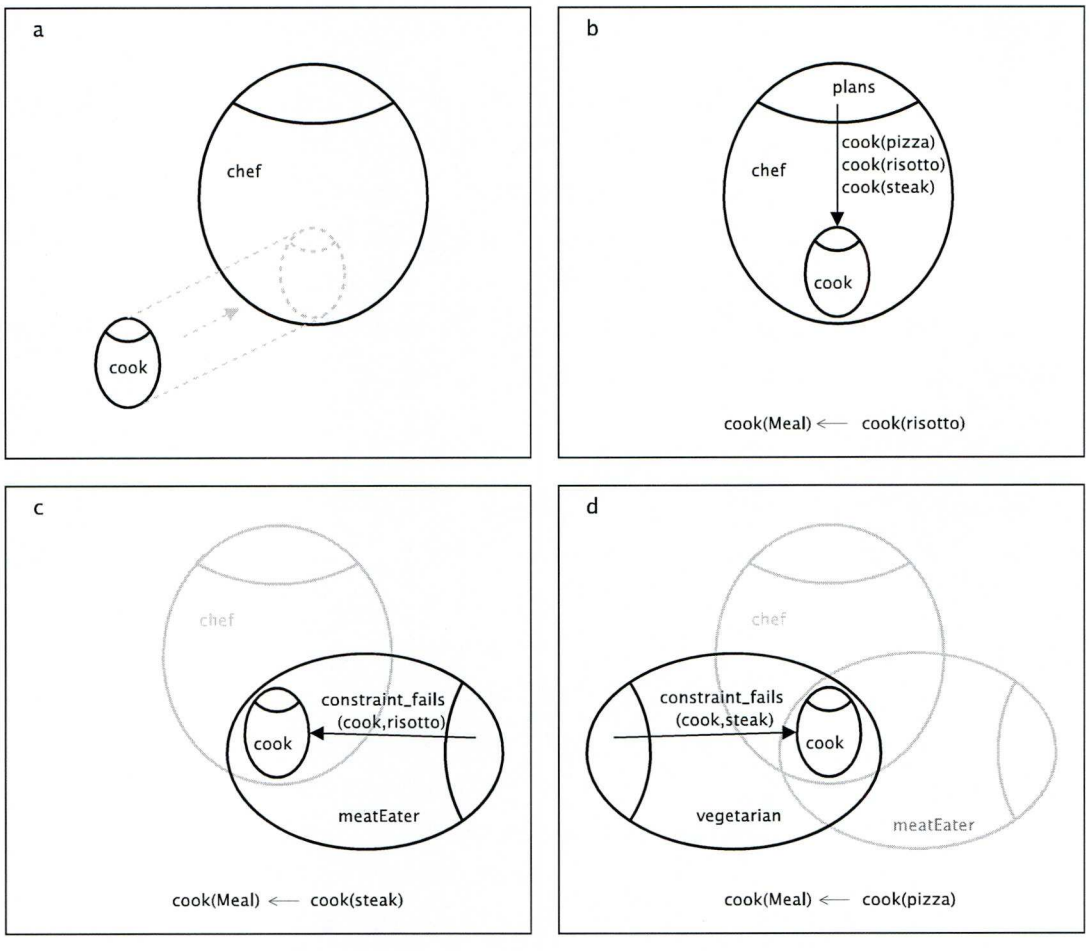


Figure 5.7: A cook with multiple constraints.

```
[chef] saying: inviting cook to join my content
[cook] saying: I believe I have the context of chef
[chef] saying: Sending cook plans...
[chef] saying: I consider cook to be in my content
[cook] doing: make(risotto)
***cook is making risotto***
[meatEater] saying: inviting cook to join my content
[cook] saying: I believe I have the context of meatEater
[meatEater] saying: Sending cook my constraints
[meatEater] saying: I consider cook to be a member of my content
[cook] doing: make(steak)
***cook is making steak***
[vegetarian] saying: inviting cook to join my group
[cook] saying: I believe I have the context of vegetarian
[vegetarian] saying: Sending cook my constraints
```

```
[vegetarian] saying: I consider cook to be in my content
[cook] doing: make(pizza)
***cook is making pizza***
```

5.4.4 Case study 2: Self deploying agents

This example demonstrates the potential for software services that migrate across geographical spaces and deploy themselves in their new location.

Scenario

Co-ordination of disaster and rescue missions is a challenging problem for the authorities involved [85]. The deployment location, the number and nature of agencies (commissioned or voluntary) involved cannot be foreseen and speed of deployment is critical. Establishing fast and reliable communication channels between all parties, no matter what their individual resources are, is essential for effective co-ordination.

In our example, disaster recovery head quarters has a co-ordination agent, *hq*, that is mobilised to a wired network in the proximity of the disaster. *hq* has domain knowledge but no local knowledge or resources—it does not know which agencies are on the scene and cannot communicate outside of its host network. In order to effectively co-ordinate the rescue effort *hq* must seek help from a variety of *helper* agents that can carry communication to the operational agencies and provide information about local resources. Examples of help provided by such agents might be: WiFi communication; environmental sensors; public display points; media communications; and utility providers. The suitability of these agents might be determined by proximity, ability or cost.

On arrival *hq* broadcasts a '*services needed*' message requesting that agents with certain capabilities offer their services. The following code snippet illustrates an agent's generic recruitment plan, used to broadcast requests for services to the entire agent space, along with the plan to recruit a WiFi service.

```
/* Broadcast for local services*/
+!recruit(Service)
   <- .broadcast(askIf, has_ability(Agent, Service)).
      ...
      !recruit(wifi).
```

Co-operative agents respond to *hq*'s plea for help by sending a reply stating their abilities and confirming their willingness to join the *group* rescue effort. Below, we show an agent's plan for responding to requests for help.

```
/* Confirm ability and willingness to join */
+!help(Group, Service)
```

```
: .my_name(Me) and has_ability(Me, Service)
<- .send(Group, tell, has_ability(Me, Service));
   .send(Group, achieve, accept(Me, Group)).
   ...
   !help(hq, wifi).
```

The plan has a guard that ensures only genuinely able agents respond, it confirms its ability and requests group membership. Note that in this case, the helper agent does not consider itself to be a member of the group until the group itself directly informs it of its membership — a hierarchical structure whereby membership is controlled by the group is appropriate in this scenario but our proposals also allow agents to control their own `Context`, as shown in Fig. 5.6.

On acknowledgement of group membership *hq* holds the belief *content*(*wifi*), *wifi* holds the belief *context*(*hq*) and *wifi* is provided with authentication procedures to apply to incoming connections; see below.

```
+!accept(Agent, Group)[source(Agent)]
  : is_useful(Agent,_)[source(self)]
  <- +content(Agent);
     .send(Agent, tell, context(Group)).

+content(Agent)[source(self)]
  <-  is_useful(Agent, communicator);
      .send(Agent, tellHow, authentication).
```

Broadcasts of this nature are unavoidable when an agent has no knowledge of the system ahead of deployment. However the context/content mechanism provides a convenient and intuitive alternative that enables more efficient multicast communication; for example, our agent *hq* may have recruited a number of communicator agents to whom it wants to broadcast information, by creating a new agent to act as a container for the communicator agents, *hq* is able to send a message to all *communicators* — using the container agent as a proxy — with the *send*(*group*, *broadcast*(*message*)) message, where the agent *group* receiving the *broadcast*(*message*) message distributes it to all members of its `Content`. Once structures are formed, multicast communication of the following type become the norm:

```
send(communicators, broadcast(found(Survivor, Location))).
send(locators, broadcast(is_clear(Zone))).
```

Fig. 5.8 shows some of the structural changes that take place during deployment of our simple disaster management system.

One of the difficulties of disaster management where life saving rescue is required, is the prioritisation of rescue attempts and subsequent allocation of resources, particularly

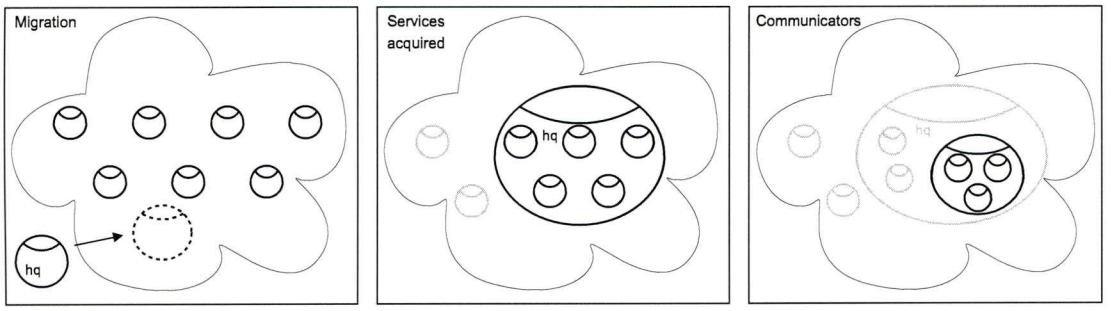


Figure 5.8: The structural view during deployment.

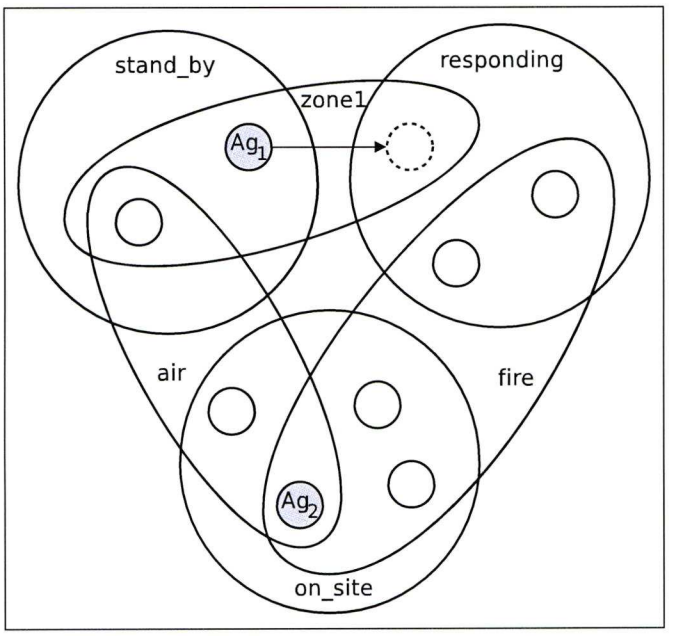


Figure 5.9: The dynamic nature of search and rescue.

when the number, location and needs of victims changes throughout the rescue mission. Continuous re-assessment of the mission's priorities must take place yet pragmatic decisions must be made to ensure rescue teams are effectively deployed and do not, for example, waste time travelling between rescue sites. The context of a rescue team's current activity, their specialisms and location must be considered before allocating them to a rescue site. Our grouping constructs provide the flexibility to model the dynamic nature of these contexts and provides a useful bound for reasoning — reducing the search space for suitable rescue teams. Fig. 5.9 illustrates how our proposal intuitively deals with this situation. The diagram shows rescue agent Ag_1 standing by in zone1 ready to be deployed and its subsequent change of context if it were to respond to a call. Another agent Ag_2 that has both air and fire specialisms is currently attending a rescue site. Using this formalism it is easy to express autonomous behaviour on behalf of the rescue agents;

```
constraint_fails(respond, _) :-
  in_context(responding), in_context(on_site).
```

Giving the agents the above rule prevents them from responding to rescue requests whilst either on route to, or at the scene of a rescue, when combined with the plan below.

```
+!respond(Rescue) : check_constraint(respond, Rescue)
  <-  !attend(Rescue).
```

5.5 Summary of proposal

In this chapter we have proposed a simple extension to BDI languages that permits the development of complex multi-agent organisations. We have shown how the addition of both content and context sets, and constraints is semantically simple and appealing. The key aspect, particularly with contexts and constraints is that an agent's behaviour may be modified, seamlessly, when the agent moves between contexts.

Although we provided a semantic definition for a simple BDI language, we gave this only for illustrative purposes. We expect that developers' favourite logical agent languages could be extended in this way. Importantly, the semantic rules show how this logical extension can be added (relatively easily) to any appropriate BDI language.

Finally, we provided some simple examples here, which complement those of Chapter 4, and illustrate and justify our statement that many agent organisational aspects can be modelling using our two simple concepts. These examples demonstrate how leading organisational and team-working concepts such as roles, joint-intentions and groups fit within our framework, a framework that, if incorporated into BDI languages will enable a consistent agent-organisation semantics across languages.

Chapter 6

Case Studies

This chapter describes the application of METATEM to two pervasive computing scenarios. The general purpose of each of these case-studies was that of evaluating METATEM, but particular emphasis was placed on evaluating the suitability of the agent structuring mechanism, the helpfulness of the context and content metaphor and the effectiveness of the various constructs provided by this project's implementation. The first case-study involves aspects of agency and ubiquity, when a shopper's smart phone interacts with a shopping centre's network services in order to provide an assisted shopping experience. The second case-study concerns the surveillance of a moving target by the cooperation of a loosely connected network of sensors. For each case-study the scenario is described before a solution and its rationale is given, including supporting code snippets and output examples.

6.1 Shopping scenario

This case-study was chosen for a number of reasons. Firstly, it seems clear from the points made in Section 2.1, that the chosen scenario is likely to become reality soon (if it has not already been achieved). Secondly, it presents a number of problems typical of pervasive computing applications and finally, whilst this scenario — at a high-level of abstraction — presents no fundamentally complex computation problems, the problems it does present combine to produce a challenging case-study for the purposes of this project. The case-study was tackled in an iterative way, beginning with a simple scenario with little complexity, which was repeatedly extended, introducing new situations, interactions and features during each iteration, resulting in a complex final implementation. This section describes the intuition behind each of the iterations and some of the design choices made in each case.

6.1.1 Basic scenario

Bob maintains a shopping list on his smartphone, the list has two items on it, bread and milk. Whilst driving home, Bob notices a shopping centre and decides it would be

convenient to buy some items from his list. Whilst walking through the shopping centre, Bob's smartphone is broadcasting messages to nearby stores in order to find stores that have stock of bread or milk. Bob's smartphone alerts him to the presence of a store that sells milk and reminds him that he needs milk. Bob visits the store and buys some milk. Bob's shopping list is updated when his smartphone receives a copy of the electronic receipt from his payment card. Bob continues on his way, and is later prompted to buy bread in a similar way.

This scenario demonstrates the seamless interaction between mobile devices, service providers and users that is expected of pervasive computing applications. Within METATEM the agents specified are

$$phone,\ bob,\ shopping\_centre,\ store\_1,\ \ldots,\ store\_5,\ \text{and}\ p\_card.$$

These agents combine to create a fully connected structural relationship, of which the following is a snapshot example:

$$\begin{aligned}
Content_{shopping_centre} &= \{store_1, ..., store_5,\ phone\} \\
Context_{shopping_centre} &= \emptyset \\
Content_{phone} &= \{bob\} \\
Context_{phone} &= \{shopping_centre,\ store_1\} \\
Content_{bob} &= \{p_card\} \\
Context_{bob} &= \{phone\}
\end{aligned}$$

For this basic scenario, when Bob enters the shopping centre the *phone* receives a broadcast message, inviting it to become a member of the centre's *Content* set. The following lines of code are an extract from the agent specification for *phone*, they ensure that driving to the shopping centre results in subsequent arrival and that he leaves when finished shopping. The third 'rule' demonstrates a programming convenience for bringing execution to an end; the predicate **end** is a special formula which, when satisfied, forcibly stops an agent's execution.[1]

```
// Make it a short drive to the shopping centre and then to home
drive(shopping_centre) & ~enterContext(shopping_centre)
  => NEXT enterContext(shopping_centre);
done(shopping) => NEXT leaveContext(shopping_centre);
in(home,context) => NEXT end;
```

Once a member, the *phone*, prompted by its shopping list, broadcasts the messages $has(store, bread)$ and $has(store, milk)$ at regular intervals within the context of the

[1] METATEM is designed for non-terminating systems and hence the theory provides no elegant means of bringing an executing system to a halt. The **end** predicate was introduced as an alternative to killing the Java Virtual Machine and as a way of enabling agents to perform a dying wish.

shopping_centre, requesting a response from those stores that stock bread or milk respectively. The *shopping_centre* forwards the messages to a selection of stores within easy reach of the smartphone's current location. Each time a response is received the *phone* agent alerts Bob. I.e. *all* stores that Bob passes-by and that sell items his shopping list, are recommended by his smartphone.

Drawbacks

Little intelligence or pro-activity is employed by Bob's smartphone in this first, simple iteration. Bob must walk-by a store for it to be recommended and he may miss a useful store if his route through the shopping centre is not a total traversal of the centre. No comparison of stores is performed and none of Bob's preferences are considered. Subsequent iterations improve these aspects.

6.1.2 Increased reasoning and an additional context

Bob maintains a shopping list on his smartphone, the list has two items on it, bread and milk. On entering a shopping centre, Bob's smartphone queries local stores in a similar way to the basic scenario, but this time it does not react to each individual response and responses themselves are more detailed; a response contains cost and location data. Bob's smartphone receives responses from many stores, and only when multiple responses have been received does Bob's smartphone alert him to a single store, within a short walk, that has stock of both bread and milk. Bob visits the store and makes the purchase. His list updates as before.

In this scenario Bob's smartphone not only relieves Bob of the task of finding the items he wants to buy but also the tasks of achieving good value and a convenient location. In displaying this intelligent behaviour, the system has changed from an integrated messaging system into a personal shopper. Additional agents are specified, including *shop*, *browse*, and *personal_shopper*.

The structure remains similar to that of the basic scenario, however, when first entering the shopping centre a *personal_shopper* agent is created for Bob, this agent performs a task previously undertaken by the *phone* agent — store discovery — but also performs the price and location comparisons. However, this personal shopper does not actually shop, it must not make a final decision and does not have the crucial ability to pay. The *personal_shopper* agent is a service provided by the shopping centre which has an advisory role only and cannot be trusted to make purchasing decisions. We believe it is important to reflect this distribution of task and decision taking responsibilities in the design of the system and have therefore introduced a further two agents; *browse* and *shop*. These agents capture a mode of behaviour rather than a fundamental behaviour — they can be viewed as roles that Bob takes on during the natural course of a shopping trip.

When Bob enters the shopping centre, his *phone* agent belongs not only to the *shopping_centre* context but also to a *browse* context, illustrated by Figure 6.1. This *browse* context suppresses the *phone*'s reaction to incoming recommendations from the *personal_shopper* by providing an appropriate preference. Instead of alerting Bob to each and every recommendation, when browsing, the *phone* agent retains these quotes until a useful number (for comparison purposes) are received. When *phone* is able to make an informed recommendation it moves from the *browse* context into the *shop* context. This change of context triggers a change in priority of the *phone*'s preferences which in turn, results in an alert being sent to Bob when the next recommendation is generated. Bob acknowledges the alert, accepts the recommendation, and proceeds with the purchase as before.

```
// Sample code from the personal_shopper specification
receive(Client, want(Product))
  => NEXT send(context, broadcast(query(Product));

// Preference from the phone's specification
prefer browse to shop when in(browse, context)

// counting recommendations
recomm(X,Product) & (Y is X+1)
  => NEXT recomm(X,Product) | recomm(Y,Product);
recomm(X,Product) & ~receive(personal_shopper, hasStock(S,L,Product,P)
    & hasStock(S1,L1,Product,P1) & S=\=S1
  => NEXT recomm(X,Product);

// Intuitive expression of shopping behaviour
need(P) & in(shopping_centre, context) => browse(P) UNTIL shop(P);
shop(P) => SOMETIME buy(P);
```

Drawbacks

There are no competing contexts that produce interesting behaviours. Again, only stores that Bob passes-by are included in the search — the 'system' does not help Bob discover any stores that he cannot discover himself. The system is reliant upon a fixed sequence of events — enter, browse, shop, buy, leave — and cannot adapt to alternative sequences of events.

6.1.3 Adapting to unexpected human behaviour

As in the basic scenario, but this time, Bob visits the recommended store but for some reason, that is not determinable by the electronic components of our system, Bob does

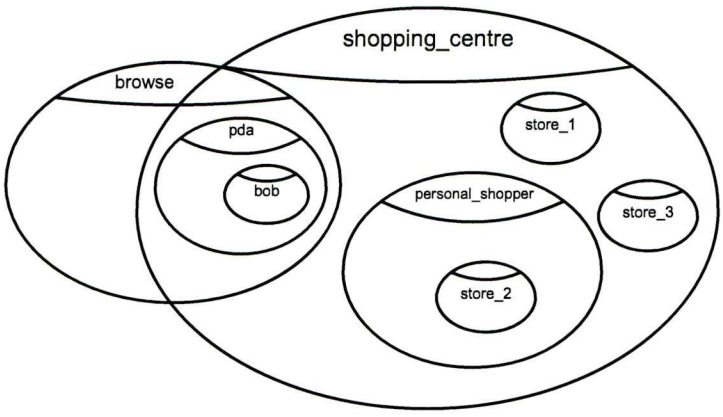


Figure 6.1: A snapshot of one possible structural configuration of agents.

not want to buy his bread and milk from the recommended store. On leaving the store, Bob's smartphone is aware that Bob is leaving the recommended store and that the items on the list have not been bought, as the electronic receipt has not been propagated. The smartphone provides Bob with another store suggestion which Bob accepts, visits and buys his provisions from. After the purchase Bob's phone updates the shopping list and records Bob's preference for the alternative store (over the original store).

This scenario demonstrates the essential adaptation requirement that is commonly associated with pervasive computing situations. With so many autonomous entities within a system, the need to adapt to, and mitigate unexpected (and possibly undesirable) outcomes is compelling [41].

For this section an entirely new 'browse' context was created, triggering a repeat of the browse, compare, shop behaviour. This time however, Bob's dislike for the rejected store is expressed as a preference, which is shared with the shopping assistant agent, by his smartphone. METATEM allows the quoting of terms within messages, allowing the transmission of agent concepts such as preferences, goals and beliefs. However, the recipient receives simple text string that carried no formal indication of the sender's intended semantics. The recipient agent's specification must therefore contain explicit interpretation of messages that convey such concepts. The current implementation of METATEM facilitates this with a number of built-in predicates that allow an agent to dynamically modify its specification by disquoting such strings. Examples of these include `addGoal`, `addRule` and `addPref`. This iteration also called for the dynamic disposal and generation of agents, using the build-in `createAgent` ability.

```
left(Store, context) & hasStock(S, L, Product, P) & need(Product)
  => NEXT send(shopping_centre, query(Product));

receive(Client, query(Product))
```

```
  => NEXT createAgent(browse, content, "src/shopping.agent");

receive(self, newAgent(Ag)) & wants(Client, Product)
  => NEXT send(newAgent, wants(Client, Product));
```

The secondary intention of this scenario was to test and demonstrate the backtracking behaviour of METATEM agents. This apparently simple objective turns out to be a little more difficult than expected due to the side-effects of changing contexts; when the changes from the *browse* context to the *shop* context it draws a metaphorical 'line in the sand' of time, over which it cannot cross when backtracking. Thus, in scenario 6.1.2 at least, the *phone* agent makes a decision (whilst in the *browse* context) that it cannot revise. A number of alternative implementations to that of scenario 6.1.2 were considered, to allow the smartphone to effectively make another recommendation to Bob, whilst retaining the conceptually-helpful *browse* and *shop* contexts. These included

1. the *phone* agent storing its alternative recommendations as instantiated predicates, as opposed to uninstantiated choices, and therefore not needing to backtrack.

2. the *phone* agent receiving an ordered list of recommendations, thus rendering the *browse* agent redundant after *phone* has moved into the content of the *shop* agent.

3. the *phone* remaining in the *browse* context when it enters the shop context, ensuring that the $Content_{browse}$ and $Context_{browse}$ sets do not change, allowing it *and it alone* to backtrack — (comparison of alternative products and deliberation over which recommendation to make can then be delegated to the *browse* agent).

4. creation of an entirely new browse context, triggering a repeat of the browse, compare, shop behaviour. But this time with a preference that expresses Bob's dislike of the rejected store.

5. the *phone* deferring the purchase recommendation until after it has entered the shop context, generating its choices after the contextual change and thus paving the way for future backtracking (assuming no other external interactions take place).

Despite the dubious intuition of deciding to shop then deciding what to buy, option (5) has the most appeal as it fulfills the objective of demonstrating backtracking. However, after all that is said above, backtracking may not be appropriate here. Bob does not walk backwards out of the first store, he does not undo his decision not to buy his provisions from the first store. It does not seem appropriate for one agent to backtrack

over time, when other agents (that do not backtrack) have not been inactive during the same period of time. Also, we can envisage difficulties creating a specification of the *phone* agent that is able to make a recommendation to Bob without performing an 'external' action. Option (4) is therefore the chosen implementation.

6.1.4 Introducing Alice — Bob's friend and lunch date

Whilst Bob is shopping, his friend Alice is also shopping. She needs to stop for lunch and would prefer not to eat alone. She asks her phone to poll the local area for members of her contact list and Bob's name is returned. Alice confirms that she would like to eat with Bob and delegates the arrangements to her phone. Once Bob has agreed, Alice's phone negotiates a restaurant with Bob's smartphone that satisfies both Bob's and Alice's preferences and fits in with their shopping plans. This scenario increases the complexity of the scenario significantly by introducing a second user-agent and thus transforming the communications between a shopper's agent and a store, from a two-party to a three-party protocol.

Sensor networks are a common feature of many people's vision of ubiquitous and pervasive computing, particularly with respect to the term Ambient Intelligence. This iteration emulates a common use of sensor networks — to identify the location of human users — to elaborate the scenario and in so doing demonstrating how the highly connected nature of pervasive systems can be handled by our approach to system specification. This, the most complex of the scenarios attempted, required the specification of further agents. These included *alice*, *dine*, *lunch*, and *restaurant* along with the use of the new concepts: *joint goal* for finding a suitable lunch venue; and *constraint* to prevent shopping alerts while at lunch.

Structurally, this scenario begins to demonstrate the complexity of relationship between entities in pervasive computing environments; the difficulty in drawing the dynamic relationships between relatively few agents illustrates this point. Nevertheless, a snapshot of the structure for this scenario is presented here.

$$
\begin{aligned}
Content_{shopping\_centre} &= \{phone, store\_1, ..., store\_5, restaurant, alice\} \\
Content_{browse} &= \{bob\} \\
Content_{dine} &= \{bob,\ alice\} \\
Context_{restaurant} &= \{shopping\_centre\} \\
Content_{restaurant} &= \{bob, alice\} \\
Context_{bob} &= \{shop,\ browse,\ dine,\ restaurant\}
\end{aligned}
$$

Bob is proceeding with his shopping expedition when the message from Alice is received, inviting him to lunch. Once accepted, the *phone* enters a further browse context (note

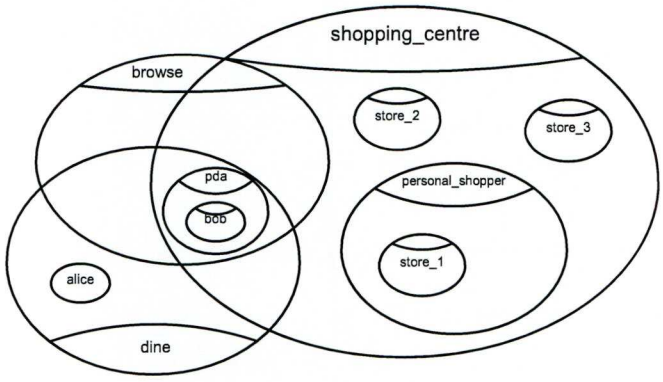


Figure 6.2: A snapshot illustrating multiple contexts and members.

that it may already be in a browse context with respect to the shopping items he intends to buy), this time with the objective of finding a suitable place to eat. Furthermore, Alice's device (Alice and her device are represented as one agent, *alice*, for convenience) is also a member of this *browse* context. An illustration of how this configuration at any one moment is depicted in Figure 6.2. Alice's device issues a goal of finding a restaurant and this goal is propagated by the *browse* context, until it is shared by *phone*. A lunch venue is found in much the same way as stores are discovered, that is, restaurants are proposed and accepted (or rejected). When the number of acceptances received by the *browse* context equals the size of its `content` set, the restaurant is considered to be agreed upon. Note that restaurant proposals can be rejected automatically by *phone* and/or *alice*, due to preferences, but that an acceptance can only be made by explicit user intervention.

Lunch provides a context in which all participants are expected to behave within a few informally defined boundaries. For example, it is normally considered unsociable to use a mobile telephone, rude to be late, and courteous to offer to pay. The author does not seek to advocate the use of technology to transfer behaviours that are clearly, and always shall be, the sole domain of human activity, to machines. However, it seems that using technology to assist humans, making it easier for them to conform to social conventions is an acceptable use of technology. Thus, our *dine* agent is specified with a constraint that it propagates to all members, preventing them from producing shopping alerts during membership and therefore whilst Bob and Alice are at lunch, neither are interrupted by shopping alerts.

```
// Example code from the agent responsible for arranging lunch
count(accept(meetAt(Rest, Time)), X) & size(content, X)
  => NEXT book(Rest, Time, X);
book(R, T, X) => NEXT send(R, confirm(T, X));
book(R, T, X) & in(X, content) => NEXT send(X, confirm(T, X));
```

```
// Constrain alerts when lunching
enter(lunch, context) => NEXT atMost(0, alert, "in(lunch, context)");
```

6.1.5 Results, outcomes and runs

During the development of this implementation, the specification and testing of simple agents (and less-simple but still isolated agents), indicated that the language and its implementation had great potential. Often single-agent systems (or multiple agents derived from a single specification), these agents provided invaluable testing of individual features of the implementation, whilst simultaneously expounding many of the benefits of declarative logic programming — concise, formal expressions with clear semantics that are amenable to automated formal methods. This case-study was the first attempt to use the implementation for the development of a truly multi-agent system, comprising of many agents with significantly different specifications. The scenarios described here were significantly more involved with respect to communication, reasoning, use of meta- and action-predicates, than the test scenarios that preceded it. Generally, it was found that an increased complexity in the scenario being modelled, led to (at least) a corresponding increase in programming difficulty.

Some of the encountered difficulties could be overcome. For instance, when the restrictions of SNF caused the bloating of specifications with blocks of hand-written rules expressing behaviours that could not be concisely expressed with fewer rules. The resulting specifications inevitably contained human error and consequently bugs which were difficult to identify (at least by the same human that introduced the bug). To overcome this, the implementation's input parser was given the capability to re-write rules such as

```
p(a) & p(b) => p(b) UNTIL p(c);

p(X) & q(Y) => q(X) UNLESS p(Y);
```

into SNF automatically. Also, many bugs within the implementation itself were identified and fixed, resulting in a high-degree of confidence in its correctness.

The simplest of the scenarios described above was specified without notable difficulty and the interpreter was able to generate an execution that satisfied each agent's specification. As the scenario is entirely virtual, no real output exists. Instead, the execution is evaluated by monitoring the logical state of each agent as logged by the interpreter. Logging output includes positive predicates, the state of `content` and `context` sets, and any meta-predicates, for each temporal state. The following text is a snippet of logging output from the shopping scenario.[2]

[2]Note that if a temporal state is logically equivalent to the previous, it is not logged.

```
 [java] INFO: [store_1] state 0: [location(store_1,a2) ^ sells(bread,20)]
content:{} context:{} known:{} Meta-predicates:[]
 [java] INFO: [store_2] state 0: [sells(bread,25) ^ location(store_2,c1) ^
sells(milk,15)] content:{} context:{} known:{} Meta-predicates:[]
 [java] INFO: [shopping_centre] state 0: [haveShops] content:{store_4,
store_3,store_2,store_1,store_5} context:{} known:{} Meta-predicates:[]
 [java] INFO: [bob] state 0: [todo(shopping) ^ drive(home)] content:{p_card}
context:{phone} known:{shopping_centre,home} Meta-predicates:[at_most(1,drive),
at_most(1,arrive), prefer(arrive,drive,drive(X1),50)]
 [java] INFO: [bob] state 1: [drive(shopping_centre,home) ^ divert ^
todo(shopping) ^ togo(home)] content:{p_card} context:{phone} known:{shopping_
centre,home} Meta-predicates:[at_most(1,drive), at_most(1,arrive), prefer
(arrive,drive,drive(X1),50)]
 [java] INFO: [bob] state 2: [drive(shopping_centre) ^ togo(home) ^
todo(shopping)] content:{p_card} context:{phone} known:{shopping_centre,home}
Meta-predicates:[at_most(1,drive), at_most(1,arrive), prefer(arrive,drive,
drive(X1),50)]
 [java] INFO: [bob] state 3: [drive(shopping_centre) ^ enterContext(shopping_
centre) ^ togo(home) ^ todo(shopping)] content:{p_card} context:{shopping_
centre,phone} known:{shopping_centre,home} Meta-predicates:[at_most(1,drive),
at_most(1,arrive), prefer(arrive,drive,drive(X1),50)]
 [java] ...
```

Following the initial implementation, satisfying each agent's specification became increasingly difficult, to the extent that goal-directed expressions were sometimes sacrificed in favour of an explicit description of a sequence of states using successive temporal `NEXT` rules. Clearly this is not desirable, as emulating an imperative language is not an aim of this research. However, such circumstances do highlight the purpose of high-level specification languages such as METATEM, and reminds developers to encapsulate appropriate instructions within conventional Java classes and abstract them to a single agent action. METATEM provides a mechanism for doing just this, termed internal- and external-abilities. They are not without their drawbacks however, as execution of any code external to an agent's forward-chaining execution algorithm can prevent backtracking and affects any potential correctness claims and/or formal analysis. These aspects will be discussed further in Chapter 7.

6.2 Surveillance scenario

Surveillance is a useful technique for many activities and is not necessarily indicative of sinister intentions. Observation of endangered species, continuous inspection of assets in hostile or inaccessible environments and health monitoring are examples of surveillance applications. Furthermore, they are surveillance application in which sensor networks could be employed and therefore can be described as applications of pervasive computing. By sensor networks, we mean large numbers of simple, distributed and autonomous sensors, which collectively form ad hoc wireless networks. Initially, research efforts in the area of sensor networks were concentrated on issues related to

the lower levels of the network protocol stack, as indicated by [1], a survey of the state of sensor network art in 2002. More recently, attention appears to be more widely spread [131], including more abstract modelling of sensors, such as the notable use of algebraic topology by Ghrist et al. Using this abstract mathematical technique they have been able to determine the extent to which a domain is coverage by sensors [33] and count surveillance targets [7], without the knowledge of sensor co-ordinates. We do not attempt the same problems as Ghrist et al, but we do take a similar approach to dealing with sensor networks — taking an abstract view of a collection of similar simple entities and attempting to describe properties of the collection as a whole. In our case, we are interested in the properties of a collection (or organisation) of agents.

Recall that, agent-oriented programming languages aim to provide a system developer with constructs that allow them to define the behaviour of complex systems in terms of one or more autonomously acting entities that have clear individual aims. By adopting concepts associated with human rationality, such as beliefs, preferences and goals, these languages also aim to provide a closer correspondence between the intuitive behaviour of a system and its source code. This case-study has been chosen, in part, to evaluate the extent to which METATEM can achieve these aims, and also to evaluate its suitability for use with sensor networks.

6.2.1 Scope

No specific surveillance scenario was intended for this case-study, instead a general scenario in which the area under surveillance is adequately served by sensors (there are no blind spots), the target is always visible but its movement is not predictable and that objects (moving or stationary) exist which have the effect of adding noise to the sensor data. For each surveillance target there exists a unique process that collects, aggregates and interprets data from multiple sensors — a process known as the 'data fusion' process. Importantly, the framework of sensors and fusion processes is considered to be a fully distributed one with no central thread of control. It is also assumed that sensors can be mobile themselves and that the fusion process has no prior knowledge of sensors. In fact, the number of sensors available and their individual attributes, including location and accuracy, will be considered to be dynamic.

This scenario is an interesting one for the application of multi-agent technology for the following reasons:

- the scenario contains many *asynchronous processes*;
- the system must exhibit *adaptive* behaviour in response to
  - unpredictable target movements,
  - sensor failure, and

- dynamic introduction of sensors;

- relationships exist between sensor and fusion processes, and these relationships are based not only upon structure but also responsibility; and

- multiple sensors and fusion processes can be viewed as working *cooperatively* to achieve the surveillance *goal*.

The aim of this case study is to demonstrate the application of a combination of the multi-agent metaphor and executable temporal specification (that METATEM provides), to applications involving many distributed devices. For this reason agent-organisation abstractions were employed where practical during the design of solutions.

6.2.2 Context modelling

Context modelling is a popular technique studied by software engineers who wish to differentiate subtle behavioural differences in systems [122]. These behaviours are often in response to, or in anticipation of, human activity. Since two aims of agent research are to endow software agents with aspects of human intelligence and to provide them with social abilities, the author believes that the context in which an agent is acting should be a first-class construct within an agent programming language and have direct influence on an agent's deliberation mechanism. Not simply an information type that is stored in some centrally held repository.

As discussed in previous chapters, METATEM implementation has built-in constructs that are intended to provide support for context modelling, it is an agent-centred multi-agent language in which the Agent is the primary entity and a multi-agent system is considered to be a society of agents and agents alone. In fact, whereas some agent-oriented languages support other entities from the object and agent paradigms (such as artifacts, objects, services and teams), METATEM supports their modelling but considers them to be agents — with appropriate behaviour and levels of autonomy. Context in METATEM is conceived as the more abstract (and flexible?) notion of something that has an influence over an agent in a system. An agent is considered to reside in (or be contained by) zero or more contexts, each of which are themselves agents. Furthermore, an agent itself can be the context for another agent, in which case it is said to contain that agent. Thus contextual information is interlaced with the very agents of the system, their structural relationships, their behaviour and their communication.

To illustrate this intuition, let us consider all entities in the surveillance area scenario to be agents (sensors, fusion processes, the surveillance environment and even the target) then the diagram in Figure 6.3 exemplifies the contextual relationships between METATEM agents. The diagram depicts an all encompassing environment agent which contains all other agents, a single fusion agent and a number of different sensor agents

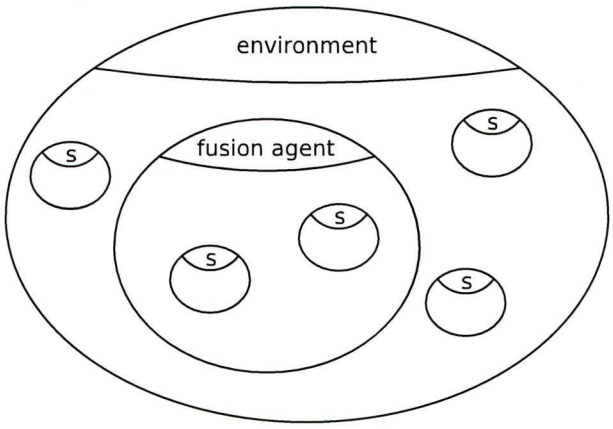


Figure 6.3: Snapshot of the contextual relationships between agents in the surveillance system.

(all labelled **s** for simplicity). The interpretation of these diagrams differs with each application, of course, but Figure 6.3 is intended to depict a fusion process that is operating in an environment with five available sensors, and which is currently receiving valid data from two of those sensors — those that are currently in range of the target. As the target moves around the area under surveillance and in doing so moves into and out of the range of sensors, the sensor agents move into and out of the fusion agent's body whilst remaining, either directly or indirectly, within the environment unless they become unavailable for any reason, such as breakdown. The figure has no explicit representation of the target, however, in this example each fusion agent tracks at most one target, therefore the target's location is directly reflected by the contents of the fusion agent.

At an abstract level, the containing relationship denotes that the agent being contained is in some way influenced by its container, whilst the containing agent has some influence over the agents it contains. Note that a given application will apply more concrete semantics to the relationship between the container agent and the agents it contains, and that these might be semantics of authority, obligation, ownership or anything else appropriate to the application concerned. In the surveillance case, the relationship between fusion agent and sensor agents is considered to be that of a co-operative team, in which the fusion agent represents the team leader and the sensor agents the team members. Sensors cooperate when they receive requests for sensor data and are obliged to provide valid and timely data for their respective locations.

6.2.3 The surveillance area

The surveillance area itself is a two-dimensional bounded grid that is divided into regions, around which a target moves in an unpredictable way. Sensors have a limited range that extends in a 360^o arc from their centre. Sensors detect all objects within

their range, and attribute a coordinate pair and an identifying colour to all objects they detect. Sensors are distributed around the area such that each sensor belongs to a single region. The range of a sensor's observations may extend across the boundary of a region but a sensor is regarded as 'covering' a single region determined by the location of that sensor. Sensors are aware of the region they cover. The case-study begins with a simple surveillance area that is divided into three regions named `north`, `central` and `south`, occupied by four sensors whose locations are fixed and positioned to give total coverage of the surveillance area, and a single target. This simple surveillance area is sketched in Figure 6.4.

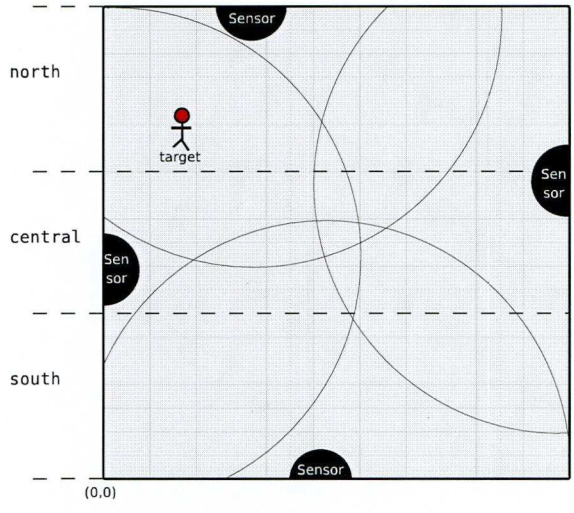


Figure 6.4: The surveillance area uses a coordinate system and is divided into regions. This illustration shows four sensors whose ranges overlap regions and the ranges of other sensors.

To aid the evaluation of sensor and fusion agents the target object was initially given predictable behaviour, following a rectangular path around the perimeter of the surveillance area at a constant speed. This guaranteed that the target moved into and out of the ranges of all sensors during its route and placed it inside regions of overlapping sensor coverage. Collectively, the sensors are able to provide continuous detection of any object as it moves around the surveillance area. It is the job of the data fusion agent to collate detection data pertaining to an individual target object and provide a reliable location for a target irrespective of possible anomalies in sensor data.

Scalability

A simple, uncomplicated and static surveillance area is useful for the demonstration and exposition of our techniques but in order to demonstrate the utility, flexibility and

robustness of the approach the surveillance area was scaled up and in doing so increased the complexity due to:

- a larger coordinate space,
- more sensors,
- multiple targets,
- non-target objects,
- mobile sensors, and
- incomplete sensor coverages

An illustration of a scaled-up scenario is depicted in Figure 6.5. In addition to the obvious increase in computational complexity that these aspects introduce, a number of interesting issues arise that can be tackled intuitively with the agent paradigm, providing solutions which are supported by METATEM's agent grouping constructs. Such solutions include the abstraction of large numbers of neighbouring/co-located sensors into a single sensor agent, the division of the environment into sub-agents corresponding to regions, and the encouragement of mobile sensors to adopt strategic positions on behalf of a fusion agent; these solutions are discussed in more detail later.

6.2.4 Surveillance example: Architecture

The CONCURRENT METATEM takes an agent-centred view of multi-agent programming in which all entities identified during analysis of a problem are considered to be agents for the purposes of implementation. An entity may be a *concrete* agent with the ability to directly interact with the environment, or an *abstract* agent that interacts only with other agents. A concrete agent often has a counterpart entity identified during analysis whilst abstract agents often help provide the fabric of an agent society by facilitating multi-agent concepts such as coordination, cooperation, normative behaviour and joint beliefs. A multi-agent system comprising of only concrete agents implies a flat structure, provides little insight into the relationships of influence that exist between agents and gives no indication of any hierarchies that may exist.

6.2.5 Surveillance example: Environment

As this is a simulated surveillance area, the environment is represented as an agent that is responsible for modelling the surveillance area and for providing sensors with raw data about objects within the modelled area. The surveillance area is a bounded environment that sensors and objects stay within at all times. Thus, the environment agent is a natural *container* for sensor and fusion agents.

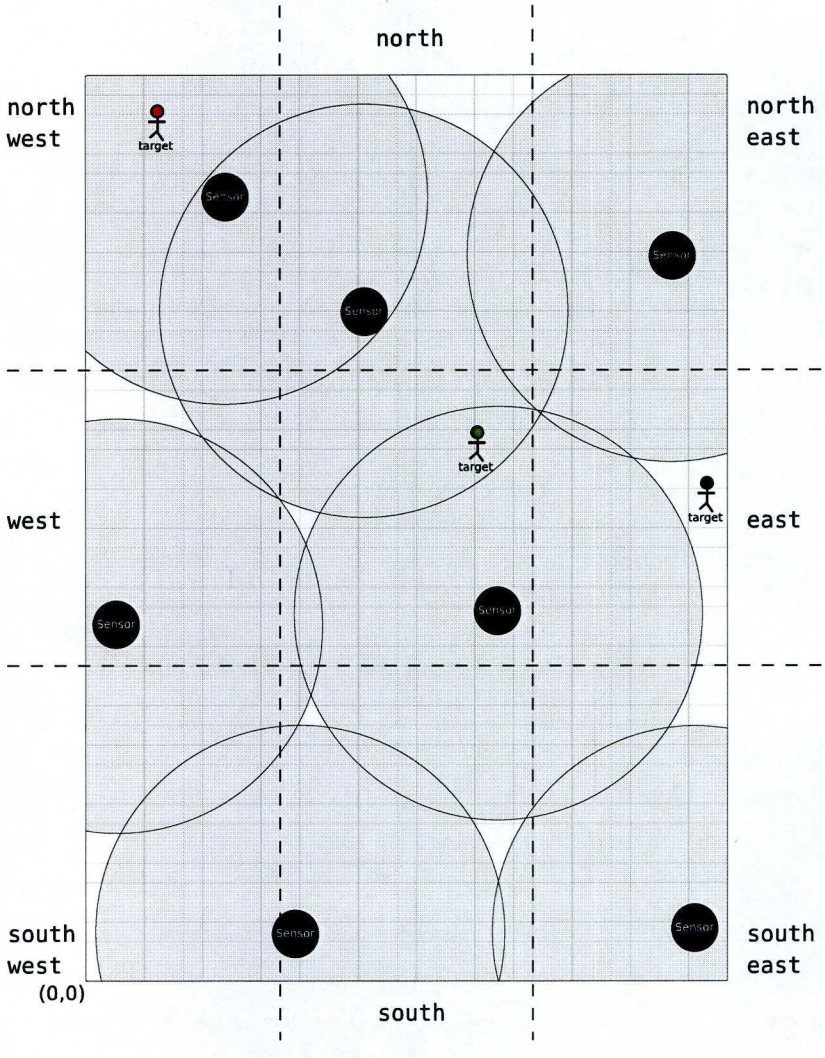


Figure 6.5: The addition of multiple targets, mobile sensors and incomplete coverage, add to the complexity of the surveillance task.

METATEM agents are able to call upon arbitrary blocks of Java code by declaring an ability that corresponds to a Java class. For instance, the environment agent declares the ability `surveillance` as follows

```
ability surveillance : surveillance.CreateSurveillanceArea;
```

where the file `CreateSurveillanceArea.class` appears in the classpath of the host Java run-time environment. This ability is then available to the environment agents as a predicate in logical rules. For example, the following rules employ the `surveillance` ability in two circumstances. The first rule creates the environment and the second starts the simulations thread.

```
start => surveillance(create);
receive(self,surveillanceArea(Area))
      => NEXT surveillance(begin,Area);
```

Note that the abilities can be modified by arguments and that Java objects can be received by agents as terms of predicates and subsequently passed to abilities and/or other agents.[3]

The environment is also able to place sensors into specified regions of the surveillance area by instantiating the `surveillance` ability with appropriate terms

```
... => NEXT surveillance(attach, Sensor, Area, Region);
```

and inform a fusion agent of its surveillance target

```
... => NEXT send(FusionAg, target(Colour, Region));
```

For simplicity, the solution focused on the problem of tracking an identified target's movements, as opposed to the identification of a target. For this reason, our environment agent provides the fusion agent with the target's distinguishing feature (its colour) and initial location. Once this initialisation is complete, the environment agent provides a communication link between sensor and fusion agents according to the following general rule.

```
communication: {

  // Broadcast all 'broadcast' messages to all members of content:
  receive(X, broadcast(M)) & inContent(X) & inContent(Y) & X=\=Y
    => NEXT send(Y, M);
}
```

The following sections describe two approaches to the solution, which differ in the abstraction used to express the relationship between fusion and sensor agents.

[3]Although passing of Java objects and terms other than basic string terms is possible, this is not encouraged as it implies some common knowledge of the meaning of the transferred term.

6.2.6 Scenario One — Fusion agent as coordinator

In this, the first of two contrasting interpretations of the *content* and *context* constructs, the fusion agent assumes the role of coordinator over a 'team' of sensor agents. The fusion agent, and its objective, are the primary *context* under which the sensor agents are operating, therefore the fusion agent assumes the container role and the sensor agents are the contained. The fusion agent coordinates the movement of sensor agents into and out of its content according to the relevance and usefulness of their data.

Specifying the system

Recall that METATEM requires a simple system file that declares the name of each agent, its corresponding specification file and the initial relationships between agents. The file declares an initial containing relationship between the environment and all other agents. It is reproduced here in its entirety.

```
agent environment: "Surveillance/environment.agent";
agent sensor1: "Surveillance/sensor.agent";
agent sensor2: "Surveillance/sensor.agent";
agent sensor3: "Surveillance/sensor.agent";
agent target: "Surveillance/fusion.agent";

environment {
  Content: sensor1, sensor2, sensor3, target;
}
```

Note that an environment agent has been declared. In fact this is a wrapper around a Java simulation of the surveillance area. Its behaviour described by METATEM code is limited to configuration rules and the communication infrastructure it provides to the other agents. It is not only convenient to consider the environment to be an agent but also natural if one does not distinguish between messages and perceptions — each are external influences that an autonomous agent assigns its own meaning to. Consequently METATEM agents do not have a percept construct, only a simple flexible message system. However, for this simple situation, it is not useful for the sensor and fusion agents to have a reference to the environment agent in their context set as they do not have any affect on it.

Specifying the sensor agents

A sensor agent is informed of its regional location by the environment. The following rules specify that the sensor agent retains this knowledge and must not believe that it is in two places at once.

```
// The environment will inform me of my location...
receive(environment, attached(self,Location))
  => NEXT myLocationIs(Location);

// ... and I must remember it.
myLocationIs(L1) & ~receive(environment, attached(self,L2))
  => NEXT myLocationIs(L1);

// I cannot be in more than one location.
myLocationIs(L1) & myLocationIs(L2) & L1=\=L2 => false;
```

A sensor agent receives regular messages from the environment that correspond to *raw* sensor data. These take the form

```
receive(environment, sensor(Data))
```

where `Data` is a complex term of in one of two forms;

```
data(noise)
```

or

```
data(Colour,X,Y).
```

As a sensor makes no judgement on the usefulness of its data (it does not recognise noise) all data is forwarded to those fusion agents in its context. How sensor agents come to reside in a fusion agent's context set is explained later.

```
communication : {
  // All sensor data is passed to my Context agent(s).
  // (Any number of fusion agents.)
  receive(environment, sensor(Data)) & inContext(FusionAg)
    => NEXT send(FusionAg, Data);
}
```

Other messages a sensor agent may receive are broadcast requests from fusion agents that are tracking a target in the vicinity of the sensor, and direct messages from fusion agents who no longer require their data (for whatever reason). In these situations, fully cooperative behaviour on behalf of the sensor is specified.

```
obligations : {

  // Always oblige when asked to provide sensor data
  receive(environment, requestDataFor(FusionAg,Location))
```

```
    & myLocationIs(Location) => enterContext(FusionAg);


  // Always leave a context when no longer needed
  receive(FusionAg, redundant) => leaveContext(FusionAg);
}
```

Hence, agents enter and leave contexts when requested to.[4] Note that this does not prevent sensor agents from leaving of their own free will.

Specifying the fusion agents

The fusion agents have the most sophisticated specifications. Each must track a target using varying amounts and frequency of data from sensors which may be transmitting only noise.

The temporal semantics of METATEM allows us to describe some desirable properties of the fusion agent with considerable concision. For example, all sensors in a fusion agent's content are categorised in each moment of time, as either sending data, sending only noise or sending nothing at all. A sensor that is categorised as sending only noise three times without sending any valid data in the intermediate time steps, is dropped from `Content`

```
nothingFrom(Sensor) & onlyNoiseFrom(Sensor) => false;
onlyNoiseFrom(Sensor) & dataFrom(Sensor) => false;
nothingFrom(Sensor) & dataFrom(Sensor) => false;
...
dataFrom(Sensor) => NEXT noisesFrom(0,Sensor);
onlyNoiseFrom(Sensor) & noisesFrom(X,Sensor) & (Y is X+1)
  => NEXT noisesFrom(Y,Sensor);
nothingFrom(Sensor) & noisesFrom(X,Sensor)
  => NEXT noisesFrom(X,Sensor);
inContent(Sensor) & noisesFrom(3,Sensor)
  => NEXT send(Sensor, redundant);
```

When the target moves or when the number of sensors in `Content` reaches a critical level, the fusion agent must recruit sensors located in the vicinity of the target's new location. It translates the target's coordinates into a region and broadcasts a request for sensors in that region.

```
targetCoord(X,Y) & (Y<10) => targetIn(south);
...
```

[4] This could be made more complex by introducing some non-trivial negotiation at this point.

```
targetIn(Region) => NEXT targetPreviouslyIn(Region);
targetIn(Region) & ~targetPreviouslyIn(Region)
  => NEXT targetMovedInto(Region);
targetIn(Region) & targetPreviouslyIn(Region)
  => NEXT ~targetMovedInto(Region);

// When moving into a region, broadcast a request for sensors
targetMovedInto(Region)
  => send(environment, broadcast(requestDataFor(self, Region)));
```

Finally, as the fusion agents and sensor agents execute asynchronously, the fusion agent cannot rely upon fresh sensor data for each of its reasoning cycles. Instead it can only believe that the target remains in its previous location.

```
receive(Sensor, data(Colour,X,Y)) & tracking(Colour)
  => dataReceived;
targetCoord(X,Y) & ~dataReceived => NEXT targetCoord(X,Y);
```

Listings of METATEM code for this case-study are provided in Appendix B.

6.2.7 Scenario Two — Sensor agents as service provider

The second abstraction of agent relationships gives sensors the role of service provider and fusion agents the role of client. In this configuration sensor agents also represent environmental sensors and provide a software interface for other agents to access the sensor data. The key difference between this abstraction and the previous is that fusion agents (the client of the sensor agents' services) enters the context of a sensor agent when they want to use the sensor's service. In essence, this is a role-reversal in comparison to Scenario One. Recall that Scenario One involved a coordination target agent that invited sensor agents into its content and ejected them when it no longer needed them. Now sensor agents invite fusion agents to use their service and fusion agents leave when the service is of no use.

For the initial, small scale, implementations with a single target and three regions, the METATEM code for this scenario differed little from that of Scenario One; the exchange of `content` for `context` and the use of different structural modification constructs was all that was needed, however, there are some significant conceptual differences between the two approaches that have ramifications on the scaling of our case study.

6.2.8 Scale, elaboration and performance

The case-study provides many opportunities to evaluate the scalability of the implementation, and several options for expansion were identified. For example, increasing

the number of regions within the surveillance area (whilst the area's size remains unchanged), increasing the size of each region, increasing the number of sensors and increasing the number of targets. Upon implementation of such expansions an interesting weakness tended to manifested itself, related to the handling of large volumes of messages by individual agents. A influx of messages during a single reasoning step of a fusion agent (for example) could cause sufficient delay for the target's position to be permanently lost. A bottle-neck of communication was being formed. The success of solutions to this problem relies on minimising the number of messages received by any agent whose specification includes rules containing disjunctions (choices) that are fired by receipt of messages. The two architectures employed different solutions to this problem and each are discussed in Chapter 7.

This section is concluded with a sample of logging output that illustrates the number of messages that the target must handle during deliberation and gives some idea of the practicality of the implementation.

```
 [java] INFO: [target] state(191): [(tracking(red) ^ inContext(sensor6) ^
enteredContext(sensor6) ^ inContext(sensor8) ^ enteredContext(sensor8) ^
inContext(sensor7) ^ enteredContext(sensor7))]
 [java] INFO: [target] state(192): [(nothingFrom(sensor6) ^
noisesFrom(0,sensor6) ^ tracking(red) ^ noisesFrom(0,sensor7) ^
noisesFrom(0,sensor8) ^ nothingFrom(sensor8) ^ nothingFrom(sensor7) ^
inContext(sensor6) ^ inContext(sensor8) ^ inContext(sensor7))]
 [java] * Target red has moved to java.awt.Point[x=5,y=3]
 [java] * Target green has moved to java.awt.Point[x=15,y=3]
 [java] * Target yellow has moved to java.awt.Point[x=17,y=30]
 [java] * Target black has moved to java.awt.Point[x=8,y=25]
 [java] INFO: [target] state(201): [(noisesFrom(0,sensor7) ^
nothingFrom(sensor6) ^ tracking(red) ^ noisesFrom(0,sensor8) ^
noisesFrom(0,sensor6) ^ nothingFrom(sensor8) ^ nothingFrom(sensor7) ^
receive(sensor7,data(red,5,3)) ^ receive(sensor6,data(green,15,3)) ^
receive(sensor6,data(red,5,3)) ^ receive(sensor8,data(green,15,3)) ^
inContext(sensor6) ^ inContext(sensor8) ^ inContext(sensor7) ^ dataReceived)]
 [java] * Target red has moved to java.awt.Point[x=6,y=4]
 [java] * Target green has moved to java.awt.Point[x=16,y=4]
 [java] INFO: [target] state(202): [(noiseFrom(sensor8) ^ targetCoord(5,3) ^
dataFrom(sensor7) ^ dataFrom(sensor6) ^ noisesFrom(0,sensor7) ^
tracking(red) ^ noisesFrom(0,sensor8) ^ noisesFrom(0,sensor6) ^
noiseFrom(sensor6) ^ inContext(sensor6) ^ inContext(sensor8) ^
inContext(sensor7) ^ targetIn(south) ^ onlyNoiseFrom(sensor8))]
 [java] INFO: [target] state(203): [(targetMovedInto(south) ^ targetInSight ^
noisesFrom(0,sensor6) ^ noisesFrom(0,sensor7) ^ nothingFrom(sensor6) ^
tracking(red) ^ nothingFrom(sensor8) ^ nothingFrom(sensor7) ^
noisesFrom(1,sensor8) ^ targetCoord(5,3) ^ targetPreviouslyIn(south) ^
inContext(sensor6) ^ inContext(sensor8) ^ inContext(sensor7) ^
targetIn(south) ^ send(environment,broadcast(requestDataFor(target,south))))]
 [java] INFO: [target] state(204): [(nothingFrom(sensor6) ^
nothingFrom(sensor8) ^ nothingFrom(sensor7) ^ targetInSight ^
noisesFrom(0,sensor7) ^ tracking(red) ^ noisesFrom(0,sensor6) ^
noisesFrom(1,sensor8) ^ targetCoord(5,3) ^ targetPreviouslyIn(south) ^
inContext(sensor6) ^ inContext(sensor8) ^ inContext(sensor7) ^ targetIn(south))]
```

```
 [java] INFO: [target] state(206): [(nothingFrom(sensor6) ^
nothingFrom(sensor8) ^ nothingFrom(sensor7) ^ targetInSight ^
noisesFrom(0,sensor7) ^ tracking(red) ^ noisesFrom(0,sensor6) ^
noisesFrom(1,sensor8) ^ targetCoord(5,3) ^ targetPreviouslyIn(south) ^
receive(sensor6,data(red,6,4)) ^ receive(sensor7,data(red,6,4)) ^
receive(sensor6,data(green,16,4)) ^ receive(sensor8,data(green,16,4)) ^
inContext(sensor6) ^ inContext(sensor8) ^ inContext(sensor7) ^ dataReceived ^
targetIn(south))]
 [java] * Target red has moved to java.awt.Point[x=6,y=5]
 [java] * Target green has moved to java.awt.Point[x=16,y=5]
 [java] INFO: [target] state(207): [(noiseFrom(sensor8) ^ targetCoord(6,4) ^
dataFrom(sensor6) ^ dataFrom(sensor7) ^ targetInSight ^ noisesFrom(0,sensor7) ^
tracking(red) ^ noiseFrom(sensor6) ^ noisesFrom(0,sensor6) ^
noisesFrom(1,sensor8) ^ targetPreviouslyIn(south) ^ inContext(sensor6) ^
inContext(sensor8) ^ inContext(sensor7) ^ targetIn(south) ^
onlyNoiseFrom(sensor8))]
```

Chapter 7

Evaluation

This project has considered a pressing problem for distributed systems designers, theoretical computer scientists and software engineers; that of how to handle the complexity of an increasingly distributed and mobile pool of resources in a way that allows the development of useful, intuitive and more context-aware software. This chapter evaluates, with respect to this problem, a number of facets of the approach taken by this project. We evaluate the multi-agent abstraction and the METATEM implementation, both with respect to the most-challenging of the case-studies (surveillance) and context-sensitive applications in general.

The purpose of this evaluation is not to appraise any single interpretation of agent theory, or to evaluate the METATEM implementation in terms of conventional performance measures, such as consumption of computing space and time. The project's goal is not to obtain an optimum solution for any individual pervasive computing scenario and therefore this does not form part of our evaluation either. Rather, we are most interested in evaluating whether or not METATEM, with its agent grouping structures and other additional features can appropriately and usefully employ multi-agent abstractions in the implementation of pervasive computing systems. Also of consideration, is the suitability of the current METATEM interpreter and tools for assisting future research in this area.

The surveillance scenario proved to be an exacting case-study (as, perhaps, is to be expected of all pervasive computing case-studies), providing ample opportunities to apply multi-agent abstractions but being unforgiving of specification errors. Both case-studies provided good working experience of the METATEM language and this experience has contributed to one of the current targets of this research area — that of formulating an appropriate design methodology for the language. The surveillance scenario is highly dynamic in nature but can be criticised as mono-contextual, as any multi-context situations in which an agent finds itself, involve similar context types. However, the shopping scenario provided ample opportunity for evaluation of multi-context situations involving contexts of differing type. Throughout our experimenta-

tion, the METATEM interpreter was subject to close scrutiny, enabling errors to be corrected, weaknesses to be identified and new features to be implemented and/or proposed.

7.1 Experiments

Three experiments involving increasingly sophisticated surveillance areas, were designed to provide some empirical evidence to support, or to refute, the utility claims of this approach. The two alternative configurations of agents employed when modelling the surveillance application, and described in Sections 6.2.6 and 6.2.7, were then applied to each of the three surveillance areas. As the surveillance areas increased in complexity, each scenario needed modification to cope with the increased load, and to enable the agents to adapt to a wider range of situations. These modifications were made in such a way that upheld each scenario's original paradigm and employed popular concepts of multi-agent organisation whenever possible. This section describes the experiments and some of the measures necessary to adapt each scenario to the increased complexity.

7.1.1 Extended surveillance scenarios

The *simple* surveillance area described in Chapter 6 was extended twice to form an *extended* and a *complex* environment. Table 7.1 details these extensions. In each

Surveillance area	Simple	Extended	Complex
No. of sensors	3	8	16
No. of target objects	1	2	4
No. of non-target objects	0	4	8
Sensor detection range		$\longrightarrow$ reducing $\longrightarrow$	

Table 7.1: Comparison of simple, extended and complex surveillance areas.

experiment the target object takes a predictable path around the surveillance area — either moving clockwise or anticlockwise around the perimeter of the area. This was felt necessary to aid the evaluation of the fusion agent's performance. The fusion agent, of course, has no prior knowledge of the target object's movements. In contrast, non-target objects move erratically around the surveillance area with no regular pattern.[1] In all experiments objects moved to adjacent positions in the grid every 3.7 seconds. An intentional asynchrony between object movement and sensor reading was introduced by passing raw data, from the environment to sensors, every 4.0 seconds.

[1]The erratic movements of non-target objects are hard-coded and therefore repeatable across experiments.

Each of the two basic specifications for the two scenarios coped well with the simple surveillance area. The respective fusion agent was able to maintain an accurate belief about the location of its target object for an hour of execution. Note that we consider the fusion agent's belief about the location of its target to be accurate if the interval between the target moving to that location and the agent holding the belief `targetCoord(X,Y)` is no more than five seconds. The following snippet from the execution log demonstrates the fusion agent, named `target` here, taking two seconds to update its state with the red target object's new position.

```
[java] * Target red has moved to java.awt.Point[x=0,y=5]
[java] 15-May-2009 22:03:52 metatem.agent.BasicAgent logState
[java] INFO: [target] state(315): [(noisesFrom(0,sensor3) ^
  nothingFrom(sensor3) ^ targetInSight ^ targetCoord(0,4) ^
  tracking(red) ^ targetPreviouslyIn(south) ^ targetIn(south) ^
  receive(sensor3,data(red,0,5)) ^ inContent(sensor3) ^ dataReceived)]
[java] 15-May-2009 22:03:52 metatem.agent.BasicAgent logState
[java] INFO: [target] state(316): [(targetInSight ^ tracking(red) ^
  targetIn(south) ^ noisesFrom(0,sensor3) ^ inContent(sensor3) ^
  targetPreviouslyIn(south) ^ dataFrom(sensor3) ^ targetCoord(0,5))]
```

When the extended surveillance area with two target objects, four non-target objects and eight sensors, is introduced, the increased volume of messages from sensors quickly overwhelms the fusion agents. Each temporal state takes longer to compute, due to the increased number of messages containing data about its target object and the non-target objects. A fusion agent must decide upon exactly one coordinate for its target object per temporal state and as the number of messages increases the number of choices it has also increases. As each temporal state takes longer to compute the backlog of messages that collect in an agent's inbox increases further, exacerbating the problem in subsequent states until the fusion agent is overwhelmed and loses track of its target. This occurs quickly — within one minute — in each of the two scenarios, but the solution to this problem was different in each case.

Adapting Scenario One to the extended surveillance area

Recall that Scenario One involves a fusion agent forming a container around the sensor agents that it believes are within range of its target object or are located within the same region of the surveillance area. Sensors remain in `content` until they send three consecutive noisy messages. This behaviour was not adequate for the extended scenario as the size of the `content` set and the number of messages it generates increases the computation time for each temporal state beyond practical levels. A number of solutions for this problem were considered, including;

- increasing the number of regions that sub-divide the surveillance area;
- removing noisy sensors sooner;
- acting upon only a sample of the total messages received;
- creating a message filter; and
- grouping concrete sensors into abstract sensors.

Increasing the number of regions was considered inappropriate as it was thought to increase the risk of losing a target. Removing noisy sensors sooner was ruled out for the same reason. Taking a sample of the total messages received might prove useful if a method of guaranteeing that messages from agents are dealt with fairly, i.e. messages from no individual agent are overlooked indefinitely. But it was assumed that the cost of ensuring fairness would be comparable to (and possibly greater than) the cost of acting upon all messages.

The chosen solution for this problem involved filtering messages and was implemented by the creation of sub-agents to which the fusion agent delegated the gross data capture from concrete sensor agents. These fusion delegates maintained a group of sensor agents from which they were responsible for receiving all data. The delegate agent performed a crude filtering only (it did not make any judgement on the accuracy of sensor data) and passed only relevant messages on to the fusion agent. Figure 7.1 illustrates the structural changes made in adapting Scenario One to the

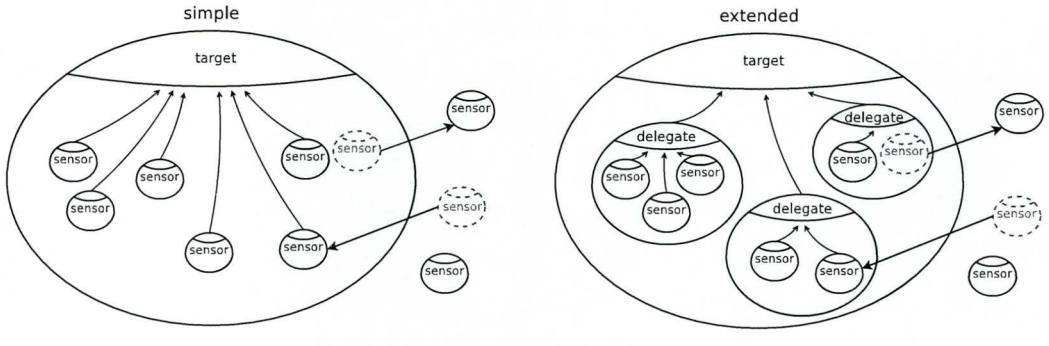


Figure 7.1: Scenario One adapted for the extended surveillance area.

extended surveillance area and illustrates the reduction in the number of multicast message recipients and the number of messages received by each fusion agent. The abstraction of fusion agent as coordinator is maintained as it continues to orchestrate the movement of sensors into delegate groups. Delegate agents reuse code that is employed by fusion agents for detecting noisy sensor messages and remove sensors from their contents when a threshold — which could be mandated by the fusion agent — of consecutive noisy messages is reached.

An interesting elaboration of this solution was necessary when a target agent received a sudden influx of sensor agents into its `content`. In such a situation a delegate agent is formed to group the incoming sensors, in the manner described above. However, it was observed that these groups (having higher than an average number of content agents) formed a bottle-neck of communication due to the number of messages they received from sensors. The solution employed for this problem involved creating a second layer of filtration to ease the burden on the delegate agent itself. This layer was contained by the delegate agent itself and remained independent of the fusion agent. Figure 7.2 depicts the location of this extra layer. This layer cascades sensor data about only one target, but does not make any judgement about the accuracy of individual pieces of data when passing on multiple, contradictory pieces of data. The

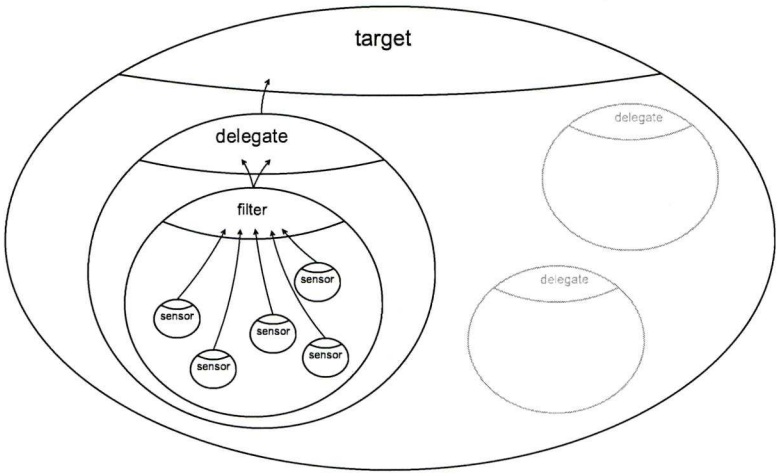


Figure 7.2: Layering the filtration of messages to reduce message processing for the delegate agents.

success of this technique relies on minimising the number of messages received by any agent whose specification includes rules containing disjunctions (choices) that are fired by receipt of messages. In our case-study, the bulk of messages are received by the filter agent and this agent has a deliberately lean specification. It is the delegate agent whose state generation is most complex due to the number of temporal rules containing disjunctions.

It should be noted that any increase in the number of agents in a system introduces new threads, which in turn increases the thread scheduling overhead. Thus, since an agent's execution is a logical one, a METATEM system is *not* a high performance system with respect to execution time and that accommodation for this must be made in order to conduct practical experiments. A number of measures were taken to increase the processing resources allocated to delegate and fusion agents, including the prioritisation of their threads and forcing other threads to yield. Clearly these platform-level tweaks are far from ideal and the fact that they are effected externally to the agent system

specification (in this case by native Java code modification) can be seen as both an advantage and a drawback.

Adapting Scenario Two to the extended surveillance area

Although Scenario Two suffered from the same problem with respect to messaging, the structural and metaphoric relationships between fusion and sensor agents was such that a different solution was needed.

Fusion agents in this scenario are viewed as roaming clients, moving into the context of sensor agents in order to make use of their services. For the simple surveillance area, sensor agents played a passive supporting role in the tracking of target objects — they passed on all sensor data and in doing so forced the fusion agent to discriminate between relevant, irrelevant, accurate and inaccurate data. Scenario Two was modified to cope with the extended surveillance area by asking the sensors to perform a less passive role, giving them some authority to withhold data from the fusion agents in their `content`, or rather, informing them of the object being tracked and explicitly requesting data pertaining to that object only. The following snippets of METATEM code capture some of this behaviour.

```
// When a fusion agent enters the context of a sensor, it provides
// the sensor with the colour of the object it is tracking.
enteredContext(Sensor) & tracking(Colour)
  => NEXT send(Sensor, tracking(self, Colour));

// When a sensor receives this information from a fusion agent, it
// retains the knowledge while the agent remains in its content.
receive(FusionAg, tracking(FusionAg, Colour))
  => NEXT relevantTo(Colour, FusionAg);

relevantTo(Colour, FusionAg) & in(FusionAg, context)
  => NEXT relevantTo(Colour, FusionAg);

receive(environment, sensor(target(Colour,X,Y)))
 & in(FusionAg, context)
 & relevantTo(Colour,FusionAg)
   => NEXT send(FusionAg, target(Colour,X,Y));
```

A sensor agent now has a list of objects that are being tracked by the members of its `content`, and will only send messages (to an agent in its `content`) if a corresponding agent/object pair exists in this list. Structurally, this scenario has not been changed for the extended surveillance area, instead it should be viewed as a distribution of responsibility across the existing structure.

There was one major consequence in the behaviour of the fusion agent as a result of this change. The fusion agent no longer received noisy messages, therefore its strategy for leaving the context of sensors — based upon the receipt of consecutive noisy messages — was rendered ineffective. This resulted in an accumulation of sensor agents in the fusion agent's `context` until, after one passage around the surveillance area, all sensor agents are members. Whilst this did not cause a real problem in our experiments, as fusion agents were not inundated with noisy messages, it is clearly not a solution that scales to surveillance areas larger than those in our modest experiments. The solution employed for this involved the fusion agent retaining knowledge of the location of sensors. The fusion agent then leaves the context of sensors that have not recently sent a message, whose location is in a different region to that of the target and whose regional location is not adjacent to the target's region.

```
// Inform agents entering content where the sensor is.
enteredContent(FusionAg) & myLocationIs(Location)
  => NEXT send(FusionAg, located(self,Location))

in(Sensor, context) & silencesFrom(X,Sensor) & (5 < X)
    & targetIn(TargetLocation)
    & located(Sensor,SensorLocation) & TargetLocation=\=SensorLocation
    & ~adjacent(TargetLocation,SensorLocation)
  => NEXT leaveContext(Sensor);
```

With these modifications the two scenarios are able to cope with the increased complexity introduced by the extended scenario. Each fusion agent is able to track its target object accurately for several circuits of the surveillance area.

7.1.2 Complex surveillance scenarios

The complex surveillance area saw the number of sensors, target objects and non-target objects double in relation to the extended version. Each sensor had a much reduced detection range, resulting in increased movement of objects between sensors. An object will often come into, pass through and go beyond, the range of a sensor in less than ten seconds. Also, objects were more likely to stray into areas of no sensor coverage.

It was felt that the successful tracking of targets in this complex scenario was only possible if sensors were given the mobility to physically track the target object's movements. Thus increasing the duration that an object stays within an individual sensor's range. But this implies a one-to-one mapping between sensors and objects, whereas previously a sensor supported the tracking of multiple objects — the focus of a sensor was a single region of the surveillance area as opposed to a single object. With multiple (four) target objects, each with a corresponding fusion agent, the physical

tracking of objects by sensors strongly suggests a division of labour and formation of sensor teams. It was with concepts of shared goals, and team leadership in mind that each scenario was modified to cope with the complex surveillance area. Some general enhancements to the simulated environment were made. In particular, sensors were given an ability to move around the grid-like coordinate space of the surveillance area by using METATEM's API to create an ability predicate. This ability predicate `move`, when made true by a sensor agent, effects a translation on that sensor's location in the environment. For example, the predicate `move(1,0)` corresponds to a movement one unit eastward. No limit was placed on the values of the two arguments to `move` in order to ensure that sensors can 'travel' at least as fast as the object it is tasked with tracking.

The complex environment requires sensors to exhibit more intelligence than the previous environments. Sensors must move autonomously as giving fusion agents the responsibility of directing all its sensors is likely to place an unfair burden them. Sensors must also differentiate between fusion agents in its content, one or more of which may have conflicting desires for its services.

Adapting Scenario One to the complex surveillance area

Recall that Scenario One assumes a containment relationship by a fusion agent on multiple sensor agents. For this reason and the implication that sensor agents work for fusion agents, fusion agents are a natural choice as *team leader* agents. As a member of each sensor's `context`, the fusion agent is positioned well to disseminate beliefs, goals and plans to sensor agents (the team members).

The greatest challenge presented by this situation was the recruitment of sensors. In the complex surveillance area there exists sixteen sensors, four target objects and hence four fusion agents. A simple division of sensors into four equal teams is not an appropriate real-world solution as the number of objects being tracked at any moment in time is likely to vary. An appropriate solution is one that encourages flexible team sizes, making full use of sensor agents when demand is low (by allowing larger teams) but also ensuring that sensor agents are available to track a new target. Fusion agents have, up to this point, been entirely ignorant of other fusion agents. Now a fusion agent must consider the needs of other fusion agents and/or the movement of the objects the other fusion agents are tracking. This consideration need not take the form of explicit representation of another agent's beliefs but does imply that a fusion agent must be willing to accept that a sensor agent may not be available to track its target and accept that sensors may leave their `content` despite being within range of its target (to track another fusion agent's target for example). Attempts to achieve this level of cooperation between sensors began by employing broadcast messaging, with the purpose of informing one another of their current target. It was hoped, in this way, that

a sensor agent might be able to reason over the decision to switch their tracking target, if they received messages informing them that another sensor is tracking (or not tracking) the same object. However, this solution presented practical difficulties as a consequence of the number of messages, thought to be due to the necessary synchronisation of the (Java implemented) in-boxes.

Adapting Scenario Two to the complex surveillance area

Throughout Scenario Two we have regarded sensors as service providers that are approached by fusion agents for the service they provide. If any suggestion of hierarchy exists in the relationship between sensor and fusion agent then the 'higher' ground is occupied by the sensor. Thus, unlike the previous scenario, it is not natural for the fusion agent to adopt the role of team leader. For this scenario, we adopted a peer relationship between fusion and sensor agents and chose to create a new agent to fulfill the team role.

This *tracking team* comprises of two distinct memberships; sensor agents and fusion agents. Each member is interested in tracking a common object. Sensor agents may be members of multiple teams — if it is able to detect more than one target object — but fusion agents belong to exactly one team. In fact the creation of the team agent is performed by the fusion agent when it receives notification from the environment of its target object.

```
receive(enviornment, target(T,Location))
  => NEXT create(trackingTeam);
...
receive(self, newTeam(Team)) & tracking(Target) & targetIn(Region)
  => NEXT send(Team, target(Target,Region));
```

The generated team then recruits sensors from appropriate regions. This structure provided a convenient way to combine team leadership and a coarse filtering of sensor messages, greatly reducing the work load of the fusion agent. On analysis of multiple executions, the fusion agent in this experiment has a low work load relative to the number of sensors in the example.

7.2 Results

In evaluating the implemented scenarios we were interested in ensuring that the system requirements were upheld by the specification, i.e. that all target objects were tracked accurately and that METATEM was able to generate a valid model for each execution. But we are most interested in evaluating the benefit of taking an agent-oriented approach, indeed a principled and logical approach to agency, and in particular the flexible agent grouping mechanism that the current METATEM implementation provides.

First we present, in Table 7.2, a summary of some statistical data collected from repeated executions of the two scenario for each of the surveillance areas. The data provides some insight into the execution performance of each scenario and helps inform implementation design by indicating which agents, if any, may be under or over burdened. Whilst it is clear that METATEM does not provide a performance advantage, of

Scenario	Scenario One			Scenario Two		
	simple	extended	complex	simple	extended	complex
Predicates / state						
fusion agents	8	12	29	8	15	14
sensor agents	2	4	10	2	6	11
delegate agents	-	8	-	-	-	-
team agents	-	-	-	-	-	20
Tracking lag / sec	<1	3	>5	<1	5	>5

Table 7.2: Average values for various performance measures.

any kind, over conventional asynchronous programming techniques. We do believe this work demonstrates that the declaration of temporal specifications is an appropriate one for programming agents and also provides an advantage for the high-level programming of asynchronous processes in pervasive computing applications such as the surveillance area studied. These advantages include:

- specification closely matches intuition of behaviour;
- behaviour is assured through direct execution of specifications;
- code is clear and concise;
- its closeness to agent-oriented techniques, in particular the autonomy of agents; and
- it provides a means of describing adaptable behaviour.

For instance, less than twenty lines of code specifies the sensor's behaviour in both the simple and extended experiments. The sensor fulfills obligations to provide sensor data to its context and gracefully stops providing data when told it is no longer of service. An indication of the generality of the `sensor.agent` code is given by the fact that minimal changes were necessary when scaling between the simple and extended experiments. Modification of METATEM's declarative code, when necessary, can be achieved with

greater ease than the modification of a procedural program, as semantics is unaffected by declaration ordering. For this reason the runtime modification of declarative (and interpreted) programs is possible.

7.3 Usability

One of the claimed benefits of an agent-oriented, as opposed to object-oriented, approach to software design and development, is the ability to achieve a closer correspondence between the intuition of the problem being solved and the resultant solution's source code [134]. This argument is certainly borne by isolated snippets of METATEM code, where the unambiguous nature of its operators correspond closely to human interpretations, and where its predicates are directly related to predicates of natural languages. From this perspective, the barriers to entry for new METATEM developers are low, in comparison to new C++ developers, for example.

However, whilst it is possible, with care, to read a large block (of the order of one hundred lines) of imperative language code and assimilate it sufficiently to predict its purpose and even find some types of error, reading a similarly sized block of declarative language code such as METATEM's does not lead to the same understanding. Of course, one should not expect to gain an intimate appreciation of a METATEM program, that could potentially provide an infinite number of possible executions. Particularly as it may not be clear to the developer how the solution is to be achieved — only the character of the solution may be known. Yet understanding blocks of code a fraction of this size, for the purposes of debugging, proved to be quite difficult.

Following our experiences of using concurrent METATEM and those of its early users, the place of METATEM as a high-level specification language, used in conjunction with other, more computationally efficient lower-level languages, cannot be over emphasised. This argument is supported by Michael Cieslar's work [19] which used the system extensively. During his work, Michael found METATEM appropriate for modelling complex multi-agent systems, but also found it both necessary and useful to create a majority of Java code, and a thin, clear layer of METATEM code to 'tie' it together. Whilst it is difficult to give a general rule for the ideal proportions of 'high-level specification' and 'low-level computation' code, our experience would suggest that if specification code represents around five percent of the total number of lines written, then this is a manageable amount that is also likely to provide the desired benefits. An interesting comparison with AgentSpeak can be made by analysing the Jason entry to the 2008 Multi-Agent Programming Contest [80], in which approximately fifteen percent, of the five thousand lines written, were written in AgentSpeak (the remainder being written in Java). The difference between the METATEM proportion and the Jason proportion can be explained by the nature and syntax of an AgentSpeak plan, which is a more self-contained and ordered unit, and is executed as such. Thus AgentSpeak

has a degree of plan structuring that METATEM does not, allows a Jason developer to more easily manage a greater proportion of high-level (AgentSpeak) code.

7.4 Implementation

There have been other implementations of METATEM in various guises and implementation languages, each of which served their immediate purposes well, but were not maintained beyond them. The history of these implementations indicated that constructing a reliable and maintainable implementation was a difficult task. Therefore, in an attempt to prevent this project's main output from suffering a similar fate, good software engineering principles were adopted throughout.

The benefits of having a code-base that is appropriately modularised, consistently styled, fully documented, employs recognised programming patters, and defines its own meaningful exceptions (handling them gracefully on most occasions), have already been realised. At no time has the code grown out of control, colleagues have found the source code accessible and when errors are identified, fixes are easy to apply. Having the code controlled by the version control system, Subversion[2], has provided security and further increases its accessibility.

Improvements can, of course, always be made and the immature graphical agent viewer is a good candidate for future work. The viewer provides a graphical view of agents, in the, now conventional, METATEM style (see Figure 7.3) and promises to be of great value to debugging of agent systems. Tools such as this, including simple syntax highlighting editors would greatly increase the potential uptake of the language, particularly if an online version were made available for students. Indeed, integrating the current interpreter and visualiser into a browser applet is feasible. Another significant aspect of the system's implementation, that also affects its usability, is its deliberation performance. Care must be taken not to introduce unnecessary disjunctions as their impact on deliberation time is substantial. The blocks-world scenario is a classic planning problem in AI that has been shown to be NP-hard [72] and therefore provided an ideal performance test for the implementation. Using the blocks-world solution as a benchmark, the implementation's memory and CPU usage has been profiled, and this has led to a number of efficiency improvements. However, the emphasis during development has been consistently placed on producing correct and maintainable code, with execution efficiency being of secondary concern. The execution of temporal formulas by forward chaining is inherently inefficient due to the nature of the underlying logic and particularly so if no specific optimisation measures are taken. There are no doubt opportunities for improving the execution speeds. For these reasons, optimising the implementation for both memory and CPU usage, may well be a fruitful future activ-

[2] `http://subversion.apache.org/`

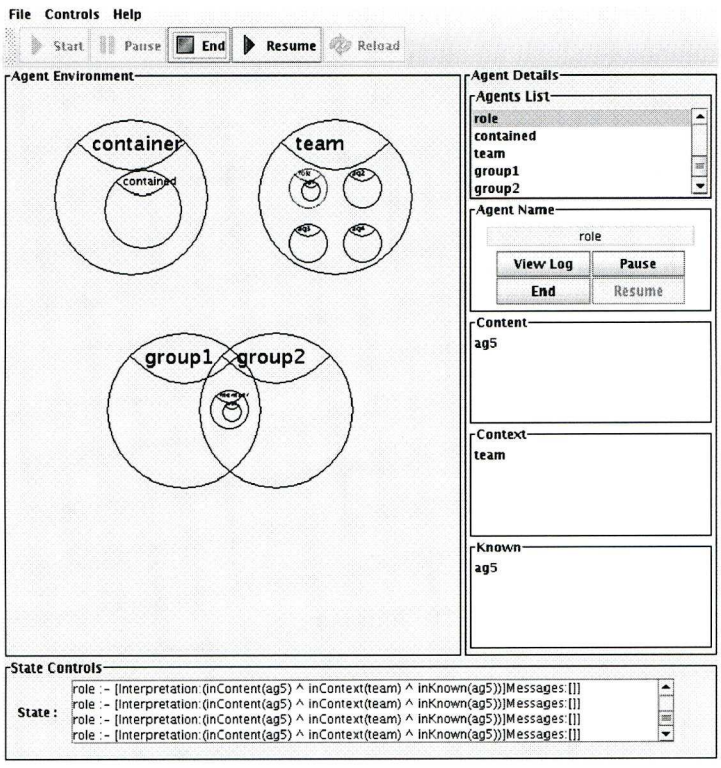


Figure 7.3: A visualisation tool for the control and monitoring of agents.

ity. Specifically, the conjoining of disjunctions when generating an agent's choices is an area where improvements could be made, perhaps by caching the results of frequently performed conjoining operations.

Chapter 8

Conclusions

This thesis has brought together formal specification, agent-oriented programming and modelling of agent organisations to demonstrate a simple and semantically coherent framework for programming pervasive computing systems comprising many 'agents'. The implementation has been shown to be useful in areas in which conventional programming frameworks find it difficult, such as concurrency and distribution. It has provided a reliable and maintainable programming platform, vital to the future development of this technique.

Chapters 1 and 2 described the central concepts of the thesis, which its title makes plain, but did so within a context of ubiquitous and pervasive computing. We have seen how the applications of computing are widening and their support for every-day human activity is increasing. Smart phone applications such as Layar and others mentioned in Section 1.2 illustrate the power available to hand-held devices today, and give us glimpses of the potential applications of tomorrow, if only we can harness the collective power of these devices. We have seen that current programming paradigms are not equipped with the constructs to model essential concepts such as autonomy, context and adaptivity, at least, not as first-class entities of the language. These chapters described some respected agent-oriented programming languages that are inspired by the archetypal Procedural Reasoning System, and surveyed a variety of existing techniques for modelling, reasoning about and programming with, context. Finally, we introduced a temporal logical framework that later gave our agents a heart beat.

In Chapter 3 we described the foundational theory behind concurrent METATEM, an agent-oriented, declarative, specification language that allows multiple agent specifications to be asynchronously executed. Having provided temporal semantics, this chapter goes on to demonstrate and formalise the execution algorithm of an agent. An execution which, providing a fair ordering of goals is applied, is guaranteed to complete, if possible. Though we do not pursue this topic further here, this opens up the potential for automated verification of an agent specification. Chapter 3 describes the implementation provided by this project, how it differs from previous implementations due to its

robust execution and maintainable code-base, as well as the extended features it provides. Features that include maintenance and manipulation of sets of terms, a range of built-in predicates, an abilities interface for executing arbitrary blocks of Java code, meta-predicates for dynamic modification of an agents specification and deliberation preferences.

It has been shown, in Chapter 4 that the METATEM language and therefore this implementation is able to model a range of multi-agent concepts such as *sharing capabilities*, *teamwork* and *roles*. Concepts which, we believe, are complementary to any agent-oriented solution to the problem of programming pervasive systems. Two distinct views of agent organisation exist within the research community; an agent-centred view, in which all entities identified during analysis of a domain are considered to be agents for the purposes of implementation, and an organisation-centred view, in which an organisation is a first-class entity with different attributes to that of an agent. In the latter, an organisation entity can be likened to a supporting environment in which agents can operate, as is exemplified by the institution abstraction, but crucially, where the organisation/institution does not exhibit autonomy, hold beliefs or otherwise behave in an agent-like fashion. On the other hand, an agent-centred viewpoint accepts that organisations of all kinds can be ascribed beliefs and act autonomously. This viewpoint is particularly useful for less infrastructural concepts such as teams, where a clear argument for team beliefs and goals exists. This work subscribes to an agent-centred view of agent-oriented programming, in which all entities identified during analysis of a problem are considered to be agents for the purposes of implementation. In Chapter 5 we showed that this does not preclude the ability to represent a range of agent organisations, and provides the additional benefit that an organisation entity can simultaneously be treated as an agent, and vice versa.

As a diversion from the main thread of research, Chapter 5 demonstrates that the simplicity and semantic clarity of the content/context constructs along with a mechanism for imposing constraints, could be used in order to provide a common underlying semantics for the implementation of agent organisation across BDI based agent programming languages. This in turn, would provide consistency of agent-organisation semantics across languages and hence increased support for development and analysis tools such as verification. Chapter 5 also provided further demonstration of the use of context for simple multi-agent applications. It should be made clear, that although it is possible in principle, to verify an agent specification, the introduction of any external influence on an agent, or any access it is given to its external environment, including the message passing, significantly reduces the possibility of automatic verification. Perhaps limiting it to only highly restricted fragments of an individual agent's behaviour.

Throughout this thesis some of the many relevant contributions to this research area, and closely related areas, have been included. Some of these contributions, such

as Cohen and Levesque's theory of teamwork [24] and other cooperation/organisation theories, provide a background that is essential for the full understanding of this work but do not represent competing or alternative approaches to this work. Others, such as the Jason interpreter for AgentSpeak, infrastructures for managing context information (e.g. JCAF [3] and the Context Toolkit [37]) and Milner's bigraph theory of interacting processes, represent competing or alternative approaches to the programming problems addressed by this work but which differ in one or more aspect from this work. Furthermore, there exist other techniques for employing temporal logic, such as deduction by resolution and model checking [54] that are closely linked to, but neither foundational to, nor competing with, this work. Thus, as far as the author is aware, the approach taken by this work is unique due to its combination of the direct execution of temporal logic, the use of multi-agent abstractions and the consideration of context as a first-class programming construct.

It cannot be denied that many other problems, in addition to the programming problem, are presenting barriers to the realisation of Weiser's 'disappearing hardware' vision [130]. Hardware technologies do appear to be well progressed, devices are indeed becoming smaller and gaining more processing capacity. Advances in hardware technologies also prevent power-consumption levels from rising in line with processing capacity, but improvements of many orders of magnitude must still be made if button-sized devices are to be endowed with the level of processing power required by current software. The security and dependability of systems (not withstanding human factors) is improving, with the help of, for example, cryptographic methods. Yet there still exists a void between the trustworthiness of encryption and the amount the public is willing to trust it. The shape of human-computer interaction must also change substantially if computers are to operate more autonomously. The fact that the prevalent mode of human-computer interaction currently involves a one hundred key keyboard and mouse is as much a testament to their success as it is to the difficulty of finding a compelling alternative. This project recognises these challenges but considers them to be parallel concerns, focusing instead on the programming problem. Thus, assuming that a secure and reliable hardware infrastructure that is open to a myriad of heterogeneous devices and mobile devices is a possibility, in Chapters 6 and 7 we demonstrated that concurrent METATEM has the potential to provide a high-level specification of system-wide behaviour of typical pervasive computing applications, and in a way not possible with conventional imperative languages. This is also the view reflected by a small but growing number of users.

System characteristics

Section 2.1.3 describes a number of characteristics of pervasive computing systems, proposed by Dobson and Nixon [40, 41]. This list of characteristics was used to inform

decision making throughout the duration of this project and is repeated below, along with any conclusions about this work, that we draw with respect to each item on the list.

(a) **Events are too noisy to serve directly as a basis for programming.**
Although an event-driven programming model may not be the ideal, this statement does imply that events remain a useful metaphor. It may be that the popular event/event-listener architectures can be employed to good effect, in sub-system components. Indeed, our approach uses the agent metaphors of percepts, beliefs and messages for the high level system-wide programming, but also allows conventional event-driven models to be employed at lower levels (as add-on Java code, for example). We believe this provides not only a better basis for programming but also a more flexible one.

(b) **Don't take anyone's word for anything.**
Trust is an important concept in the real-world and also in the world of multi-agent research. This statement actually refers to the inaccuracy and noise that must be expected from electronic sensors embedded within an environment, but it has a deliberately human tone. We believe that taking an abstract approach to the accuracy of sensors, by modelling the sensors as trustworthy (or otherwise) agents is natural and appropriate.

(c) **Interconnection is more important than data.**
We have argued that agents and the relationships between agents are suitable abstractions to describe the high-level of interconnection that such system will undoubtedly possess. The agent-organisation techniques discussed and demonstrated in this work lead us to conclude that they are indeed suitable for some scenarios. However, it may be that alternative techniques for coordinating or analysing interconnected entities, such as algebraic topology [33, 7], are more useful in other scenarios.

(d) **Any decision needs a mitigation strategy.**
Again, the architecture of BDI agent languages provide an inherent advantage for adapting to changing circumstances by means of belief revision and the selection of alternative plans. Furthermore, the forward chaining execution of a METATEM agent employs backtracking when an undesirable (logically inconsistent) state is reached. However, this alone is not true mitigation — in its crudest form it is simply trial-and-error. Agent languages require explicit consideration of mitigating actions and these tend to be implemented as plans, triggered by the failure of regular plans. A METATEM agent also has the concept of a reversible action, i.e. a complementary 'undo' action that is performed in the event of backtracking over

an action. Of course, only internal actions can be considered as truly defeasible and the power of these external undo actions to actually mitigate is also highly dependent upon circumstances. In conclusion, our approach does facilitate the programming of basic mitigation strategies, but an additional theory such as a defeasibility of actions theory, is likely to be more advantageous.

(e) **Everything interesting comes from composition.**
The approach taken by this work places system composition at the forefront of system design, by the integration of inter-agent relationships into the specification of agents. This approach allows a wide variety of multi-agent compositions, some of which have been demonstrated to be of use when programming typical pervasive computing scenarios in an agent-oriented way. It is hoped that future work will allow the run-time interactions between components (agents) to be better understood (analysed) at design time.

8.1 Future work

This work has provided valuable insight into the use of METATEM for typical multi-agent systems that contain many (ten or more) agents, as is expected when modelling pervasive computing scenarios as multi-agent systems. However, it has also emphasised the need for further work in this area.

It is recommended that investigation into this approach be continued. In particular, the surveillance case-study is worth pursuing, with a number of general aims. The adaptation of agents (to increased message passing load) by creation of sub- and group-agents, has been demonstrated in this work and it is hoped that such adaptation can be further automated, perhaps with the aim of producing a design pattern of *adaptation* for METATEM agents.

Although this work pushed the execution of multiple agents to the practical limits for METATEM in its current guise, it is hoped that with optimisation of the interpreter and more intelligent specification of agents, that the multi-agent systems simulated can be scaled up. Although it must be recognised that direct execution of a logical specification is by its very nature a time inefficient process.

Of course, the scaling up of experiments not only presents difficulties for the execution environment but also for the design of more complex scenarios. With this in mind, work has begun on devising and formalising a visual design methodology, specifically targeting the contextual structuring of agents. Initial work [55] has concentrated on a process of refinement whereby agent specifications are iteratively refined in such a way that the original specification is a logical entailment of the refined specification. The content/context approach naturally supports both top-down and bottom-up refinements, where agents are either decomposed to create new content agents, or composed

to form new context agents. This work has raised questions regarding the accessibility of an agent specification and the extent to which agent autonomy must be sacrificed in order to benefit from automated verification. Even so, this is an interesting development that deserves further attention.

Appendix A

Documentation

A.1 README file

```
Concurrent MetateM
==================

This software is an implementation of the agent programming
language MetateM [Fisher et al.] in which agents are
specified using a declarative language of temporal logic
rules and meta-statements. Multiple agent specifications
are interpreted asynchronously and agents are able to
communicate by message passing.

For an introduction to agent and MetateM theory, see the
following file included with this download:

  metatem_intro.pdf

This download should contain the following directories and
contents:

  /lib        -all necessary executable files to create
               your own multi-agent systems.
  /doc        -developer documentation generated by javadoc,
               including a description of the agent-ability
               API that enables a MetateM agent's interaction
               with the real world.
              -the introduction to MetateM theory mentioned above.
              -the grammar of system and agent files, in BNF.
  /examples   -some elementary single- and multi-agent examples.
```

```
  /src        -the (mainly Java) source code for this implementation.

For all else, please contact the developers:

  hepplea@liverpool.ac.uk


System requirements
-------------------

The minimum system requirement are a working Java Runtime environment,
version 1.6 or later, access to a command line and a text editor. If
your agents are to be of any practical use then you will also need a
Java development kit to compile your agents' abilities.

Getting started
---------------

There are many ways to execute MetateM. Currently the way we advise is
via the command line.

To try one of the packaged examples, use the following command from the
'examples' directory (replacing 'helloworld.sys' with the system
specification file of your choice):

$ java -jar ../lib/metatem.jar helloworld.sys

Or, having appended 'metatem/lib/metatem.jar' to your operating
system's CLASSPATH variable (see below for help with this), with the
slightly abbreviated command

$ java metatem.Main helloworld.sys

Finally, you can use the supplied GUI to load a system file (remember
that this software comes with no warranty ;-) as follows (where again,
'metatem/lib' is in your CLASSPATH):

$ java metatem.tools.Launch
```

Once you have successfully executed an example, you can use them as a basis for creating your own multi-agent systems, by referring to the documentation on-line and in the 'metatem/doc' directory of this download.

Other information

The following will not be found in this download but can be found on-line:

- A first MetateM tutorial.
- Frequently asked questions.

Appending the metatem jar file to your CLASSPATH:

Windows
> set CLASSPATH=CLASSPATH;<path>\metatem\lib\metatem.jar

Unix
$ export CLASSPATH=$CLASSPATH:<path>/metatem/lib/metatem.jar

Where <path> is replaced with the full path to the directory where you unzipped the MetateM download.

Please see the file COPYING for details of licensing.

Id

A.2 Java documentation

http://www.csc.liv.ac.uk/~anthony/metatem/javadoc/overview-summary.html

MetateM—Multi-agent Language Interpreter

This documentation describes the implementation of an interpreter for a multi-agent specification language called MetateM, an agent-oriented programming language in which the first-class entities are agents whose behaviour is specified by a mixture of temporal logic and meta-statements; for a detailed introduction to MetateM theory and this implementation see here, otherwise see below.

See:

Description

Packages	
metatem	This is the parent package for all packages and classes that comprise the MetateM multi-agent specification language.
metatem.agent	The classes and sub-packages contained in metatem.agent describe the agent-oriented behaviour of an agent.
metatem.agent.ability	Interfaces, abstract classes and some concrete classes that provide MetateM agents with the ability to act in their environment.
metatem.agent.communication	A collection of classes that enable agents to communicate by message passing.
metatem.parser	These classes have been generated by the parser generator Javacc, from the parser description file src/metatem/parser/Parser.jj.
metatem.temporal	metatem.temporal is an API for temporal logic that is intended to be entirely independent of the other metatem packages.
metatem.tools	This is the parent package for all packages and classes that provide support for the development of MetateM programs.

This documentation describes the implementation of an interpreter for a multi-agent specification language called MetateM, an agent-oriented programming language in which the first-class entities are agents whose behaviour is specified by a mixture of temporal logic and meta-statements; for a detailed introduction to MetateM theory and this implementation see here, otherwise see below.

A MetateM agent is capable of executing arbitrary Java code (and thus, by integration with Java, arbitrary code of any language) through the use of the API described by the package metatem.agent.ability. The class metatem.agent.ability.AbstractAbility will be of particular interest to a MetateM developer who wants to endow their agents with existing abilities or create custom abilities of their own.

Appendix B

Source code from Chapter 6

B.1 bob.agent

```
type agent;

logging FINE;

//Bob can only drive to one place at a time, drive(destination).
at_most 1 drive ;
at_most 1 arrive ;

prefer arrive to drive when drive(X) weight 50;

initial : {

  start => drive(home);
  start => todo(shopping);
  start => ~doing(shopping);

  // Bob should go shopping at sometime, at the shopping
  // will not be done unless he does it.
  todo(shopping) => todo(shopping) UNTIL doing(shopping);
  todo(shopping) => ~done(shopping) UNLESS doing(shopping);

  // Bob cannot do shopping until he is in the shopping_centre
  ~in(shopping_centre,context) => NEXT ~doing(shopping);

  // Bob must drive to the shopping centre
  todo(shopping) => SOMETIME drive(shopping_centre);
```

```
// If driving one moment then the next moment we should be driving
// to the same destination, arriving at that destination or
// driving to another destination.
drive(X) => NEXT arrive(X) | drive(X) | drive(Y,X);
drive(X) => SOMETIME arrive(X);
drive(X) & ~entered(X,context) => NEXT ~arrive(X);

// Bob will consider a shopping diversion when driving,
// apart from when he is driving to work.
drive(X) & todo(shopping) & X=\=shopping_centre
  => NEXT drive(shopping_centre,X) | drive(X);
drive(Y,X) => divert;
drive(X) & divert => false;
drive(X) & X=work => NEXT drive(X) | arrive(X);
drive(Y,X) => togo(X) UNTIL drive(X);


// Whilst shopping, the agent cannot drive
arrive(X) & togo(Y) => ~drive(Y) UNTIL depart(X);
in(X,context) => NEXT ~depart(X);
arrive(X) => ~drive(X); // stop driving when arriving
drive(X) & arrive(Y) => false; // also applies when X=Y
left(shopping_centre,context) => NEXT depart(shopping_centre);
left(shopping_centre,context) => NEXT drive(X); // hopefully home


// The following rules are for the purposes of the simulation and
// are placed here for convenience. Ideally they would be built
// into an environment

// Stop bob from arriving home before he has been shopping
arrive(home) => false | done(shopping);

doing(X) => doing(X) UNTIL done(X);

// Force Bob to go shopping
drive(X) & todo(shopping) & X=\=shopping_centre
  => NEXT drive(shopping_centre,X);
```

```
  // Make it a short drive to the shopping centre and then to home
  drive(shopping_centre) & ~enterContext(shopping_centre)
    => NEXT enterContext(shopping_centre);
  done(shopping) => NEXT leaveContext(shopping_centre);
  in(home,context) => NEXT end;
}
```

B.2 delegate.agent

```
type agent;

ability send : metatem.agent.ability.Send;
ability print : metatem.agent.ability.Print;

logging OFF;

/* A delegate agent represents a group of sensors on behalf of
 * a fusion agent. It has been delegated the responsibility of
 * receiving the sensor data from the sensor agents and will
 * forward, to the fusion agent, only relevant messages.
 * It can be viewed as the fusion agent's secretary, filtering
 * unwanted mail.
 */

basics : {

  // Soon after creation, this agent should receive a message
  // from its creator.
  receive(FusionAgent, tracking(Target)) & inContext(FusionAgent)
    => NEXT tracking(Target);

  // The agent is created for a specific set of agents and
  // a single target.
  tracking(Target) => NEXT tracking(Target);

  receive(FusionAgent, addToContent(SensorAgent)) & inContext(FusionAgent)
    => NEXT addToContent(SensorAgent);

  // When sensor agents have all been ejected this agent
  // will leave the Content of the fusion agent and die.
```

```
  send(Sensor, redundant) & inContent(Sensor) & size(Content,1)
    => NEXT end;
}

receiving_data : {

  // Messages from content agents are not received every moment,
  // but multiple messages (about multiple targets/noise) can be
  // received in a single moment. Sensors are given a status of
  // either sending data, only noise or nothing, which refer to
  // the immediate past.
  nothingFrom(Sensor) & onlyNoiseFrom(Sensor) => false;
  onlyNoiseFrom(Sensor) & dataFrom(Sensor) => false;
  nothingFrom(Sensor) & dataFrom(Sensor) => false;

  inContent(Sensor) & ~receive(Sensor, Message)
    => NEXT nothingFrom(Sensor);

  inContent(Sensor) & receive(Sensor, data(Colour,X,Y)) &
  tracking(Colour)
    => NEXT dataFrom(Sensor);

  inContent(Sensor) & receive(Sensor, data(noise))
    => NEXT noiseFrom(Sensor);

  inContent(Sensor) & receive(Sensor, data(Colour1,X,Y)) &
  tracking(Colour2) & Colour1=\=Colour2
    => NEXT noiseFrom(Sensor);

  noiseFrom(Sensor) & ~dataFrom(Sensor) => onlyNoiseFrom(Sensor);

  // Count the number of times an agent sends noise and ask the sensor
  // to leave your context if it sends noise three consecutive times
  enteredContent(Sensor) => NEXT noisesFrom(0,Sensor);
  dataFrom(Sensor) => NEXT noisesFrom(0,Sensor);
  onlyNoiseFrom(Sensor) & noisesFrom(X,Sensor) & (Y is X+1)
    => NEXT noisesFrom(Y,Sensor);
  nothingFrom(Sensor) & noisesFrom(X,Sensor)
    => NEXT noisesFrom(X,Sensor);
```

```
  noisesFrom(X,Sensor) & noisesFrom(Y,Sensor) & X=\=Y
    => false;
  inContent(Sensor) & noisesFrom(3,Sensor) => NEXT send(Sensor, redundant);
}

filtering : {

  // Only pass on data about the target identified by tracking(target),
  // and only send one message per time step.
  receive(Sensor, data(Colour,X,Y)) & tracking(Colour) & inContext(FusionAg)
    => NEXT send(FusionAg, data(Colour,X,Y)) | ignore(targetCoord(X,Y));

  inContext(FusionAg) & send(FusionAg, Message1) & send(FusionAg, Message2)
      & Message1 =\= Message2
    => false;
}
```

B.3 environment.agent

```
// This agent has the responsibility of creating
// the SurveillanceArea instance co-ordinating the
// initial registration of agents. It isn't exactly
// a wrapper because the SurveillanceArea instance
// does send messages directly to sensor agents once
// the sensor agents are registered.
type agent;

ability surveillance : surveillance.CreateSurveillanceArea;
ability print : metatem.agent.ability.Print;
ability send : metatem.agent.ability.Send;

logging INFO;

locations : {

  start => location(north);
  start => location(south);
  start => location(central);

  // Once a location always a location
```

```
  location(L) => NEXT location(L);
}


configuration : {

  // Start by creating and starting the surveillance area
  start => surveillance(create);
  receive(self,surveillanceArea(Area))
    => NEXT surveillance(begin,Area);

  // surveillanceArea(X) => NEXT ALWAYS surveillanceArea(X)
  receive(self,surveillanceArea(Area)) => NEXT surveillanceArea(Area);
  surveillanceArea(Area) => NEXT surveillanceArea(Area);

  // Ensure that if we have a surveillance area then all agents
  // are either attached or already attached to a sensor.
  // Agents and sensors are explicitly matched at the moment.
  receive(self,surveillanceArea(Area))
     => NEXT surveillance(attach, sensor1, Area, north, point(2,27));
  receive(self,surveillanceArea(Area))
    => NEXT surveillance(attach, sensor2, Area, north, point(18,28));
  receive(self,surveillanceArea(Area))
    => NEXT surveillance(attach, sensor3, Area, north, point(10,20));
  receive(self,surveillanceArea(Area))
    => NEXT surveillance(attach, sensor4, Area, central, point(2,15));
  receive(self,surveillanceArea(Area))
    => NEXT surveillance(attach, sensor5, Area, central, point(18,15));
  receive(self,surveillanceArea(Area))
    => NEXT surveillance(attach, sensor6, Area, south, point(10,10));
  receive(self,surveillanceArea(Area))
    => NEXT surveillance(attach, sensor7, Area, south, point(2,2));
  receive(self,surveillanceArea(Area))
    => NEXT surveillance(attach, sensor8, Area, south, point(18,2));

  // Remember which agents are attached and its location.
  receive(self,attached(X,L)) => NEXT attached(X,L);
  attached(X,L) & ~receive(environment, detached(X))
    => NEXT attached(X,L);
```

```
  attached(X,L) => attached(X);

  // An agent must not be attached more than once.
  attached(Sensor) & surveillance(attach, Sensor, Area, Location, Point)
    => false;
  attached(X,L1) & attached(X,L2) & L1=\=L2 => false;

  // Once the area is started, get location of target(s).
  receive(self,areaStarted(Area)) => NEXT surveillance(targets,Area);

  // Inform the fusion agent of the target's colour and location
  // whenever this information is received from the environment.
  receive(self,target(red,Location))
    => NEXT send(target1, target(red,Location));
  receive(self,target(green,Location))
    => NEXT send(target2, target(green,Location));
}

communication: {
  // Broadcast all 'broadcast' messages to all members of content:
  receive(X, broadcast(M)) & inContent(X) & inContent(Y) & X=\=Y
    => NEXT send(Y, M);
}
```

B.4 fusion.agent

```
type agent;

ability send : metatem.agent.ability.Send;
ability createGroup : surveillance.CreateGroup;

logging FINE;

at_most 1 tracking;
at_most 1 targetCoord;

delegates : { }

basics : {
```

```
  // A fusion agent must be given a target to track,
  // but can only track one target at a time.
  receive(environment, target(T,Location)) => NEXT tracking(T);

  // Only stop tracking if the environment proposes
  // another target
  tracking(X) & ~receive(environment,target(Y,L))
    => NEXT tracking(X);

  // On receipt of a new target, broadcast for sensors within
  // range of the target
  receive(environment, target(T,Location))
    => NEXT send(environment,broadcast(requestDataFor(self,Location)));

  targetCoord(X,Y) => NEXT targetInSight;
}

receiving_data : {

  // Messages from content agents are not received every moment,
  // but multiple messages (about multiple targets/noise) can be
  // received in a single moment. Sensors are given a status of
  // either sending data, only noise or nothing, which refer to
  // the immediate past.
  nothingFrom(Sensor) & onlyNoiseFrom(Sensor) => false;
  onlyNoiseFrom(Sensor) & dataFrom(Sensor) => false;
  nothingFrom(Sensor) & dataFrom(Sensor) => false;

  inContent(Sensor) & ~receive(Sensor, Message)
    => NEXT nothingFrom(Sensor);

  inContent(Sensor) & receive(Sensor, data(Colour,X,Y)) &
  tracking(Colour)
    => NEXT dataFrom(Sensor);

  inContent(Sensor) & receive(Sensor, data(noise))
    => NEXT noiseFrom(Sensor);

  inContent(Sensor) & receive(Sensor, data(Colour1,X,Y)) &
```

```
  tracking(Colour2) & Colour1=\=Colour2
    => NEXT noiseFrom(Sensor);

  noiseFrom(Sensor) => onlyNoiseFrom(Sensor) | dataFrom(Sensor);
  dataFrom(Sensor) => ~onlyNoiseFrom(Sensor);

  // Count the number of times an agent sends noise and ask the sensor
  // to leave your context if it sends noise three consecutive times
  enteredContent(Sensor) => NEXT noisesFrom(0,Sensor);
  dataFrom(Sensor) => NEXT noisesFrom(0,Sensor);
  onlyNoiseFrom(Sensor) & noisesFrom(X,Sensor) & (Y is X+1)
    => NEXT noisesFrom(Y,Sensor);
  nothingFrom(Sensor) & noisesFrom(X,Sensor)
    => NEXT noisesFrom(X,Sensor);
  noisesFrom(X,Sensor) & noisesFrom(Y,Sensor) & X=\=Y
    => false;
  inContent(Sensor) & noisesFrom(3,Sensor) => NEXT send(Sensor, redundant);
}

tracking : {

  // Sensor agents send messages containing sensor data of the form
  // data(Colour,X,Y)  or data(noise) if no targets are detected
  receive(Sensor, data(Colour,X,Y)) & tracking(Colour)
    => NEXT targetCoord(X,Y) | ~targetCoord(X,Y);

  // The general location of a target is split into the regions
  // north, central and south. Note that origin (0,0) is in the
  // south west and that these regions do not overlap.
  targetCoord(X,Y) & (Y<10) => targetIn(south);
  targetCoord(X,Y) & (Y<20) & (9<Y) => targetIn(central);
  targetCoord(X,Y) & (19<Y) => targetIn(north);

  // The general movement between regions is tracked.
  targetIn(Region) => NEXT targetPreviouslyIn(Region);
  targetIn(Region) & ~targetPreviouslyIn(Region)
    => NEXT targetMovedInto(Region);
  targetIn(Region) & targetPreviouslyIn(Region)
    => NEXT ~targetMovedInto(Region);
```

```
  // When moving into a region, broadcast a request for sensors
  // in that region
  targetMovedInto(Region)
    => send(environment, broadcast(requestDataFor(self, Region)));

  // When no data is received in the next state the fusion agent
  // assumes the target has not moved
  targetCoord(X,Y) & tracking(Colour) & ~receive(Sensor, data(Colour,X,Y))
    => NEXT targetCoord(X,Y);
}

managing_sensors : {

  // Create a group when content size exceeds 2.
  size(Content,X) & size(delegates,Y) & (Z is X-Y) & (2<Z) &
  ~awaitingGroup & ~createdGroup
    => NEXT createGroup(sensorGroup);
  createGroup(X) => createdGroup;
  createGroup(X) & ~receive(self, newGroup(Y)) => NEXT awaitingGroup;
  awaitingGroup & ~receive(self, newGroup(X)) => NEXT awaitingGroup;
  receive(self, newGroup(X)) => NEXT ~awaitingGroup;

  // When a new group is received add all sensors in content.
  // (Not, the group itself or any other groups previously formed.)
  receive(self, newGroup(X)) & inContent(Y) & X=\=Y & ~in(Y,delegates)
    => NEXT send(X, addToContent(Y));
  receive(self, newGroup(X)) & inContent(Y) & X=\=Y & ~in(Y,delegates)
    => NEXT removeFromContent(Y);
  receive(self, newGroup(X)) => NEXT add(X,delegates);

  // Tell the newly created delegate agent which target data to pass on.
  receive(self, newGroup(X)) & tracking(Y)
    => NEXT send(X, tracking(Y));
}
```

Bibliography

[1] Ian Akyildiz, Weilian Su, Yogesh Sankarasubramaniam, and Erdal Cayirci. A survey on sensor networks. *IEEE Communications Magazine*, 40(8):102–114, 2002.

[2] Jakob Bardram. Applications of Context-Aware Computing in Hospital Work: Examples and Design Principles. In *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, pages 1574–1579, New York, NY, USA, 2004. ACM.

[3] Jakob Bardram. The Java Context Awareness Framework (JCAF) A Service Infrastructure and Programming Framework for Context-Aware Applications. In *In Proceedings of the 3rd International Conference on Pervasive Computing (Pervasive 2005)*, Lecture Notes in Computer Science, pages 98–115, Munich, Germany, 2005. Springer Verlag.

[4] Jakob Bardram and Thomas Hansen. The AWARE Architecture: Supporting Context-mediated Social Awareness in Mobile Cooperation. In *CSCW '04: Proceedings of the 2004 ACM conference on Computer supported cooperative work*, pages 192–201, New York, NY, USA, 2004. ACM.

[5] Howard Barringer, Michael Fisher, Dov Gabbay, Graham Gough, and Richard Owens. METATEM: An Introduction. *Formal Aspects of Computing*, 7(5):533–549, 1995.

[6] Howard Barringer, Michael Fisher, Dov Gabbay, Richard Owens, and Mark Reynolds, editors. *The Imperative Future: Principles of Executable Temporal Logic*. Research Studies Press, May 1996.

[7] Yuliy Baryshnikov and Robert Ghrist. Target Enumeration via Euler Characteristic Integrals. *SIAM Journal on Applied Mathematics*, 70(3):825–844, 2009.

[8] Claudio Bettini, Oliver Brdiczka, Karen Henricksen, Jadwiga Indulska, Daniela Nicklas, Anand Ranganathan, and Daniele Riboni. A Survey of Context Modelling and Reasoning Techniques. *Pervasive and Mobile Computing*, 6(2):161–180, June 2009.

[9] Olivier Boissier, Julian Padget, Viginia Dignum, Gabriella Lindemann, Eric Matson, Sascha Ossowski, Jaime Sichman, and Javier Vázquez-Salceda, editors. *Coordination, Organization, Institutions and Norms in agent systems (COIN).* Springer-Verlag, 2006.

[10] Rafael Bordini, Michael Fisher, Willem Visser, and Michael Wooldridge. Verifying Multi-Agent Programs by Model Checking. *Journal of Autonomous Agents and Multi-Agent Systems*, 12(2):239–256, 2006.

[11] Rafael Bordini and Jomi Hübner. BDI Agent Programming in AgentSpeak Using *Jason* (Tutorial Paper). In Francesca Toni and Paolo Torroni, editors, *CLIMA VI*, volume 3900 of *Lecture Notes in Computer Science*, pages 143–164. Springer, 2005.

[12] Rafael Bordini, Jomi Hübner, and Renata Vieira. Jason and the Golden Fleece of Agent-Oriented Programming. In Bordini et al. [15], pages 3–37.

[13] Rafael Bordini, Michael Wooldridge, and Jomi Hübner. *Programming Multi-Agent Systems in AgentSpeak using Jason (Wiley Series in Agent Technology).* John Wiley & Sons, 2007.

[14] Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah Seghrouchni, editors. *Multi-Agent Programming: Languages, Tools and Applications.* Springer Publishing Company, Incorporated, 2009.

[15] Rafail Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah Seghrouchni, editors. *Multi-Agent Programming: Languages, Platforms and Applications*, volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations.* Springer-Verlag, Heidelberg, Germany, 2005.

[16] Paolo Bresciani, Anna Perini, Paolo Giorgini, Fausto Giunchiglia, and John Mylopoulos. Tropos: An Agent-Oriented Software Development Methodology. *Autonomous Agents and Multi-Agent Systems*, 8:203–236, 2004.

[17] Lawrence Cavedon, Anand Rao, and Gil Tidhar. Social and Individual Commitment. In *PRICAI '96: Proceedings from the Workshop on Intelligent Agent Systems, Theoretical and Practical Issues*, pages 152–163, London, UK, 1997. Springer-Verlag.

[18] Harry Chen, Filip Perich, Tim Finin, and Anupam Joshi. SOUPA: Standard Ontology for Ubiquitous and Pervasive Applications. *Mobile and Ubiquitous Systems, Annual International Conference on*, 0:258–267, 2004.

[19] Michael Cieslar. Adapting METATEM for the Multi-Agent Programming Contest. Manuscript, 2009. Honours Year Project, Dept. Computer Science University of Liverpool.

[20] Helder Coelho. Future Challenges for Autonomous Systems. In *Artificial Intelligence An International Perspective*, volume 5640/2009 of *Lecture Notes in Computer Science*. Springer, 2009.

[21] Cognative Robotics Research Group WWW Page, June 2010. `http://www.cs.toronto.edu/cogrobo/main/systems/index.html`.

[22] Philip Cohen and Hector Levesque. Intention is Choice with Commitment. *Artif. Intell.*, 42(2-3):213–261, 1990.

[23] Philip Cohen and Hector Levesque. Confirmations and Joint Action. In *IJCAI*, pages 951–959, 1991.

[24] Philip Cohen and Hector Levesque. Teamwork. Technical Report 504, Centre for Study of Language and Information, SRI International, Menlo Park, CA, 1991.

[25] Joëlle Coutaz, James Crowley, Simon Dobson, and David Garlan. Context is Key. *Communication of the ACM (CACM)*, 48(3):49–53, 2005.

[26] Joëlle Coutaz and Gaëtan Rey. Foundations for a Theory of Contextors. In *Proceedings of Computer-Aided Design of User Interfaces (III)*, pages 13–32. Kluwer Academic Publishers, 2002.

[27] Mehdi Dastani, Frank de Boer, Frank Dignum, and John-Jules Ch. Meyer. Programming Agent Deliberation: An Approach Illustrated Using the 3APL Language. In *AAMAS '03: Proceedings of the 2nd international joint conference on Autonomous agents and multiagent systems*, pages 97–104, New York, NY, USA, 2003. ACM.

[28] Mehdi Dastani, Jürgen Dix, and Peter Novák. The Multi-Agent Programming Contest WWW Page, June 2010. `http://www.multiagentcontest.org`.

[29] Mehdi Dastani, M. Birna van Riemsdijk, and John-Jules Ch. Meyer. Programming Multi-Agent Systems in 3APL. In Bordini et al. [15].

[30] Nivea de Carvalho Ferreira, Michael Fisher, and Wieve van der Hoek. A Logical Implementation of Uncertain Agents. In Carlos Bento, Amílcar Cardoso, and Gaël Dias, editors, *Progress in Artificial Intelligence*, volume 3808 of *Lecture Notes in Computer Science*, pages 536–547. Springer, 2005.

[31] Giuseppe De Giacomo, Yves Lespérance, and Hector Levesque. ConGolog, A Concurrent Programming Language Based on the Situation Calculus. *Artificial Intelligence*, 121(1–2):109–169, 2000.

[32] Giuseppe De Giacomo, Yves Lespérance, Hector Levesque, and Sebastian Sardina. *IndiGolog: A High-Level Programming Language for Embedded Reasoning Agents*, pages 31–72. In Bordini et al. [14], 2009.

[33] Vin de Silva and Robert Ghrist. Coordinate-free Coverage in Sensor Networks with Controlled Boundaries via Homology. *The International Journal of Robotics Research*, 25(12):1205–1222, December 2006.

[34] Daniel Dennett. *The Intentional Stance (Bradford Books)*. The MIT Press, Cambridge, MA, March 1987.

[35] Louise Dennis, Berndt Farwer, Rafael Bordini, Michael Fisher, and Michael Wooldridge. A Common Semantic Basis for BDI Languages. In *Proc. Seventh International Workshop on Programming Multiagent Systems (ProMAS)*, Lecture Notes in Artificial Intelligence. Springer Verlag, 2007.

[36] Anind Dey. Understanding and Using Context. *Personal and Ubiquitous Computing*, 5:4–7, 2001.

[37] Anind Dey, Daniel Salber, and Gregory Abowd. A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications. *Human-Computer Interaction (HCI)*, 16 (2–4), 2001.

[38] Clare Dixon, Michael Fisher, and Boris Konev. Temporal Logic with Capacity Constraints. In *Proc. 6th International Symposium on Frontiers of Combining Systems*, volume 4720 of *Lecture Notes in Computer Science*, pages 163–177. Springer, 2007.

[39] Simon Dobson, Spyros Denazis, Antonio Fernández, Dominique Gaïti, Erol Gelenbe, Fabio Massacci, Paddy Nixon, Fabrice Saffre, Nikita Schmidt, and Franco Zambonelli. A Survey of Autonomic Communications. *ACM Trans. Auton. Adapt. Syst.*, 1(2):223–259, 2006.

[40] Simon Dobson and Paddy Nixon. More Principled Design of Pervasive Computing Systems. In *Engineering Human Computer Interaction and Interactive Systems*, volume 3425/2005. Springer Berlin, 2005.

[41] Simon Dobson and Paddy Nixon. Whole-system Programming of Adaptive Ambient Intelligence. In *Proceedings of HCI International*, volume 6. Springer-Verlag, 2007.

[42] Marc Esteva, David de la Cruz, and Carles Sierra. ISLANDER: an electronic institutions editor. In *AAMAS '02: Proceedings of the 1st international joint conference on Autonomous agents and multiagent systems*, pages 1045–1052, New York, NY, USA, 2002. ACM.

[43] Marc Esteva, Juan-Antonio Rodrguez-Aguilar, Carles Sierra, Pere Garcia, and Josep Arcos. *On the Formal Specification of Electronic Institutions*, volume 1991, pages 126–147. Spriger, 2001.

[44] Jacques Ferber and Olivier Gutknecht. A Meta-model for the Analysis and Design of Organizations in Multi-agent Aystems. In *Proceedings of the 3rd International Conference on Multi-Agent Systems (ICMAS98)*, pages 128–135, 1998.

[45] Jacques Ferber, Olivier Gutknecht, and Fabien Michel. From Agents to Organizations: An Organizational View of Multi-agent Systems. In *AOSE*, pages 214–230, 2003.

[46] Michael Fisher. A Normal Form for First-Order Temporal Formulae. In *CADE-11: Proceedings of the 11th International Conference on Automated Deduction*, pages 370–384, London, UK, 1992. Springer-Verlag.

[47] Michael Fisher. Concurrent METATEM—A language for modelling reactive systems. In *Parallel Architectures and Languages Europe (PARLE)*, LNCS, pages 185–196. Springer, 1993.

[48] Michael Fisher. A Survey of Concurrent METATEM—The Language and its Applications. In *Proceedings of International Conference on Temporal Logic, ICTL'94*, volume 827 of *Lecture Notes in Computer Science*, pages 480–505. Springer, 1994.

[49] Michael Fisher. A Temporal Semantics for Concurrent METATEM. *Journal of Symbolic Computation*, 22(5/6):627–648, 1996.

[50] Michael Fisher. A Normal Form for Temporal Logics and its Applications in Theorem-Proving and Execution. *Journal of Logic and Computation*, 7(4):429–456, August 1997.

[51] Michael Fisher. Implementing BDI-like Systems by Direct Execution. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI)*, volume 1, pages 316–321, San Fransisco, CA, USA, 1997. Morgan Kaufmann.

[52] Michael Fisher. METATEM: The Story so Far. In *Programming Multi-Agent Systems II (PROMAS)*, volume 3862 of *Lecture Notes in Artificial Intelligence*, pages 3–22, Heidelberg, Germany, 2005. Springer-Verlag.

[53] Michael Fisher. Agent Deliberation in an Executable Temporal Framework. Technical Report ULCS-08-014, Department of Computer Science, University of Liverpool, UK, July 2008.

[54] Michael Fisher. *An Introduction to Practical Formal Methods using Temporal Logic.* In preparation.

[55] Michael Fisher, Louise Dennis, and Anthony Hepple. Modula Multi-Agent Design. Technical Report ULCS-09-002, Department of Computer Science, University of Liverpool, January 2009.

[56] Michael Fisher, Chiara Ghidini, and Benjamin Hirsch. Organising Logic-Based Agents. In *Formal Approaches to Agent-Based Systems*, volume 2699 of *Lecture Notes in Computer Science*, pages 15–27. Springer-Verlag, October 2003.

[57] Michael Fisher, Chiara Ghidini, and Benjamin Hirsch. Organising Computation through Dynamic Grouping. In *Objects, Agents, and Features*, pages 117–136, 2004.

[58] Michael Fisher, Chiara Ghidini, and Benjamin Hirsch. Programming Groups of Rational Agents. In *Computational Logic in Multi-Agent Sytems (CLIMA-IV)*, volume 3259 of *Lecture Notes in Computer Science*. Springer-Verlag, November 2004.

[59] Michael Fisher and Anthony Hepple. *Executing Logical Agent Specifications*, pages 3–29. In Bordini et al. [14], 2009.

[60] Michael Fisher and Antony Kakoudakis. Flexible Agent Grouping in Executable Temporal Logic. In *Proceedings of Twelfth International Symposium on Languages for Intensional Programming (ISLIP)*. World Scientific Press, 1999.

[61] Mark Fox. An Organizational View of Distributed Systems. *IEEE Transactions on Systems, Man, and Cybernetics*, 11, 1981.

[62] Dov Gabbay. Declarative Past and Imperative Future: Executable Temporal Logic for Interactive Systems. In Behnam Banieqbal, Howard Barringer, and Amir Pnueli, editors, *Proceedings of Colloquium on Temporal Logic in Specification*, pages 402–450, Altrincham, U.K., 1987. Published in *Lecture Notes in Computer Science*, volume 398, Springer-Verlag.

[63] Michael Georgeff and Amy Lansky. Reactive Reasoning and Planning. In *AAAI*, pages 677–682, 1987.

[64] Michael Georgeff and Anand Rao. A profile of the Australian Artificial Intelligence Institute. *IEEE Expert: Intelligent Systems and Their Applications*, 11(6):89–92, 1996.

[65] Ubiquitous Computing: Experience, Design and Science (Ubiquitous Computing Grand Challenge: Manifesto), Februrary 2006. `http://www-dse.doc.ic.ac.uk/Projects/UbiNet/GC/Manifesto/manifesto.pdf`.

[66] Ubiquitous Computing Grand Challenge WWW page. `http://www-dse.doc.ic.ac.uk/Projects/UbiNet/GC/index.html`.

[67] Adam Greenfield. *Everyware.* New Riders Publishing, 2006.

[68] Steve Gregory. *Parallel logic programming in PARLOG: the language and its implementation.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[69] Davide Grossi, Frank Dignum, Mehdi Dastani, and Lambèr Royakkers. Foundations of Organizational Structures in Multiagent Systems. In *Proc. 4th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 690–697. ACM, 2005.

[70] Ramanathan Guha. *Contexts: A formalization and Some Applications.* PhD thesis, Stanford University, 1991.

[71] Ramanathan Guha and Douglas Lenat. Language, Representation And Contexts. *Journal of Information Processing*, 15(3):340–349, 1992.

[72] Naresh Gupta and Dana Nau. On the Complexity of Blocks-World Planning. *Artif. Intell.*, 56(2-3):223–254, 1992.

[73] Karen Henricksen and Jadwiga Indulska. Developing context-aware pervasive computing applications: Models and approach. *Pervasive and Mobile Computing*, 2(1):37–64, 2006.

[74] Anthony Hepple, Louise Dennis, and Michael Fisher. A Common Basis for Agent Organisations in BDI Languages. In *Proc. Languages, Methodologies and Development Tools for Multi-agent Systems*, pages 171–188. Springer, 2008.

[75] Koen Hindriks, Frank de Boer, Wiebe van der Hoek, and John-Jules Ch. Meyer. Agent Programming in 3APL. *Autonomous Agents and Multi-Agent Systems*, 2(4):357–401, 1999.

[76] Benjamin Hirsch. *Programming Rational Agents.* PhD thesis, University of Liverpool, June 2005.

[77] Benjamin Hirsch, Michael Fisher, Chiara Ghidini, and Paolo Busetta. Organising Software in Active Environments. In *Computational Logic in Multi-Agent Systems (CLIMA-V)*, volume 3487 of *Lecture Notes in Computer Science.* Springer-Verlag, 2005.

[78] Benjamin Hirsch, Thomas Konnerth, and Axel Heßler. Merging Agents and Services—the JIAC Agent Platform. In Rafael Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah Seghrouchni, editors, *Multi-Agent Programming: Lanaguages, Platforms and Applications*, pages 159–185. Springer, 2009.

[79] Brayan Horling and Victor Lesser. A Survey of Multi-Agent Organizational Paradigms. Technical report, Univerisy of Massachusetts, May 2005.

[80] Jomi Hübner, Rafael Bordini, and Gauthier Picard. Using *Jason* and *M*oise $^+$ to Develop a Team of Cowboys. In Koen Hindriks, Alexander Pokahr, and Sebastian Sardiña, editors, *ProMAS*, volume 5442 of *Lecture Notes in Computer Science*, pages 238–242. Springer, 2008.

[81] Jomi Fred Hübner, Jaime Simão Sichman, and Olivier Boissier. A Model for the Structural, Functional, and Deontic Specification of Organizations in Multiagent Systems. In *SBIA '02: Proceedings of the 16th Brazilian Symposium on Artificial Intelligence*, pages 118–128, London, UK, 2002. Springer-Verlag.

[82] Darrel Ince. *An Introduction to Discrete Mathematics and Formal System Specification.* Clarendon Press, New York, NY, USA, 1988.

[83] Ole Høgh Jensen and Robin Milner. Bigraphs and mobile processes (revised). Technical Report 570, Computer Laboratory, University of Cambridge, February 2004.

[84] Yonit Kesten, Zohar Manna, and Amir Pnueli. Temporal Verification of Simulation and Refinement. In *A Decade of Concurrency*, volume 803 of *LNCS*, pages 273–346. SV, 1994.

[85] Hiroaki Kitano and Satoshi Tadokoro. RoboCup Rescue: A Grand Challenge for Multiagent and Intelligent Systems. *AI Magazine*, 22(1):39–52, 2001.

[86] Robert Kowalski. The early years of logic programming. *ACM Communications*, 31(1):38–43, 1988.

[87] Victor Lesser. Reflections on the Nature of Multi-Agent Coordination and Its Implications for an Agent Architecture. In *Autonomous Agents and Multi-Agent Systems*, volume 1, pages 89–111. Springer, March 1998.

[88] Hector Levesque, Philip Cohen, and José Nunes. On Acting Together. In *Proceedings of the Eighth National Conference on Artificial Intelligence (AAAI-90)*, pages 94–99, 1990.

[89] Hector Levesque and Maurice Pagnucco. Legolog: Inexpensive experiments in cognitive robotics. In *Proceedings of the 2nd International Cognitive Robotics Workshop*, Berlin, Germany, August 2000.

[90] Hector Levesque, Raymond Reiter, Yves Lespérance, Fangzhen Lin, and Richard Scherl. GOLOG: A Logic Programming Language for Dynamic Domains. *Journal of Logic Programming*, 31:59–84, 1997.

[91] Rodrigo Machado and Rafael Bordini. Running AgentSpeak(L) Agents on SIM_AGENT. In *Intelligent Agents VIII, 8th International Workshop, ATAL 2001 Seattle, WA, USA, August 1-3, 2001, Revised Papers*, volume 2333 of *Lecture Notes in Computer Science*. Springer, 2002.

[92] John McCarthy. Notes on Formalizing Context. In *IJCAI'93: Proceedings of the 13th international joint conference on Artifical intelligence*, pages 555–560, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

[93] Robin Milner. Bigraphical Reactive Systems. In *Proc. 12th International Conference on Concurrency Theory (CONCUR)*, volume 2154 of *Lecture Notes in Computer Science*, pages 16–35. Springer, 2001.

[94] Robin Milner. Pure bigraphs: Structure and dynamics. *Inf. Comput.*, 204(1):60–122, 2006.

[95] Robin Milner. Ubiquitous Computing: Shall we Understand It? *Computer Journal*, 49(4):383–389, 2006.

[96] Kris Nagel, Cory Kidd, Thomas I'Connell, Anind Dey, and Gregory Abowd. The Family Intercom: Developing a Context-Aware Audio Communication System. In *Proceedings of Ubicomp 2001*, pages 176–183, 2001.

[97] Daniela Nicklas and Bernhard Mitschang. The NEXUS Augmented World Model: An Extensible Approach for Mobile, Spatially Aware Applications. In *OOIS*, pages 392–404, 2001.

[98] Pablo Noriega, Javier Vázquez-Salceda, Guido Boella, Olivier Boissier, Virginia Dignum, Nicoletta Fornara, and Eric Matson, editors. *Coordination, Organization, Institutions and Norms in agent systems (COIN) II*, volume 4386 of *Lecture Notes in Computer Science*. Springer-Verlag, 2007.

[99] Timothy Norman, Alun Preece, Stuart Chalmers, Nicholas Jennings, Michael Luck, Viet Dang, Thuc Nguyen, Vikas Deora, Jianhua Shao, Alex Gray, and Nick Fiddian. Conoise: Agent-based Formation of Virtual Organisations. In *Proceedings of the 23rd SGAI International Conference on Innovative Techniques and Applications of Artificial Intelligence*, pages 353–366. Springer-Verlag, 2003.

[100] Lin Padgham and Michael Winikoff. *Developing Intelligent Agent Systems: A Practical Guide*. John Wiley and Sons, 2004.

[101] Barney Pell, Douglas Bernard, Steve Chien, Erann Gat, Nicola Muscettola, P. Pandurang Nayak, Michael Wagner, and Brian Williams. An Autonomous Spacecraft Agent Prototype. In *Autonomous Robots*, pages 253–261. ACM Press, 1997.

[102] Amir Pnueli. The Temporal Logic of Programs. In *18th Annual Symposium on Foundations of Computer Science*, pages 46–57, 1977.

[103] Alexander Pokahr, Lars Braubach, and Winfried Lamersdorf. Jadex: A BDI Reasoning Engine. In Bordini et al. [15], pages 149–174.

[104] Enrico Pontelli. Adventures in Parallel Logic Programming. `http://www.cs.nmsu.edu/~epontell/advent.html`, June 1996.

[105] Michael Prietula, Kathleen Carley, and Les Gasser, editors. *Simulating Organizations: Computational Models of Institutions and Groups.* MIT Press, 1998.

[106] David Pynadath and Milind Tambe. Team Coordination among Distributed Agents: Analyzing Key Teamwork Theories and Models. In *In Proceedings of the AAAI Spring Symposium on Intelligent Distributed and Embedded Systems*, 2002.

[107] David Pynadath and Milind Tambe. The Communicative Multiagent Team Decision Problem: Analyzing Teamwork Theories and Models. *Journal of Artificial Intelligence Research (JAIR)*, 16:389–423, 2002.

[108] David Pynadath, Milind Tambe, Nicolas Chauvat, and Lawrence Cavedon. Toward Team-Oriented Programming. In *6th International Workshop on Intelligent Agents VI, Agent Theories, Architectures, and Languages (ATAL)*, volume 1757 of *Lecture Notes In Computer Science*, pages 233–247. Springer-Verlag, 1999.

[109] Anand Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. *Lecture Notes in Computer Science*, 1038, 1996.

[110] Anand Rao and Michael Georgeff. Modeling Rational Agents within a BDI-Architecture. In Richard Fikes and Eric Sandewall, editors, *International Conference on Principles of Knowledge Representation and Reasoning (KR)*, Cambridge, Massachusetts, April 1991. Morgan Kaufmann.

[111] Anand Rao and Michael Georgeff. An Abstract Architecture for Rational Agents. In Charles Rich, William Swartout, and Bernhard Nebel, editors, *Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning (KR'92)*, pages 439–449. Morgan Kaufman, 1992.

[112] Anand Rao and Michael Georgeff. BDI Agents: From Theory to Practice. In *Proceedings of the 1st International Conference on Multi-Agent Systems (ICMAS)*, pages 312–319, Washington, DC, USA, 1995. IEEE Press.

[113] Anand Rao and Michael Georgeff. Decision Procedures for BDI Logics. *Journal of Logic and Computation*, 8(3):293–343, 1998.

[114] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall Press, Upper Saddle River, NJ, USA, 2009.

[115] Bill Schilit, Norman Adams, and Roy Want. Context-Aware Computing Applications. In *WMCSA '94: Proceedings of the 1994 Workshop on Mobile Computing Systems and Applications*, pages 85–90, Washington, DC, USA, 1994. IEEE Computer Society.

[116] John Searle. *Speech Acts: An Essay in the Philosophy of Language.* Cambridge University Press, Cambridge, UK, 1969.

[117] Ehud Shapiro. Concurrent Prolog: A Progress Report. In *Fundamentals of Artificial Intelligence: An Advanced Course, held in Vignieu, France, July 1985*, pages 277–313, London, UK, 1986. Springer-Verlag.

[118] Ira Smith and Philip Cohen. Toward a Semantics for an Agent Communications Language Based on Speech-Acts. In *Proc. American National Conference on Artificial Intelligence (AAAI)*, pages 24–31, 1996.

[119] Robert Stalnaker. Pragmatics. *Synthese*, 22:272–289, 1970.

[120] Robert Stalnaker. *Context and Content.* Oxford University Press, 1999.

[121] Roy Sterritt and Michael Hinchey. Radical Concepts for Self-Managing Ubiquitous and Pervasive Computing Environments. 3825, 2006 2006.

[122] Thomas Strang and Claudia Popien. A Context Modeling Survey. In *Workshop on Advanced Context Modelling, Reasoning and Management, UbiComp 2004 - The Sixth International Conference on Ubiquitous Computing*, September 2004.

[123] Milind Tambe. Teamwork in Real-world Dynamic Environments. In *Proceedings of the 1st International Conference on Multi–Agent Systems.* MIT Press, 1995.

[124] Gil Tidhar. Team-Oriented Programming: Preliminary Report. Technical Report 1993-41, Australian Artificial Intelligence Institute, April 1993.

[125] Kagan Tumer and Adrian Agogino. Distributed Agent-based Air Traffic Flow Management. In *AAMAS '07: Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, pages 1–8, New York, NY, USA, 2007. ACM.

[126] Javier Vázquez-Salceda, Virginia Dignum, and Frank Dignum. Organizing Multiagent Systems. *Journal of Autonomous Agents and Multi-Agent Systems*, 11(3):307–360, 2005.

[127] Roy Want, Bill Schilit, Norman Adams, Rich Gold, Karin Petersen, David Goldberg, John Ellis, and Mark Weiser. *The Parctab Ubiquitous Computing Experiment*, pages 45–101. Springer US, 1996.

[128] Mark Weiser. Some Computer Science Issues in Ubiquitous Computing. *Commun. ACM*, 36(7):74–84, July 1993.

[129] Mark Weiser. The World is not a Desktop. *Interactions*, 1(1):7–8, January 1994.

[130] Mark Weiser. The Computer for the 21st Century. *SIGMOBILE Mobile Computing and Communications Review*, 3(3):3–11, July 1999.

[131] Matt Welsh, Tarek Abdelzaher, and others, editors. *ACM Transactions on Sensor Networks (TOSN)*, volume 7. ACM, New York, NY, USA.

[132] Michael Winikoff. An AgentSpeak meta-interpreter and its applications. In *In Proceedings of the 3rd international Workshop on Programming Multi-Agent Systems*, pages 123–138. Springer, 2005.

[133] Michael Winikoff. JACK Intelligent Agents: An Industrial Strength Platform. In Bordini et al. [15], pages 175–193.

[134] Michael Wooldridge. *Introduction to MultiAgent Systems.* John Wiley & Sons, June 2002.

[135] Michael Wooldridge and Nicholas Jennings. Intelligent Agents: Theory and Practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.

[136] Michael Wooldridge, Nicholas Jennings, and David Kinny. The Gaia Methodology for Agent-Oriented Analysis and Design. *Autonomous Agents and Multi-Agent Systems*, 3:285–312, 2000.

[137] Rong Yang. *P-PROLOG: A Parallel Logic Programming Language.* World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1987.

Index