MAGE: Multi-Agent Game Environment

by
Visara Urovi

Thesis submitted to the University of London for the degree of Doctor of Philosophy

Department of Computer Science,
Royal Holloway, University of London,
Egham, Surrey TW20 0EX,
United Kingdom

March 3, 2011

Declaration

I declare that this dissertation was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

Visara Urovi

Visara Urovi

Supervisor and Examiners

Supervisor:	*Dr. Kostas Stathis*
Internal Examiner:	*Dr. Sanjay Modgil*
External Examiner:	*Dr. Jeremy Pitt*

Abstract

We study the use of games as a metaphor for building social interaction in norm-governed multi-agent systems. As part of our research we propose MAGE (Multi-Agent Game Environment) as a logic-based framework that represents complex agent interactions as games. MAGE seeks to (a) reuse existing computational techniques for defining event-based normative system and (b) complement these techniques with a coordination component to support complex interactions.

A game in MAGE is defined by a state, a set of normative rules describing the valid moves at different states and a set of effect rules describing how the state evolves as a result of a move taking place. Given a specification of the normative rules, in the implementation of a game, we use game containers as components that mediate the moves of players by checking their compliance with the rules of the game and by maintaining the state of the game.

The reuse part of MAGE relates physical actions that happen in an agent environment to valid moves of a game representing the social environment of an application. MAGE allows to model complex interactions from simpler atomic sub-games. In this context, we investigate how coordination patterns can be used to dynamically play more than one game in parallel, change the status of games or choose amongst games. For this purpose, we examine how to define compound games from atomic sub-games. Compound games are build by describing the conditions and the patterns that activate a sub-game and the coordination mechanisms of MAGE ensure that sub-games are activated according to how interactions are specified to evolve at run-time.

To illustrate the MAGE approach, we discuss how to use the framework to specify the social interaction in two different scenarios: (i) Open-Packet-World - a simple simulation where agents compete to collect and deliver packets in a grid and (ii) an earth-observation application - where agents represent services, both for clients and providers, and negotiate the provision of these services by combining argumentation and communication protocols.

We also use the Open-Packet-World scenario to evaluate the effectiveness of the framework. We show that we can effectively support at run-time a large-scale multi-agent systems regulated by norms. We conclude the dissertation by summarising our contributions and identifying areas for future work.

Acknowledgements

This work has been possible due to a number of people who supported me as follows:

- I would like to express my gratitude to my supervisor, Dr. Kostas Stathis, who has supported me throughout my thesis with his patience and knowledge of the field. I attribute the level of my PhD thesis to his encouragement and effort in completing my thesis.
- I would like to thank my advisor, Dr. Hugh Shanahan, who has been very supporting and always making sure that I was being looked after. I would also like to thank my MSc professor Andrea Omicini for introducing me to Artificial Intelligence and helping me to become a better student.
- I had the chance to work in the ARGUGRID European project which partially funded my studies as well as my conference presentations. I would like to thank the ARGUGRID team for the smooth running of the project and the discussions raised during the project meetings.
- The Department of Computer Science at Royal Holloway has provided the support and equipment, including a fee waver scholarship for my studies. I would also like to thank the Institute of Applied Science of Switzerland, in particular Michael Ignaz Schumacher for hosting me during my writing up period.
- In my daily work at Royal Holloway I have been blessed with a friendly and cheerful group of office mates and friends. Pedro Contreras has helped me with his good arguments about MAS theory as well as with his friendship and good long talks from which I always recovered some

of my positive thinking. Jarred McGinnis for our collaborations prior to my PhD thesis which resulted in a better understanding of agent protocols but also for his friendship and support. Laurence O'Toole who entertained with his repertoire of mathematical quizzes and stories about Irish culture. Eun Jung Kim for being such a strong minded and inspirational woman. Leticia Mora for sharing some of her organisational skills with me, it was quite a revelation at the time and Yangyue Feng for being such an affectionate friend.

- Special thanks go to my husband, Stefano Bromuri. In this occasion we have been working together also professionally. Our confrontations and arguments have been an inspirational exchange of ideas. He has contributed in my thesis in many ways, first by developing a multi-agent platform which I then extended as part of my PhD thesis and secondly with valuable support when I was writing up the thesis. From a personal perspective, his love and support has been essential to me to finish this work.

- Finally, I would like to thank my parents for supporting me morally and financially throughout all my studies at University, for encouraging me to be better on what I did and at the same time being so proud in any achievement I had.

Contents

List of Figures

Chapter 1

Introduction

With a wider use of global networks, such as the Internet, a large number of computing applications are becoming increasingly more open, heterogeneous and distributed. Openness in such systems means that there are no restrictions for the components on when to join or leave the network [39] thus, these systems need to be able to operate in very dynamic settings. Heterogeneity means that the components of these systems are not designed in the same way and with the same functionalities therefore, the system should enable interactions amongst components and yet prevent that design choices on a single component reflect on the other components of the system. Distribution means that the components are distributed over a network bringing challenges such as the consistency of the data maintained in the system, synchronisation and communication between the distributed components.

One way to deal with open, heterogeneous and distributed systems is to model these systems as Multi-Agent Systems (MAS). A MAS is designed and implemented as several interacting components called agents. An agent is situated in an agent environment where it is capable of flexible autonomous action in order to meet its design objectives [70]. In general, agents can reason autonomously upon a problem and use the interaction capabilities to take actions in the system. The autonomy and the cognitive abilities of an agent makes it act depending on its internal reasoning and decision making mechanism. Because of their autonomy, agents cannot be assumed to be acting for the best interest of the system as a whole. Therefore the need to incentivise these agents to act according to some "standard" behaviour (i.e. the behaviour that the designers have in mind) arises. This is why when we develop a MAS, having a model of interaction is crucial. On one hand we need to preserve the agent's autonomy and on the other hand we want to constrain interactions to be in line with the system's purposes.

In MAS applications agents need to collaborate with other agents in order to achieve common tasks. For this reason, they are often described as social entities [51]. The agents of a MAS will typically have incomplete information or capabilities to solve a problem. Their ability to have social interactions becomes essential when we require them to work together to solve problems that are beyond the individual capabilities or knowledge of a single agent [44].

Moreover, very often in such MAS applications, software agents are developed by different parties and deployed within a specific domain to achieve given objectives [99]. In this case a MAS is said to be an *open* MAS. An important characteristic other than agents being developed by different parties is that agents can be non-cooperative [10]. *Openness* also often implies that it is not possible to assume to have access to the internal reasoning of the agents without severely undermining their autonomy [6].

Early work in MAS has focused on the representation of interactions in terms of communication protocols that agents can use to interact with each other [29, 28, 14, 27]. As these protocols standardise the way in which agents partake in social activities, more recent work puts the emphasis on normative concepts such as obligation, permission, and prohibition, amongst other, as a more elaborated specification for representing agent protocols or their interactions in general (see [13, 94]).

Although openness of this kind may encourage many agents to participate in an application, interactions in the system must be regulated so that the overall specification of the application domain is respected.

Norm-governed multi-agent systems [13] are open multi-agent systems that are regulated according to the normative relations that may exist between member agents, such as permission, obligation, and institutional power [72], including sanctioning mechanisms dealing with violations of prohibitions and non-compliance. As a result, there are multiple challenges on how to design norm-governed multi-agent systems and how to engineer these systems to systematically support the vision of artificial agent societies [99] that will arise from their deployment.

1.1 Motivation

The problem we are trying to address is how to design, specify and implement complex interactions using computational logic to support classes of applications such as the formation of Virtual Organisations in a service grid [56]. In an attempt to address this problem we found that there is a lack of off-the-shelf, generic tools to develop open multi-agent systems in general and normative multi-agent systems in particular. Many frameworks have been proposed as knowledge representation frameworks for norm-governed systems: such as the use of semantic concepts and ontologies to represent norms [121, 58], temporal logic representation of normative concepts [33, 54] and representation of coordination mechanisms [85] for distributed norm-governed systems, are a few of the many works in the area. Despite the proliferation of these frameworks, there is relatively less work on how to develop systematically more complex activities that require agents to coordinate their interactions in an agent environment. There is, in other words, the need for computational frameworks that support the composition of complex interactions and allow for their coordination.

Coordination between interacting agents is a very important aspect of social interaction. In agent based systems, we often use protocols as a way to define which actions are to be exchanged during the interactions and possibly with what order. However, since agents are autonomous, they may decide to take actions that do not follow the system's specification. Thus, when modeling interactions, the coordination mechanism provides the means to anticipate possible interactions in advance and design the system to avoid undesirable interactions and to exploit desired ones [133]. Coordination mechanisms such as workflow patterns [128] are important to ensure that the agents interact by following an interaction flow.

Many existing approaches, such as [13, 7, 98], use a logic-based approach that focuses on the definitions of normative relations between interacting agents. The focus of these approaches is in the specification of normative relations rather than in defining a systematic development of the interactions in a distributed and open setting of the agent environment. Other work, such as [48, 79, 82], focuses on combining protocols and specifying their coordination. However, these works are often focused on the expressive power of the formalism proposed and typically abstract away from evaluating computational aspects based on experimentation. As a result the computational behaviour of many representation frameworks for norm-governed multi-agent systems are often studied in isolation, at times theoretically only, and in many occasions their experimental evaluation is left unexplored.

Our specific motivation is to extend the GOLEM platform with a methodology for developing agents interactions. GOLEM [21] is a multi-agent environment platform that allows the development of agent application in terms of distributed cognitive agents and objects that are the resources agents interact with [73, 103]. Interaction in GOLEM is tackled from the physical perspective by describing how agents can perform physical actions in a distributed agent environment. However, because its lack of the necessary social concepts, GOLEM is not intended to model agent interactions that allow for cooperation and coordination of agents.

The lack of concepts that deal with the social aspects of an agent environment were more evident when we applied GOLEM to model applications based on the notion of VOs [83], as in the ARGUGRID European project. ARGUGRID [3] was a research project that aimed at providing a new model for programming a service GRID at a semantic, knowledge-based level of abstraction through the use of argumentative agent technology. Agents in this context act on behalf of (a) users who specify abstract service requests or (b) providers who offer electronic services on the GRID. User requests result in agents interacting with other agents by forming dynamic VOs in order to enable the transformation of abstract user requests to concrete services that the GRID can support. To guarantee that interactions in VOs are regulated, agent-oriented provision of services must conform to service level agreements, while agent interaction more generally must be governed by electronic contracts. A requirement of ARGUGRID is that agreements and contracts need to be negotiated on the fly by agents. In other words, there is a need to support protocols and workflows that enable the activities of VO creation, operation, and dissolution. One of the issues then becomes how to represent these complex activities at a knowledge-based level to be suitable for argumentation-based agents that use the framework to be coordinated during their interactions.

1.2 Approach, Aims and Objectives

We are interested in the process of social interaction between agents to support practical applications. We understand interaction as actions performed between agents situated in a physical environment and which are mediated by a social environment. The social environment defines mediation mechanisms that provide a structured interaction model based on social rules. In this context we view agent interactions as *social games* played within an agent environment. A *social game* is a mechanism in an agent environment which contains the social rules of the agent environment, has its own state, and mediates the interaction of agents. We propose a Multi-Agent Game Environment (MAGE) framework to bridge the gap between complex models of normative systems and the dynamics of interactions in an agent environment.

In MAGE, we are interested in a systematic representation of agent interactions. Protocols are a well known approach to interaction, however interaction mechanisms sometimes require many protocols to be combined in a flexible manner to form more complex protocols in a systematic way. In MAGE, in order to be able to combine protocols into complex ones, coordination of *social games* becomes very important as a mechanism to enable interactions amongst agents.

The aims of our work are as follows.

- Introduce a general game-based model for modeling the concept of social agent environment.
- Separate physical from social interactions in a MAS application. In this context, an open multi-agent system will be seen as two concurrent but interconnected composite structures that evolve over time: one providing a physical environment for action execution while the other acting as a social environment that, seen as a social game, evaluates norms of the multi-agent system.
- Provide a framework to compute, at run-time, the norms applying to an agent such as permissions, prohibitions, and sanctions. We assume that agents cannot compute these normative relations on their own because of computational constraints, and incomplete knowledge about the application state.
- Identify mechanisms for coordinating complex interactions as compound social games.

Given this set of general aims, we want to achieve them by means of the following specific objectives.

- We seek to provide a social environment which mediates and enables social interactions in a MAS by means of a game-based specification that builds on top of Event Calculus [77, 74] to regulate how the state of the social interactions evolves.

- Extend the GOLEM [21] platform by implementing a game-based component that supports the social aspects of an agent's interaction. In particular we develop an infrastructure for the coordination and the enactment of social interactions on top of a physical environment.

- Show how to distribute the social environment and map it to a distributed physical environment by combining the Ambient Event Calculus [22] with a game-based framework.

- Develop a methodology showing how to instantiate the framework for developing practical applications.

- Evaluate the proposed framework on two scenarios: (i) the open packet world scenario [124, 125] to explain the main functionalities of the framework and (ii) and a more complex scenario called the Earth Observation scenario [120] which is drawn from the ARGUGRID project and involves negotiation and composition of services.

1.3 Contribution, Scope and Significance

We identify the following contributions of the work presented in this thesis.

- Propose a knowledge representation framework for engineering agent interactions in social environments that allows us to distribute physical and social environments of a norm-governed application.

- Introduce the notion of social containers to capture portions of the social environments engineered as social games.

- Present the idea of social games in terms of existing work developed in [115, 113] but by separating the specifications of legal moves from the physical environment, both conceptually and during implementation.
- Integrate normative notions in the specification of valid moves, similarly to the work presented in [13] but with the compositional part of [115, 113], thus ,making the resulting framework applicable to coordinating complex agents interactions.
- Use the representational framework to develop a set of coordination primitives that allow us to specify complex interactions, such as basic workflows, to be reused in practical applications.
- Implement the framework in MAGE as an extension of GOLEM to support the concept of social environment.
- Illustrate the functionalities through a case study.
- Evaluate the system's efficiency to compute the physical and the social state through experimentation.

1.4 Overview of the Thesis

The reminder of the thesis is organised as follows:

In Chapter 2 we present the background concepts upon which we base our ideas. First we discuss the notion of social environment and we describe what it requires if this environment is to be treated as a first class abstraction. We then identify the components of a social environment and report on the existing work that has been carried out in this area. We classify the literature in terms of the kinds of social environment that arise, including groups, institutions, virtual organisations and open agent societies. We then report on existing work that has been done in trying to: (i) implement agent environments and; (ii) interpret these kind of environments using an event-based approach to capture the interactions, which is the approach that we inherit from GOLEM, and which we want to extend with normative concepts.

In Chapter 3 we present the GOLEM platform. We explain the three main building blocks of the GOLEM platform: objects, agents, and containers. Objects are reactive entities used to wrap resources of the external environment. Agents are cognitive and active entities that manipulate resources and interact with other agents. Containers represent a portion of the distributed agent environment containing agents and objects. We also summarise how the agent environment in GOLEM evolves in time and how agents can query the environment to perceive properties of entities in it. We conclude this Chapter by arguing that despite the modularity and flexibility that GOLEM provides to model agent environments, it does not provide the mechanisms for defining a social environment.

In Chapter 4 we present our version of the game metaphor [115] to model atomic interactions between agents in a multi-agent system. Under this metaphor, agent interactions count as moves of a social game governing the interaction between the participating agents that enact the roles of different kind of players. The outcomes of these games will not necessarily give rise to a winner or looser, but often to win-win situations where a number of agents achieve their goals. We illustrate how social games can be seen as social containers which maintain the social state of the interactions amongst agents/players. We illustrate our framework by using the Open Packet World [124, 125], a scenario where agents interact and compete with one another to collect packets and bring them into destination points. Chapter 4 implements the following contribution of the thesis: it captures the building blocks of the social environment in terms of social containers; it presents the idea of social games following the work presented in [115, 113] but by separating the specifications of legal moves from the physical environment; it integrates normative notions in the specification of valid moves and it presents partially our proposed framework for engineering agent interactions for norm governed applications, further expanded in Chapter 5.

In Chapter 5 we study how to build more complex interactions as complex social games. We show that it is possible to distribute the social environment so that the evaluation of the social rules reflects the needs of an application. In this context we show how the social rules will be enforced. We also argue that enforcing the rules depends on how the designer of the social environment organises objects and the roles of agents for a specific application. Chapter 5 implements the following contribution of the thesis: it proposes a framework for engineering agent interactions for norm governed applications by extending the model presented in Chapter 4 and it proposes a set of coordination primitives that allow us to specify complex interactions.

In Chapter 6 we illustrate the implementation of the main functionalities of the social environment. We first explain the GOLEM architecture and then we show how we extend it with the MAGE architecture. We implement a propagation mechanism so that agent's actions are propagated to the social environment and a coordination mechanism that evaluates if such actions comply with the social rules defined in the social environment. We also explain how we implement games and how agents can observe their state. In Chapter 6 we implement the MAGE framework as an extension of GOLEM to support the concept of social environment which is a main contribution of the thesis.

In Chapter 7 we illustrate how we apply our framework in the ARGUGRID project [3, 120] to propose a new model for building GRID-based applications by supporting the formation of dynamic VOs. To demonstrate how MAGE contributes to ARGUGRID, we use the Earth Observation scenario [119]. In such scenario agents in the environment negotiate with other agents of the GRID to form a VO. We show how the interaction of agents is modeled to allow them to form VOs. One of the contributions of the thesis is to illustrate the MAGE platform using the Earth Observation Scenario whose specification is presented in this chapter.

In Chapter 8 we propose a methodology for defining MAGE-based applications and we evaluate the performances of the framework using the Open Packet World scenario. Using this scenario, we define a set of tests where we distribute the agent environment and evaluate how we can improve the performances to support larger-scale, normative multi-agent systems. We also compare the aspects of the game-based model proposed in MAGE with the GOLEM agent framework. Chapter 8 implements the following contribution of the thesis: it evaluates the system's efficiency to compute the physical and the social state of the agent environment and it develops a methodology to show how MAGE is used to design practical applications.

Finally, in Chapter 9 we conclude with a summary of our research where we highlight the advantages and discuss possible weak points of our approach. We conclude with future work where we discuss the future extensions of MAGE and its implementation.

1.5 Publications

The work and the ideas presented in this thesis have been partially presented to scientific workshops, conferences and journals related to MAS research.

Early work in game-based interaction was presented in [81] where we studied the idea of deliberative games to support agent argumentation and in [122, 123] where we further extend these ideas with a Multi-Agent platform for the automation of the workflow selection and execution.

In [23] we presented a prototype Multi-Agent system whose goal was to support a 3D application for e-retailing. We argued that these types of applications can be engineered in a flexible and reliable way by using the notion of agent environment. We also extended the agent environment abstraction to support the deployment and discovery of e-retailing services visualised by a 3D virtual environment interface. We further extended this work in [25] by introducing a game-based approach to support social interactions among agents and show how to combine them with semantic-web technologies to develop an e-retailing application.

In [126] we presented MAGE as a game-based logical framework that uses games as a metaphor for representing complex agent activities within an artificial society. The framework supports the construction of social games as composite games built from component sub-games. Social games use a normative interpretation of valid moves and a set of coordination patterns to build complex activities from simpler sub-activities. An application for such framework is presented in [24] where we use the framework to support service discovery, selection and negotiation in a multi-agent system.

In [26] we presented iCampus as a prototype multi-agent system whose goal is to provide the ambient intelligence required to connect people in a University campus and make that campus inclusive and accessible. The work outlines how to specify iCampus in the Ambient Event Calculus and implemented it using the agent environment GOLEM to deploy guide agents over a campus network.

In [125] we presented a knowledge representation framework with an associated run-time support infrastructure that is able to compute, for the benefit of the members within a norm-governed multi-agent system, physically possible and/or permitted actions current at each time, as well as sanctions. This line of work forms the basis of the MAGE evaluation which will be discussed later in this thesis. More precisely, in [124] we show our experimental results on a benchmark scenario and show how by distributing norms we can provide run-time towards supporting to large-scale, norm-governed, multi-agent systems.

Chapter 2

Interaction in Social Environments

In this chapter we present the state of the art of social interaction in MAS, from the perspective of social agent environment as a first-class abstraction mechanism. Although there is a lot of work that relates to the formulation of social environments, our emphasis is on comparing with existing implemented frameworks and in reviewing approaches that are *event centric*.

This chapter is organised as follows: In Section 2.1 we discuss the notion of social environment. In Section 2.2 we define the components of a social environment. An overview of the selected literature on social environments is presented in Section 2.3. More specifically, this section discusses the concept of agents organising in groups, the concept of institutions as a way to represent normative aspects of social environments, virtual organisations as a concept used for representing dynamic relationships and agreements in the social environment, and the concept of agent societies as a way to organise the social environment. Section 2.4 describes how existing implemented frameworks design the notion of social environment into these frameworks. Section 2.5 identifies the main research works that are based on events and discusses their main characteristics. In this section we also identify the key aspects that we provide in our approach compared to these works. Finally, Section 2.6 summarises and concludes this chapter.

2.1 The Notion of Social Environment

A system constructed with two or more agents is called a *Multi Agent System* (MAS) if it comprises the following elements [51]: an environment containing the agents and objects, a set of operations used to sense, perceive the environment or to manipulate objects and the relationships that link agents to each other. As we argued in the previous Chapter, agents in a MAS typically have incomplete information or capabilities to solve a problem in the environment. For this reason, agents need to work together to solve problems that are beyond the individual capabilities or knowledge of a single agent [44]. The need for *cooperation* amongst agents comes with many requirements for the agent environment which should enable agent interaction, communication in a common language and possibly coordination.

In general, we want to distinguish the concept of environment into physical environment and social environment. The physical environment being the entity that defines the principles and processes that govern and support a population of entities [91] and the social environment being the entity that defines a communication environment in which agents interact in a coordinated manner [91]. Here we assume that agents are already situated in a *physical* environment that provides a model for the physical interactions based on physical rules and we focus on the model of *social* environment as the entity which models social aspects of the agent interactions.

Following Odell's et al definition of social environment, we define a *social environment* as the mediation mechanism providing the social rules and the coordination mechanisms that apply in *"time"* and *"space"* to actions performed by agents that are situated in a physical environment.

The main focus when designing social environments is the specification of the social rules. The social rules specify what it is socially agreed (or not) to do, in terms of stipulated or generally accepted standards or criteria, when agents interact in the environment. These rules are important because they are used to mediate agent interactions. The type of mediation we are focusing on is a model for the social environment that, given the dynamic communicative acts exchanged amongst interacting entities, can determine if there exists a social meaning for these acts, and, whenever necessary, it triggers a set of consequences which are defined as part of the mediation mechanism.

From a software engineering perspective, in order to have a platform for defining a social environment we need to identify the building blocks [1] from which we can build social environments. In [19], Bromuri suggests the need to consider the concept of agent environment as a first-class abstraction. We extend this view to social environments as a way to separate concerns of the social environment. Separation of concerns [40] refers to the ability to identify a distinctive piece of software that separates a computer program into independent features that can be manipulated as first-class entities, and integrated with a composition mechanism. Sunkle, in [117], defines a first class abstraction exhibiting the following characteristics:

- **Instantiation:** First-class abstractions can be instantiated at compile-time or run-time and possibly other stages of program execution.
- **Storing:** First-class abstractions can be stored in variables and data structures.
- **Manipulation:** First-class abstractions allow for compile-time or run-time structural manipulation.
- **Typing:** First-class abstractions can be passed as parameters to other program elements such as methods or returned from methods.
- **Composition:** First-class abstractions can be composed in various expressions and statements within a programming language.

We will interpret the social agent environment as a first-class abstraction by identifying a main entity from which we can build a social environment that contains the five features proposed by Sunkle [117]. We describe next the components of a social environment.

2.2 Components of Social Environments

Various approaches have been proposed to represent social interactions in a MAS, all having in common that they define a rule-based medium to mediate the interactions. We identify such a rule-based medium as the social environment. The rules are used as a way to specify a structure in the interactions between members of such environment. Conceptually, we can identify four main components for modeling social environments:

- **Member agents** which interact with each other and generate actions in the environment. Member agents may also access and change the state of the environment.
- **Physical Territory** which refers to the physical space in which the member agents are situated.
- **Social rules** which are used to govern the agent's interactions in the physical territory.
- **Social State** which maintains the state and the evolution of the social environment as a result of the actions performed by member agents in the physical territory.

.

2.2.1 Member Agents

Odell et al, in [91], define the social environment as an entity aimed to provide the necessary mechanisms so that agents can collaborate or interact in a system in a meaningful way. Thus, agents are the main component that influences the design of agent environments (social and physical). Agents can be designed with different goals and capabilities. In designing an agent environment, it is not important the reasoning process and capabilities of single agents, but it is crucial to define how agents are enabled to act in the environment.

The JAVA Agent Development Environment (JADE) [69] is a widely known agent platform that uses the concept of directory facilitator as the directory where agents register so agents can find each other. Although, JADE does not address issues of social environment, the concept of directory facilitator defines the membership of an agent in the agent environment. Here, we make a clear distinction between membership in the physical environment and membership in the social environment. These two memberships are quite different. This is due to the nature of the two environments. While both physical and social environments occupy a space, the shape of physical space is a designated area (or volume) while the social space can be expressed in terms of the degree of interaction [91]. Thus, the membership of an agent in the physical environment is determined by its situation in a physical space that the agent occupies, while, the membership in a social environment is a dynamic relationship caused as a result of interactions that agents have in the social environment.

The membership of the agents in the physical environment is determined by their physical location. The same agents, are members of the social environment from the moment they interact in the environment. Their acts are mediated first at a physical level where they must be physically possible to perform an then at a social level where they are attributed a social meaning. Thus, the member agents of a social environment are also members of a physical one. Their membership in a physical environment is essential in order to be members of the social environment.

2.2.2 Physical Territory

The physical territory refers to the portion of the physical environment containing the members of the social environment. This portion has its own physical state whose properties change following a set of physical rules that apply as a result of agent actions. As we stated in the previous Section, a social environment requires a physical one. They both regulate the activities of agents, but they regulate them with different rules and at a different level. At the physical environment level, many actions can be defined as physically possible, however, at a social level many of these actions might be considered as invalid ones because they may not be in line with the social rules defined in the environment. In other words the purpose of the social environment is to regulate the many events that are possible to happen in the physical territory. Although the social environment conceptually regulates the physical territory, it is considered a good practice to separate the two, see for example Artikis et al [13, 12, 11].

2.2.3 Social Rules

A basic form of social rules results from defining agent protocols. Pitt et al in [97] define a *protocol* as the definition of structured messages exchanged between agents which constitute a declarative agreement for communicative behaviour. In the literature we can find protocols being used to interact in a particular situation. For example, there are negotiation protocols that agents use when they need to negotiate resources (see [43, 12]), or deliberation protocols that are used when a group of agents need to deliberate a common decision (see [80, 81]), security protocols used when the need to exchange confidential information [17] and many others.

Protocols have an initiator and a number of participants that are involved in the communication, a state and rules to determine how messages should be exchanged and when the protocol terminates. Several standard protocols for agents communication were proposed by FIPA [2], however, despite the existing standard protocols, new interaction protocols for specific types of interaction (i.e negotiation protocols or argumentation protocols) continue to be proposed. FIPA protocols are not widely used because of their point to point nature which makes them limited to only two interacting parties. Another reason is that their semantics assumes that agents are able to share their mental states, see [100].

Protocols can be expressed in terms of *normative* aspects such as prohibition, permission, obligation and empowerment. In the *normative* interpretation of protocols, the rules of the protocol use normative concepts to define the state or point in time when an act can occur (i.e. [101, 12, 109, 134]). The normative concepts are used as it is possible to capture more fine grained relationships between actions that agents perform in the environment. In such protocols agents take a role within a protocol so that they can perform actions using the normative relations that exist within a role.

The interaction of agents in the social environment can have different outcomes depending on the goals that every participant has. When agents interact using a particular protocol, the outcome may be different depending on the reasoning process of the agents. If we formalise the rules of a protocol and try to build a system that follows the rules of this protocol, we might realise that we need to add some flexibility in the system by extending the protocol. Extending a protocol means that we adapt it by adding new rules so that interactions amongst agents can progress. Sometimes, it is desirable to introduce new actions in a protocol so that agents can go beyond the rules of a single protocol. McGinnis et al, in [38] define expansion mechanisms for protocols where extra rules are added to allow agents to perform additional actions. In a similar way the authors can restrict the actions that agents can perform. In the same line of work, Stathis [112], defines how to extend and filter protocols is by adding extra rules to deal with the expansion and the filtering of the protocol (see [112] for more details). These new modified protocols can be seen as complex protocols.

Applications that require complex interactions benefit from allowing agents to combine protocols during social interactions. In order to enable such protocols, coordination mechanism needs to be put in place so that mediation is offered at the social environment level. In defining a *coordination mechanism* we want to be able to parallelise, choose and synchronise protocols. To capture these control-flow aspects, coordination mechanism should be defined as part of the framework to provide more complex interactions than a single protocol. A way to synchronise, choose and parallelise processes is given by workflow coordination patterns. In general, the term *workflow* refers to the specification of a work procedure or a business process in a set of *atomic activities* and relations between them in order to coordinate the *participants* and the activities they need to perform [4]. The link with the definition of the workflow here is that the *participants* are the agents and the *atomic activities* are protocols.

2.2.4 Social State

In describing interactions in a MAS we often need to distinguish the physical effects that an action has from the social effects of the act. For example, in an auction house an agent raising its hand has as a consequence that the hand is moving in an upward position. The consequences of the raising a hand act can be stored in the physical state of the agent environment.

The social state, on the other hand, evolves according to a set of social rules. Acts performed by agents in the environment can create social facts which may change the social state. Guerin and Pitt, in [66] define social facts as facts created by interactions and the rules governing their creation. In the previous example, the same act of the agent raising its hand in an auction house has a social meaning which modifies also the social state. In this case the act means that the agent is bidding in the auction (A more theoretical treatment of the meaning of actions and the ”count as” relationship at a social level has been discussed in [108]). Guerin and Pitt [66] believe that the history of the agent’s acts, the domain in which the interactions took place and the previous social facts holding in the system affect the meaning of the acts. All these aspects are included in the social state which agents can observe as public knowledge.

2.3 Kinds of Social Environments

In this section we look at concepts that are often used in the literature to define social environments. In particular we focus on *groups* as a general model for social environments, *institution* as a way to introduce normative concepts for agent interactions, the concept of *Virtual Organisation* which is commonly used to define collaborations between agents in highly dynamic social environments and *open agent society* defining the rules of interaction as norms in a social environment populated by heterogeneous agents.

2.3.1 Groups

Odell et al, in [91] define a group as a set of agents associated together to serve a common purpose. Groups have a separate identity within the social environment and can be composed of agents, as well as other groups [91]. In this way, a social environment can be understood as a society containing a set of groups that are formed to support agents interactions to solve common goals.

In general, there are three main concepts which characterise almost all the models that treat groups as an element of the social environment. These concepts are agent, groups and roles. For this purpose a well known formalism is the Agent-Group-Role (AGR) model [52], also known as the AALAADIN model, where an $a$gent is an entity which plays roles within groups. No assumption on the formalism of agents their internal architectures or constrains is made. The aggregations of agents in sets form a $g$roup. An agent can be a member of one or more groups at the same time. The $r$ole is an abstract representation of an agent function, service or identification within a group. An agent can have many roles, and a role is local to a group. The structure of the group, called $g$roup structure in [52] describes a group as a finite set of roles identifiers, an interaction graph specifying all the possible interactions between two roles and the interactions within the group.

In [71], Mandami et al focus on how interaction within a group can be stored and used to define a connected community. In this work, groups are defined as an aggregation of connected agents. Agents are associated with personal spaces or memories where their interaction history is kept. Individual agents can form a community group with an associated group space. The group space acts as a memory for the activities of the group. The group activities are then based on roles assigned to agents in order to manage the shared group space.

2.3.2 Artificial Institutions

Fornara et al in [53] explain the term *artificial institution* as a shared description of concepts and rules that regulate a fragment of the social environment. The function of an institution is to guarantee that a desirable outcome is produced if institutional norms are followed by all agents [55]. Institutional norms can thereby delimit the behaviour of an agent by specifying what an agent can or cannot do within the institution [84].

Earlier work on institutions, presented by Esteva et al [47, 49, 48], views agent interactions as delimited by a set of rule based protocols called Electronic Institutions. Later works were focused mainly on modeling and verifying institutional models [1] where artificial institutions are defined using explicit and formal representations of institutional norms. Generally, norms capture which actions performed by agents are allowed to be executed [53] in the environment. Thus, using norms, it is possible to describe protocols and general rules that apply in the social environment in terms of valid actions. Norms play an important function, in that if they are respected, they make the behaviour of the agent partially predictable [88]. Using norms to model interactions, agents are enabled to coordinate and plan their actions according to the expected behaviour of the others, as studied in [88, 16].

Norms are expressed in terms of what events are permitted, prohibited, obliged, empowered. These four elements are main concepts used and recognised in the literature to define institutional norms. The first three, namely the obligation, permission and prohibition, were identified by Searle in [108]. The formalisation of empowerment or institutional power was first introduced by Jones and Sergot in [72] to distinguish between permission and power to perform an act. We can informally summarise these four concepts as follows:

- **Permission** and **Prohibition**: The permission defines approval for an event to be brought about while the prohibition defines refusal to approve an event.
- **Obligation**: The definition of obligation specifies the events that agents are ought to bring about. Searle [108] explains obligations in terms of promises. An agent, by making a promise act, places oneself under a corresponding obligation. Once an agent places oneself under an obligation then (as regards this obligation), it ought to perform the corresponding action.
- **Institutional Power**: Jones and Sergot define the institutional power as, the constraints upon which the performance of a designated action by a specified agent is a sufficient condition to guarantee some speci-

[1]In Section 2.4 we present some of these works

fied (usually normative) state. In other words, the institutional power defines the capability of an agent to bring about an event in a meaningful way [72]. Without the institutional power, the event may not be brought about and it has no effect on the state of the institution [9].

In non regimental approaches, the actions performed by agents, may or may not follow the norms, resulting in a deviated behaviour from the one defined by the norms [36]. Thus, along with the definition of norms, enforcement mechanisms are required to motivate agent compliance by defining, in the case of norm violation [86], some loss of utility for agents. The contrary is also possible, if agents perform actions that benefit the institution, it is possible to apply a rewards in utility to motivate further compliance. The enforcement mechanism requires monitoring of the actions of the agents. By monitoring, the system can observe agents actions, recognise if they are complying with the norms or violating them and apply the proper enforcement mechanisms [86].

The new challenges in defining norms as the rules that govern agents social interactions are on how to change norms dynamically to keep up with the dynamic nature of the social environment. Theoretical aspects of norm change have been studied in [65, 18] and more practical mechanisms on how to define rule based constructs to deal with norm change have been proposed in [118]. The models of artificial institutions are usually thought to have norms that change very little over time. However, in other kinds of social agent environments, such as virtual organisations the need to add flexibility to the rules that apply in the social environment becomes essential.

2.3.3 Virtual Organisations

Other models for defining the social environment look at more dynamic approaches such as virtual organisations (VOs). Menard [84] suggests that there is a fine line between VOs and institutions. In particular, the main characteristic of an institution is that it is based on well established norms which can be considered not to change frequently. An organisation on the other hand, is more dynamic and adapts to allow agents to achieve their individual and/or collective goals of the organisation.

Foster and Kesselman [56] define VOs as organisations and individuals who bind themselves dynamically to one another in order to share resources within temporary alliances. Similarly to the artificial institutions, a VO defines interaction rules (policies) for which the organisation's members should conform to. The concept of organisation is mostly used in applications where collaborations amongst agents might be shorter and more dynamic. In some class of problems, for example ARGUGRID [3], agents that are autonomous members of the environment may need to interact within an organisation to achieve a common goal. There is a dynamic process for these interactions to take place where agents create an organisation by taking agreements (in an implicit or explicit way). The challenges here are many, from how to create an organisation to how to define the rules in such a way that agents can obtain agreements that are in line with their personal goals.

In order to be part of a VO, agents are required to adopt various roles empowering them with duties and responsibilities. Agents must act on behalf of their organisation to make important decisions such as the VO creation itself. In order to be able to do this effectively agents need to have an underlying process to support their social interactions. The process during which a VO collaboration takes place is called VO life-cycle. The VO life-cycle was first presented by Strader et al in [116] where the authors identify four main phases for the VO life-cycle. The four phases are summarised as follows:

- **Identification**: In this phase agents identify opportunities to organise with other agents in a way that it is suitable to achieve their goals. During this phase, agents select which is the best opportunity to pursue. The selection of the opportunity is the input of the formation phase.
- **Formation**: The formation phase defines the VO. This phase includes

a *partner identification*, a *partner selection*, and *partnership formation* as sub-phases. The *partner identification* sub-phase uses the information from the *identification phase* to identify a set of potential partners. Then, *partner selection* identifies the set of partners selected to work in a VO. The *partnership formation* sub-phase involves the actual formation of the VO. During this phase agents need to engage in negotiations to reach an agreement. Once the organisation has been formed, it can begin its operation phase.

- **Operation**: During the Operation phase of a VO agents have already an agreement. During this phase the specification of an agreement is executed. The input into this phase is all of the information related to the VO and the member agents gathered during the first two phases. The output from this phase is a summary of all of the activities and transactions that took place during the operation of the VO. The operation phase ends once the need to collaborate as a virtual organisation has passed for the member agents. Once this occurs the termination phase can begin.

- **Termination**: In the termination phase the goal of the VO has either been completed or failed to be completed. This phase includes *operation termination* and *asset dispersal*. In the *operation termination* sub-phase final information about the VO is finalised (such as evaluation of the members or inventory of the resources). In the *asset dispersal* process the agreement is terminated and any of the resources and the members of the VO are dispersed.

In [98], Pitt et al state that the important aspects of the VO such as its life-cycle are realised through agent interactions. Thus, agents are expected to be able to perform important decisions which may go through reiterations so that a there is a consensus between the interacting participants. Thus, interactions protocols become very important for the interoperability of the VO members [57] as they guide the interactions and support the decision making of agents. Many protocols for this purpose have been proposed in the literature to ease and coordinate the decision making of agents during the VO life-cycle from negotiation (i.e. Dung et al define a negotiation protocol to be used during formation phase [43]) to voting protocols (i.e. Pitt et al [98] formalise a voting protocol for shared decision making during the VO life-cycle).

2.3.4 Open Agent Societies

In applications where agents can solve their goals in isolation, there is no need for interaction or agent participation in agent societies. However, usually this is not the case, especially with distributed applications. The lack of information or resources makes an agent interact with other agents.

Pitt [96] defines an open agent society as a network requiring the interoperability of heterogeneous members. The heterogeneity that charachterises open agent societies means that the agent's internal reasoning and goals are unknown and yet such components have to interoperate for advancing their individual goals and/or common goals [96]. The membership of agents in the open society is dynamic and agents may compete with each other and may not conform to the rules of the society. Thus agents find themselves in a networked environment which can change in unpredictable ways, possibly as a result of the behaviour of other components which itself cannot be predicted [96].

In [9] Artikis defines the characteristics of an open agent society as a system where:

- Agents can be implemented by different parties therefore follow self-interested strategies when they act in the environment.
- No assumptions can be made on how the agent behaves in the system.
- Agents can enter and leave the system at any time.

Pitt et al [99] argue that the main feature of a system that enables open agent societies is that it needs to be governed by contractual and normative relations found in social interactions. In these systems agents are defined separately from the system itself. This decoupling implies that, when an agent enters in a system (environment), the system itself is required to have a way to prevent the agent from performing undesired actions and also can suggest permitted actions (if any) to the agent. This is why modeling normative relationships to govern agent societies is important to guarantee that the autonomy of the agent is preserved and to maintain the desired functionalities of the whole system.

Supporting *Openness* also implies that the social environment should have no dependencies with the reasoning of the agent and should be able, at runtime, to detect which agents in the network misbehave, without knowing the reasoning mechanisms of the agent. Thus, engineering a flexible and distributed solution to model the social interactions that includes normative relations is an important step towards *Open* Systems [59].

2.4 Implemented Social Environments

In the previous section we looked at the main concepts identified in literature for modeling Social Environments as a Norm-Governed MAS. These have lead to a large amount of theoretical work in terms of formal models and theories. In this section however, we consider only those frameworks and models that address the issues of implementing systematically a social environment to support the social interactions of agents.

2.4.1 AMELI

The Electronic Institution (EI) approach briefly described in Section 2.3.2 and presented in [48] is implemented by the AMELI framework [49] and uses organisational concepts to model the interaction of agents.

In [49], EIs are defined as composed of four main elements: dialogical framework, scenes, performative structure and norms. In most multi-agent systems every agent immersed in the agent environment has its own internal ontologies and languages, but in order to be able to interact with other agents, every agent must share a *dialogical framework*. The activities inside a multi-agent environment are seen in terms of multiple, concurrent dialogical activities.

The AMELI framework [49] is a tool implemented Java that offers a set of regimentation devices to block forbidden actions performed by interacting agents. Agent activities inside a multi-agent environment are seen in terms of multiple, concurrent dialogical activities. An agent has its own internal ontology and, in order to be able to interact with other agents, agents must share a dialogical framework. A dialogical framework is composed by *Scenes* and *Performative Structures*. Agents play a role within a protocol (*Scene*). A set of behavioural rules called (*Norms*) determine the legal sequences of locutions within an instance of a *scene*. *Scenes* can be connected into a *Performative Structure* which defines the relationships among *scenes* using transitions. The *Performative Structure* enables a developer to define dependencies such as choice points, synchronisation and parallelism mechanisms between scenes based on the roles of the institution, and flow policies among scenes specifying which paths can be followed by which agent's role.

The AMELI architecture is divided in three layers:

- An Autonomous Agent layer which represents the set of external agents acting in the system.
- A Social layer defining the infrastructure that mediates the interactions of agents based on the social rules.
- A Communication layer in charge of providing a reliable and orderly transport service.

External agents intending to communicate with other external agents need to redirect their messages through the social layer which is in charge of forwarding the messages to the communication layer. Erroneous or illicit messages may be blocked in the social layer to prevent them from arriving at their addressees. The social layer is represented by a multi-agent system and the agents belonging to it are called internal agents. AMELI defines a *Governor* agent for every external agent that acts in the system. The *Governors* keep the social state and decide whether to forward an act from an external agent to the *Scene Manager* or not. The *Scene Manager* agents maintain the state of the environment by deciding whether an act from an external agent is valid or not. EIs also support a *performative structure* that enables a developer to define dependencies such as choice points, synchronisation and parallelism mechanisms between scenes based on role flow policies among scenes specifying which paths can be followed by which agent's role.

One limitation of the AMELI framework is that for every external agent acting in the system there is a corresponding *Governor* agent to mediate the interactions of that agent. Additionally, the framework allows only for permitted actions to happen. It has been argued that regimentation is not always desirable or practical [72]. Therefore, in more recent works sanctioning (i.e. see [37]) and monitoring mechanisms (i.e. see [50, 86]) are preferred as opposed to developing regimentation devices.

An important feature of the Social environment is that the state of the interactions in the agent environment needs to be easily inspectable. In the AMELI framework this is not easily achievable as agents playing specific roles need to communicate to build a coherent state. It is also important that the social environment is represented in terms of a formal framework so that agents are enabled to reason about their actions and the effects their actions have. This is not achieved in AMELI because the nature of the framework is not logic based. Work presented later in this thesis tries to address some of these limitations within the AMELI framework.

2.4.2 AMELI+

To address the issue of maintaining a coherent state within the AMELI framework, in [61] Garcia-Camino et al propose AMELI+. AMELI+ is an extended version of the AMELI framework [49], with a mechanism to handle distribution of norms and possible conflicts [59] that may arise due to normative positions generated by the actions of agents. For example, one same action can be simultaneously forbidden and obliged or permitted.

The AMELI architecture is extended with additional normative manager agents which together with governor agents and scene manager agents are in charge of the system. The Normative agents deal with conflict resolution that can occur when applying normative transition rules.

Similarly to AMELI, AMELI+ is based on defining the agent environment using a hierarchical structure of internal agents to deal with the enforcement of norms. These internal agents decide how to update and propagate the changes made in the state of the system to other internal agents interested in these changes. The limitation here is that the AMELI+ approach requires many internal agents being involved into propagating messages for changes they perform locally. In a large and dynamic agent environment, where the number of agents and the way they interact is not predictable a priory, this solution is not practical. Agents need to be able to inspect the environment and make decisions on their actions at run time, which is not easily achievable here due to the hierarchical structure of the system. Another AMELI+ limitation is that the language for defining norms within the system does not support constraints nor it is possible for agents to reason about how the properties of the agent environment are changing in time.

2.4.3 CArtAgO

The CArtAgO (Common Artifact for Agent Open Environment) framework (Ricci et al [105]) overcomes the limitations of having hierarchical structures managing the agent environment by defining the Agents and Artifacts (A&A) [104] meta-model. The A&A meta-model introduces the concept of artifact as a first class abstraction to model the agent environment. Artifacts are conceived as passive, function-oriented computational entities designed to provide some kind of function, and then to be used by agents to support individual and collective agent activities [104]. A&A model introduces the concept of workspace as a container of agents and artifacts. The agent environment is conceived as a set of distributed workspaces containing dynamic sets of agents working together by communicating and by sharing artifacts.

CArtAgO provides an API to define artifacts, an API to instantiate, use, manipulate artifacts, and a run-time environment supporting the existence and dynamic management of working environments. CArtAgO does not provide a specific model or architecture for agents and agent societies. Instead CArtAgO allows a developer to build agent societies that share the same working environments, and interact through suitable mediating artifacts besides communicating via speech acts. Agents can use an artifact by triggering the execution of an operation through a usage interface provided by the artifact and by perceiving observable events generated by the artifact itself as observable properties.

The basic elements of CArtAgO are:

- Agent bodies, which are the entities that make possible to situate agents inside the working environment;
- Artifacts, which are the building blocks to structure the working environment;
- Workspaces, which are the logical containers of artifacts, useful to define the topology of the working environment.

CArtAgO offers a distributed topology by means of distributed workspaces. By using workspaces to mediate the interactions amongst agents it is possible to define agent interactions in terms of protocols as we have described in previous work in [123, 122, 81].

CArtAgO supports social constraints but its specification language for modeling the artifacts, ReSpecT [92], is not as well suited for the definition of normative concepts. This is due to the fact that with ReSpecT it is possible only to define event-condition-actions (ECA) rules with a specific syntax, but, it is not possible to reason about how the state of the environment changes in time. Another limitation of the CArtAgO framework is that it does not provide a methodology on how to define a norm based system thus, the developer has to design the social environment without guidance on how agent interactions can be specified using norms.

2.4.4 Moise+ Organisation Model

Moise+ [67] proposes organisational notions (like roles, groups, and missions) to enable explicit specification of organisations in a multi-agent system. This specification is used both by the agents to reason about their organisation and by an organisation platform to enforce the rules of the organisation.

The specification of an organisation in Moise+ is decomposed into three dimensions: structural, functional and deontic. The structural dimension specifies roles, groups and links within the organisation. The functional dimension specifies how the global collective goals should be achieved. The deontic dimension binds the structural dimension with the functional dimension by specifying permissions and obligations.

In a more recent work [68], the Moise+ [67] specification of organisation is defined on top of CArtAgo [105] which, as stated in the previous section, is based on the A&A model of Artifact [104].

The resulting framework is named ORA4MAS [68] and uses Artifacts and the Moise+ language to define a basic set of organisational artifacts as follows:

- The *OrgBoard* artifact keeps track of the state of deployment of the organisational entity and it contains Moise+ specifications that agents can reason about.

- The *GroupBoard* artifact manages the group instances of an organisation by regimenting two norms that define the structural properties that the group should have (i.e. such as an agent might be forbidden to adopt a new role if it is incompatible with the roles that it is already playing in the group). By observing this artifact agents can understand available roles and their constraints, the participants of the group and the links of the *GroupBoard*.

- The *SchemeBoard* artifacts are used to support and manage the execution of a social scheme. The execution of a scheme has three phases: i) formation, where agents commit to the mission of the scheme; ii) goal achievement, where each agent has to achieve the goals of the mission they are committed to; iii) and finally it finishes when the root goal of the scheme is satisfied and the scheme can be removed from the organisation.

- The *NormativeBoard* artifacts are used to maintain information concerning the agents compliance to the norms by managing permissions and obligations defined between roles and missions. When an agent starts playing a role within a group which is responsible for some scheme, instances of norms are created for the agents in all related normative boards. The instantiation process copies the norm definition and grounds the variables of the norm. The norm is initially inactive and becomes active if the conditions defined by the norm hold. The norm becomes fulfilled when the agent executes the action as it is specified by the norm.

An organisational entity is then defined by one *OrgBoard*, one *GroupBoard* for each instance of a group of agents, one *SchemeBoard* for each scheme being executed by the agents and one *NormativeBoard* for each *SchemeBoard*. These organisational artifacts are linked together to allow the synchronisation of the operations and to maintain a coherent and consistent state of the organisation by sharing their information. The links between the artifacts are the following ones: an *OrgBoard* is linked to all other artifacts of the organisation; a *GroupBoard* is linked to all schemes its agents are responsible for; the *SchemeBoard* is linked to exactly one *NormativeBoard* that verifies the status of the norms related to the execution of the scheme; and finally a *NormativeBoard* is linked to its *SchemeBoard* and all *GroupBoards* responsible for the corresponding scheme.

ORA4MAS overcomes many of the limitations we first described in Section 2.3.2. Instead of the hierarchal structure of internal agents, developers can define specialised artifacts that mediate the interactions using their own mediation rules. Agents can act and perceive artifacts that are distributed over a network. They also encapsulate a state which changes as agents act upon them and the interested agents can perceive it in a coherent way. Finally, ORA4MAS links the distributed states of the artifacts in the environment to maintain coherent information about the state of the environment.

There are still limitations to overcome in designing social agent environments. In ORA4MAS the interactions cannot be combined in different ways and it is not possible for agents to coordinate to change the rules when necessary (i.e. by switching artifacts). This type of coordination has not been addressed here because agents are assumed to interact withing a set of fixed schemes defined in the organisation. The linking of artifacts in fact is rigid and defined at design time, therefore adapting it to different situations that may emerge during agent interactions is difficult. Moreover, there is no general mechanism to query the state of distributed artifacts nor is there support for reasoning about temporal events within artifacts. Finally, ORA4MAS abstracts away from organisations that are part of a physical environment, therefore, it does not provide a general framework for defining agent environments (physical and social).

2.5 Event-centric social interaction

We are concerned with developing an event-based approach to social interaction. Thus, in this section we review existing work that is most relevant to our goal of extending GOLEM with a complementary model for a social environment.

2.5.1 Artikis et al

Artikis et al [13] propose a model for norm-governed multi-agent systems as executable specification of open agent societies. An *Open Agent Society* is an open agent system where each agent occupies at least one social role and where the behaviour of the members is governed by a set of social laws. To formulate these specifications the authors use C+ [64] which has the advantage that it can be given an explicit semantics in terms of transition systems and allows for verification of properties. An agent enters in a society after establishing its role. In this way, the social constrains prescribe the behaviour of an agent occupying a specific role.

Artikis et al [13] suggest that in open agents societies there is a clear distinction between physical capabilities, institutional power, permissions and sanctions to enforce policies.

In [9, 13] Artikis et al specify four levels of rules in an agent society:

- The rules that describe the physical capabilities of agents;
- The rules that define the institutionalised powers [72];
- The rules defining permissions, prohibitions and obligations of the agents;
- Sanctions and enforcement policies that deal with the performance of forbidden actions and non-compliance with obligations.

In [10] the authors distinguish between valid and invalid actions through the *institutionalised power* concept. The term refers to the empowerment of agents within an institution to create facts that have a conventional significance within that institution. Searle [108] was one of the first to distinguish between brute facts and institutional facts. Later, Jones and Sergot [72] presented a formalisation of this concept in terms of the conventional significance of certain acts, in that they *count as* other kinds of acts or states of affairs. For the actions to have an implication within the institution the action must be performed by an empowered agent.

The social constrains in Artikis et al work define in what state of affairs an action can initiate/terminate. To reason about social constrains, the authors use Event Calculus [77] which is a formalism to specify and reason about actions and their effects [2]. They use an implemented version of Event Calculus which the authors refer to as *EClp* and which briefly presented in [11]. Within this approach the same action, in two different institutions might have different semantics as it can initiate/terminate different states of affairs. To specify the constrains of the open system the authors specify first when an action is valid:

```
holds_at(valid(Agent, Action)=true, T) ←
        holds_at(pow(Agent, Action)=true, T),
        happens(action(Agent, Action)=true,T).
```

The above Event Calculus specification states that a valid action counts as an action at a point in time if the agent who performed that action had the institutional power to perform it at that point in time.

Secondly the authors specify permissions, prohibitions and obligations in an application dependent manner. For example the constraint bellow:

```
holds_at(permitted(Agent, Action)=true, T) ←
        holds_at(pow(Agent, Action)=true, T).
```

[2]The Event Calculus is extensively explained in Section 3.3.1

is a way to expresses that an act is permitted at a point in time, if the agent who performs it has the power to do so.

Thirdly, based on permissions, prohibitions and obligations and on the actions performed by the agents, their behaviour can be classified to social or anti-social accordingly. To do so the authors specify sanctions and enforcement policies in an application dependent manner.

Similarly to Artikis et al [12, 13], in Chapter 4, we will model executable specifications in terms of social game using the Event Calculus formalism. In particular we have chosen to use the Ambient Event Calculus (AEC) formalism [22], as it extends the Event Calculus by allowing us to specify complex events happening in a distributed agent environment [3]. By using AEC we can query properties of the state of the social games and reason in a distributed setting. In Chapter 5, we also look at the social games in terms of components that can be dynamically combined and executed using coordination patterns, which are not addressed in Artikis work.

2.5.2 Alberti et al

Alberti et al [7] propose a logic-based approach for the specification and verification of agent interaction. To express interaction protocols and to give a social semantics to the behaviour of agents, the communicative acts of the agents are constrained through the definition of constrains called *Social Integrity Constraints* (SIC) [8].

The agent society is composed of a social infrastructure and a knowledge base, containing information about structure and properties of the society. The social knowledge base contains information about protocols and regulations for entrance, exit and role assignment.

The society records in a history *HAP* the observable and relevant events for the society (happened events are denoted by H). A course of events *HAP*

[3]The Ambient Event Calculus is extensively explained in Section 3.3.3.

might give rise to social expectations about the future behaviour of its members, the expectations are collected in a set EXP that contains events which are expected to happen (denoted by the functor E), and events which are expected not to happen (denoted by the functor NE).

Using SIC, the authors can describe the evolution of the expectations in the society, based on the current history. Social integrity constraints are used to express that expectations are raised on the behaviour of agents as consequence of their communicative acts. For example, to express that if an agent does accept a request, it is obliged to fulfill it by some deadline is expressed as follows:

$\mathsf{H(request(Agent1, Agent2, Content, Dialog,} \ \mathsf{T}_\tau \ \mathsf{))}$
$\wedge \ \mathsf{H(accept(Agent2, Agent1, Content, Dialog,} \ \mathsf{T}_a \ \mathsf{))}$
$\wedge \ \mathsf{T}_\tau < \mathsf{T}_a$
$\rightarrow \mathsf{E(do(A, B, P, D, Td))} : \mathsf{Td} \leq \mathsf{T}_a + \tau$

where *do* is a physical action, which fulfills the expectation of the constraint if it matches with its content, provided that it is performed before a certain amount of time τ has passed since the request has been accepted.

The expectations E are represented by 2 arguments: the first argument is the event associated with the expectation, the second is a list of constraints over the variables contained in the event. An event (defined by terms beginning with "do") fulfills an expectation if the contents of the event satisfies the constraints of the expectation. In case of negative expectations, instead of fulfillment there is a violation.

Expectations can be viewed as obligations which we represent within a protocol as it will be explained in Chapter 4. We will also present in our approach coordination mechanisms amongst protocols which complements the Alberti's et al work presented in this section.

2.5.3 Cliffe et al

In [35], Cliffe et al describe the use of Answer Set Programming (ASP) [15] as an institutional modeling technique. ASP permits the statement of problems and queries in domain specific terms thus eliminating the gap between specification and verification language. Similarly to Artikis et al, Cliffe et al provide a formal Event Calculus like model for the specification of institutions that captures all the essential properties namely empowerment, permissions, violations and obligations and a verifiable translation to ASP resulting in a decidable and executable model for institution. This work models an institution as a set of institutional states that evolve over time subject to the occurrence of events. In particular, the institution is defined as a quintuple that consist of institutional events, fluents, a consequence relation, an event generation relation and an initial state:

- The *institutional events* describe events that may occur within the institution. The events can be observable events or institutional ones.
- The *institutional fluents* comprises the union of four distinguished sets of fluents: the set of domain fluents that describes the domain of the institution; the set of institutional powers that denotes the capability of some event to be generated in the institution; the set of event permissions that denotes what is permitted when an event is brought about; and the set of obligations that denotes which event should be brought about before the occurrence of event. Institutional fluents may be held to be true at some instant of time.
- The *consequences* define a function that describes which fluents are initiated and terminated by the occurrence of a certain event in a state matching some criteria.
- The *event generation* defines an event generation function which describes when the occurrence of one event counts as the occurrence of other events inside the institution.

- The *initial state* defines the set of fluents that hold when the institution is created.

The institution changes state due to events taking place. The start of an institution is defined in terms of fluents that are true at a given time. The authors define then an event generation function that describes what events are generated at a given state of the institution (i.e events can be generated from other events or due to violations). Events have effects on the fluents as they may alter their truth value at a given time. For reasoning with ASP, the authors define an ordered trace as a sequence of observable events. The ordered traces allow the system to monitor the evolution of the institution over time. The ASP has a one-to-one relationship with the institutional event traces of the formal model, therefore the authors can provide executable institutional specification so that agents can dynamically compute and query to establish both how the current institutional state was reached and which actions will have what consequences in the future of the current state.

An example of definition of the social rules is shown bellow:

```
occurred(E,T) ← observed(E,T).
occurred(viol(E),T) ← occurred(E,T),
                      not holds_at(perm(E),T),
                      event(E), event(viol(E)), instant(T).
```

The above definition shows a way to define what occurs in the social environment. In particular, the first clause states that an event has occurred at a time T if it is observed at that time. The second clause states that a violation due to the event E has occurred at time T, if the event E has occurred at time T and it was not permitted. The event(E) in the second expression denotes the type of the event and instant(T) denotes the time instance.

There are many similarities between our work and the executable specification of an institution as proposed in [35]. Similarly to what we stated in

Section 2.5.1, by using the AEC formalism we can express the normative relations and the evolutions of the institutions in terms of how a game evolves due to actions taking place. However, in addition to these approaches, we consider the distribution of the agent environment. This has as a consequence that the state of the environment itself is distributed and in order to meditate agent interactions we need ways to identify what properties hold in the distributed state. Additionally, we define coordination mechanisms to provide a component based mechanism for defining agent environments. On the contrary however we cannot verify system properties as it would be possible by using ASP.

2.5.4 Fornara et al

Another model of interaction in a multi-agent system understood in terms of artificial institutions is that of Fornara et al, where the authors define OCEAN (Ontology, Commitment Authorisation Norms) [54, 55]. OCEAN describes an abstract syntax and semantics for the components of an institution. The main components of an institution in Fornara's et al view [55] are a core ontology for the definition of entities and their properties (such as natural attributes, physical properties, institutional attributes), their actions and for the specification of roles and events (such as passing of time calls, signals, change in state); a set of conventions and authorisations for the performance of institutional actions; a set of event-condition-action rules (ECA) that are essential for the definition of norms.

Fornara et al focus on the importance of representing commitments as the fundamental concept of an institution. Commitments are viewed as an essential concept to express the meaning of most types of communicative acts and to define norms. A social *commitment* expresses a relation between at least two agents [30] and it is defined as follows:

commitment(State, Debtor, Creditor, Content) which expresses a commitments in terms of a state *State* that keeps track of the life-cycle of the commitment, a debtor agent *Debtor* that is committed to fulfill the *Content* of the commitment, a creditor agent *Creditor* who the debtor agent is committed to. The content is expressed using temporal propositions. A temporal proposition is characterised by a statement about a state of affairs or about an action. A temporal proposition is represented as follows:

tp(Statement, [Tstart, Tend], Mode, TruthValue)

which defines a *Statement* about a state of affairs or about an action. The statement is referred to a time interval with two possible different modes: *exist* or *for all*. The *Truth-value* of a temporal proposition is initially *undefined*, it becomes *true* if the *Mode* is *for all* and the statement is true for every instant of the associated time interval or if the *Mode* is *exist* and the statement is true for some instant in the associated time interval, otherwise it becomes *false*. For example the following statement:

commitment(pending, a1, a2, tp(open(a1, auction), [now, now+10m], $\exists$, $\perp$)

represents the commitment of an agent a1 to the agent a2 to open an auction within 10 minutes.

A commitment has a life cycle that evolves due to actions performed by agents or to domain-dependent events. The events modify the truth value of the temporal proposition in its content. If its temporal proposition becomes true, the commitment becomes fulfilled; if it becomes false the commitment becomes violated. An agent creates a commitment by performing a *makeCommitment* institutional action, which creates an unset commitment. The debtor of an unset commitment may refuse it by executing *setCancel*, or it may undertake the proposed commitment by executing *setPending*. A refused commitment is represented with a canceled state, whereas an accepted commitment is depicted with the pending state.

Norms then are defined as those rules that manipulate commitments of the agents and the general structure is defined as follows:

```
on E:Event-Template
if Condition then
    foreach Agent in Selection-expression
    do Commitment-Operations
```

where *Agent* is an identifier on the set of agents that satisfy the *Selection expression*; selection-expression is a list of agent identifiers or a role defined in a certain artificial institution; *Commitment-operations* is a sequence of commitment operations (i.e cancel, set ect). Due to this kind of modeling an obligation is represented by commitments, a prohibition is a commitment to not perform the prohibited act and violations are commitments that have been violated. A strongly related work to the Fornara's et al is the work proposed by Yolum and Singh in [134]. The authors define interaction protocols where agents manipulate commitments by means of a set of operations. The operations are namely: creating, discharging, canceling, releasing, delegating or assigning commitments. The discussion provided in the following paragraph is similar for both of these two lines of work.

Within OCEAN Fornara et al make explicit the model of commitments as a mechanism to deal with what are the obligations, prohibitions and permissions of agents in the system. Commitments however need a generation mechanism which, if it is done by agents as the author suggests, it might not necessarily be a good solution when agents are situated in an open agent environment. Also, OCEAN does not address coordination of such agents when creating the commitments, therefore, there is no guarantee that the system converges towards a cooperative environment. In some complex applications, the number of commitments generated for every agent can be high. Thus, commitments need an ordering mechanism to define which one of the pending commitments should be performed first by the debtor of the commitment.

In our approach, we will view commitments as obligations, similarly to the Artikis and Cliffe's work.

2.5.5 Stathis and Sergot

In [113], Stathis and Sergot view interactions as a rule governed activity which is regarded as a game. The games metaphor was originally proposed to model human-computer interaction by Stathis and Sergot in [115] and was subsequently applied to formulate agent interaction protocols in [113].

The basic unit of the games metaphor is the notion of an *a*tomic game which describes a set of rules about an initial state, a set of player roles, a set of game moves, the effects the moves have on the state, a specification of when a move is valid, a set of terminating states, and a set of results [113].

A move in Stathis work is represented as an Act selected by a Player as follows:

```
select(Player, Act).
```

Stathis and Sergot describe the rules of a game using the following logic program:

```
game(State, Result)←
     terminating(State, Result).
game(State, Result)←
     not terminating(State, Result),
     valid(State, Move),
     effects(State, Move, NewState),
     game(NewState, Result).
```

Specifying the valid moves of a game corresponds to defining the preconditions on the state of the environment that validates an actions in the multi-agent system. In particular, a valid move is defined as:

```
valid(State, Move)←
   available(State, Move),
   legal(State, Move).
```

Every move in the game can produce effects which can be due to either permitted acts or to forbidden or violations of agents. The rewards or punishment of agents are described as effects of the moves in the game. The game terminates when the termination conditions become true. The result of a game does not necessarily need to be zero-sum [90], by requiring a winner and a loser, but it can also give rise to a win/win or loose/loose situations.

Given a specification of the rules, implementation of an interactive system requires construction of an umpire, a component that enforces compliance of the players with the rules and thereby controls the interaction. The umpire displays the state of the game, provides means by which players select moves, and enforces compliance of all the players with the rules.

The Stathis and Sergot framework includes a coordination component by defining the conditions under which a sub-game becomes active. The coordination mechanism specify how games interleave in a compound game and it is specified as:

```
valid(State, Move) ←
   active(State, SubState),
   valid(SubState, Move).
```

The *active/2* definition determines which sub-game *SubState* can be interleaved in the compound-game.

There are various versions of this framework, the earliest versions were based on assert retract predicates, but later versions were event based [114]. Although the framework has an important idea of treating protocols as atomic games and coordinate the protocols through use of an umpire, the framework has not been tested in distributed applications. Like [13], we are going to look at how to accommodate more complex normative relations than the *legal* definition provided in Stathis and Sergot's work. In particular, we are going to provide a clear separation between the physical and the social state of the environment and how the moves of agents influence the two. We are also going to extend our work with coordinations patterns to make the framework more applicable to a MAS.

The games metaphor provides an intuitive model for representing social interactions. From the engineering perspective, the developer of the social environment has a structured mechanism to define the interaction rules as atomic games by simply specifying the set of moves, their effects and the initial and the final state of a game. More complex interactions can be specified by:

- expanding (adding more valid moves) and filtering (removing moves by disabling their validity) from existing games; and
- combining simpler, component sub-games to build more complex games and coordinate moves by using different interleaving patterns.

From the agent perspective, given the specification of a game, agents are enabled to reason about the rules and build strategies to bring about success in their individual goals. Agents can reason using a shared interpretation of games or forced to play a game in the presence of an umpire. In both cases, i.e. sharing an interpretation of the rules or playing the game via an umpire, the framework provides a standard and flexible way with the implementation of the actual interactions. Moreover, by being able to reason about the evolution of games, agents are also enabled to play many games in parallel.

As we have argued throughout this section, the current event-based approaches are not game-based approaches and these models of social interaction do not integrate agents interaction and coordination. Games have the potential of combining both into a single approach with the advantages mentioned above.

2.6 Summary

This chapter was aimed at giving a comprehensive background on how current literature views interactions in social environments. We presented the concept of Social Agent Environment as a first class abstraction. Then we looked at the components that are common to social environments, namely member agents, physical territory, social rules and social state.

We reviewed the ideas found in literature on how to specify social environments. In particular, we looked at groups as social environment, which often provides very general and theoretical models. Sometimes it is convenient to describe normative aspects on how agent in a community/group interact, thus institutions are another kind of social environment that regulate interactions. The definition of an institution is based on defining normative concepts such as the institutional power, permission and prohibitions. Another way to represent social environments is by using organisational concepts which are more suitable to capture operational aspects of agent interaction. Openness is another important aspect in social environments which is not always considered. Open agent societies extend the concept of institution by (ideally) addressing how agents enter and leave a society and how agent can interact in multiple societies/collaboration/organisations or groups.

We then looked at existing implemented frameworks that model social environments. We introduced AMELI/AMELI+ which are based on hierarchical structure of agents thus making it difficult to define and maintain a coherent social state and to define a non regimental approach to agent interaction. We also described CArtAgo, Moise+ and ORA4MAS which introduce improvements to the AMELI/AMELI+, but also them have disadvantages such as expressiveness of the norms, dynamic coordination of different interactions and reasoning with events time. Moreover, none of these platforms interface or address applications that require a model also for the physical environment and the physical acts.

We also reviewed existing work in the area of social agents and MASs. We focused only on those works that follow an event-based approach such as ours. We introduced the research proposed by Artikis, Alberti, Cliffe, Fornara and Stathis as works that provide a formal and an executable specification of interaction, including a basic formulation of normative concepts. In all of these works the institutional or organisational models are used as a way to model agent societies. The interaction of agents is based on social rules specified by organisations or institutions. Agents can enter an agent society and take roles within it. Their roles will determine the way they interact within the environment.

In the next chapter we introduce GOLEM. GOLEM is a platform for developing physical agent environments. In this thesis we want to extend GOLEM with an additional social environment structure so that agent interactions can be modeled in terms of physical and social acts.

Chapter 3

The GOLEM Platform

The main motivation behind our work is to extend the GOLEM platform with the notion of Social Environment. The GOLEM platform was developed as part of ARGUGRID (Argumentation over the Grid) [3], an EU research project investigating a new model for programming a service Grid at a semantic, knowledge-based level of abstraction through the use of argumentative agent technology. In this chapter we first introduce the reader to the GOLEM architecture and then analyse the limitations of the platform. In particular we show how GOLEM focuses on representing physical aspects of an agent environment but it abstracts away from the social aspects of the environment.

The chapter is organised as follows. The functionalities of GOLEM are illustrated through the Distributed Packet World example as presented in Section 3.1. In Section 3.2 we present the various components of the GOLEM platform and we describe their functionalities. In particular we explain the three main components of GOLEM: *containers, objects and agents.* In Section 3.3 we discuss how, given a representation of the agent environment, it can evolve in time. Finally, in Section 3.4 we discuss the limitations of GOLEM, namely, that it is missing any form of modeling social rules within the agent environment.

3.1 Distributed Packet World Scenario

Bromuri and Stathis [22] and Bromuri [19] illustrate the functionalities of GOLEM through the Packet World (PW) scenario which was originally presented in [130]. In this scenario a set of agents are situated in a rectangular grid consisting of a number of differently coloured packets and destinations points. Packets are delivered to destinations that have the same colour as the packet. Each agent living in the PW has a battery that discharges as the agent moves in different locations in the grid. The battery can be recharged using a battery charger. This charger emits a gradient whose value is larger if the agent is far away from the charger and smaller if the agent is closer to the charger. The agents have the goal to bring the packets to the collection points and can communicate with other agents to exchange information about the environment. For example, agents can coordinate by placing flags in locations for letting other agents know that a particular area has been explored and has no packets left.

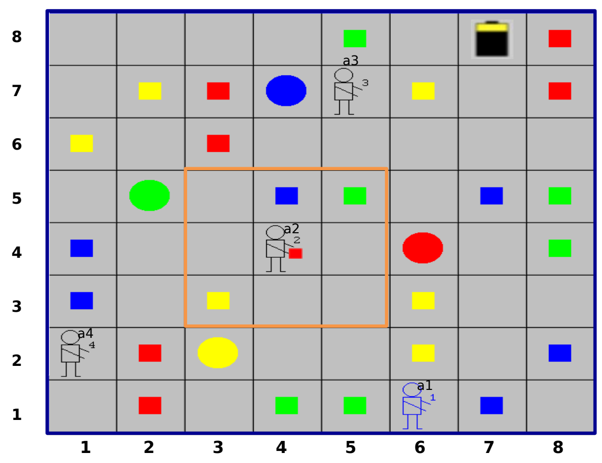


Figure 3.1: The Packet-World Scenario.

Fig. 3.1 illustrates PW with a set of agents that are situated in a 8 x 8 rectangular grid consisting of a number of coloured packets (squares) and destination points (circles). Agents (a1, a2, a3, and a4 in Fig. 3.1)) move around the grid to pick coloured packets which they must deliver in destinations that match a packet's colour. As agents can see only part of the grid at any one time, the red square around agent a2 represents the perception range of this agent. The battery of the agents in the grid can be recharged using the battery charger situated in location (7,8) of Fig. 3.1)).

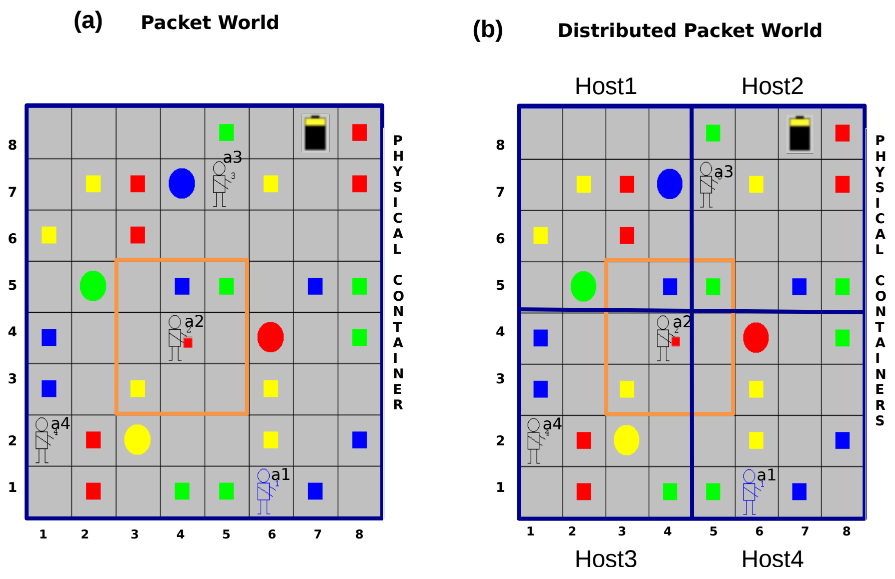


Figure 3.2: Distributed Packet-World Scenario.

Bromuri and Stathis [22] extend the original packet world with a distributed version of the Packet-World scenario, where the world is split in many parts and runs in multiple hosts as shown in Fig. 3.2. Every different host is responsible for the agents and the packets deployed within it. Agents can perceive what happens in a location that is logically nearby, independently of whether this location is distributed in another host.

3.2 The GOLEM Model

GOLEM (Generalised Onto-Logical Environment for Multi-agent Systems) is an agent environment middleware that can be used to create multi-agent system applications. GOLEM supports the deployment of *agents* - cognitive entities that can reason about sensory input received from the environment and act upon it, *objects* - resources that lack cognitive ability, and *containers* - virtual spaces containing agents and objects and capturing their ongoing interactions in terms of an *event-based* approach.

GOLEM uses the concept of *affordances* to describe “all the action possibilities latent in the environment, objectively measurable, and independent of an agent’s ability to recognise those possibilities” [63]. In this way, GOLEM entities suggest how agents can interact with them in a way that can be designed in advance and declaratively thus providing a link between the way entities are described and the way these descriptions can be perceived by agents.

To define an agent environment the developer describes a set of *containers* where *agents* and *objects* are deployed. Agents use their *sensors* to perceive the status of other entities in the agent environment and perform actions via *effectors* to change the environment.

Interaction between agents or agents and other entities in GOLEM is formulated in terms of events happening in the environment. According to the happening of an event the agent environment notifies those sensors capable of perceiving the action of the event. In GOLEM there are three types of acts embedded in an event: *speech acts* - to allow agents to communicate with other agents and users; *sensing acts* - to allow an agent to actively perceive the environment; and *physical acts* - to allow the agent to interact with other entities, in particular objects, but also agents as well.

3.2.1 Objects

GOLEM objects are entities that lack of cognitive abilities. They provide functionalities in the agent environment via their affordances and can be interacted upon via simple sensors and effectors called receptors and emitters respectively.

The object is described in terms of the perceived affordances. To present these perceived affordances Bromuri and Stathis [21] use the object-based notation of C-logic [31], a formalism that allows the description of complex object. As we are going to keep this notation in this thesis we have included in Appendix A a short description and explanation of C-logic. As an example of a C-logic term consider the description of a packet in the packet world (taken from [19][1]):

```
packet: o1[
            colour ⇒ red
            receptors ⇒ { receptor:r1 },
            emitters ⇒ { emitter:em1 }
            position ⇒ square:sq1
            ].
```

This complex term states that o1 is a complex term describing an object of class packet, with a red colour, with a multi-valued attribute receptors containing one receptor sensor r1, a multi-valued attribute emitters containing one emitter effector em1 and an attribute defining the object position. Some of the attribute values are complex terms themselves, like the receptor and the emitter. The C-logic syntax to represent the perceived affordances of an object as a complex term has a first-order logic translation, see Appendix A for an example of translation.

[1]The discussion of the object affordances and the discussion in appendix A is based on Bromuri's thesis.

The receptor sensor of an object captures notifications of *physical acts* performed on the object by entities in the agent environment. To represent events that receptors can capture complex terms are used too. The term:

```
do:e1 [ actor ⇒agent:ag1[effector ⇒ ef1], act ⇒ drop,
         object ⇒ packet:p1, location⇒square:sq1]
```

describes an event e1 where the effector ef1 of agent ag1 performs a physical act drop on a packet object p1 in location sq1. The event produced in the agent environment will be captured by the receptor of the object via notification sent to the object by the environment in order to trigger the method of the object. A triggered method is likely to change the internal state object state and typically will result in the output of the change transmitted as another event via the object's *emitter* effector. As before, emitted events are complex terms. To simulate an object reaction to the physical act represented by e1, the event description:

```
do:e2 [actor ⇒ object:o1[emitter ⇒ em1],
        act ⇒ produce_sound,
        sound ⇒ packet_drop]
```

showing the kind of event emitted by the object. For more details see Bromuri [19] and Bromuri and Stathis [22].

3.2.2 Agents

In general, the *agent* concept is attributed to *a*utonomous software component demonstrating proactivity by acting without direct human intervention. Due to their autonomy, agents have control over their actions and their internal state [70]. In this thesis, we refer to agents that are *s*ituated in an environment and whose autonomy relates and depends also on their environment.

Given a class of problems, an agent is engineered to solve it. This is why agents need knowledge and reasoning capabilities. Agents have their own knowledge base, a set of goals, a private state and a plan and strategies on how to achieve the goal [132]. Generally, it is not possible to observe the mental state of an agent (i.e an agent has a goal and a plan on how to achieve it, this agent might have also a strategy on how to optimise the profit for a specific goal but this is all private to the agent), however it is possible to infer the agent behaviour from the acts the agent performs in its environment.

Agents in GOLEM are cognitive entities that can reason about sensory input received from the environment and act upon it. An agent has a *body* and it is described by affordances. Affordances then can be perceived by other agents making possible the interactions among them. A description of the form:

```
agent: ag1[ understands ⇒ ontology:o1,
            sensors ⇒ {sight:s1, hearing:s2, touch:s3},
            effectors ⇒ {speak:ef1, arm:ef2, arm:ef3},
            activity ⇒ idle
          ]
```

states that ag1 is a cognitive agent which understands the ontology o1, has sensors of class sight, hearing and touch, and effectors of class speak and arm, while it is currently idle. An agent attempts to execute physical actions in the agent environment using its effectors, and it uses sensors to respond to event notifications by the agent environment.

The body of an agent has affordances which makes the agent perceivable in the agent environment, a set of sensors and effectors to act and perceive actions in the environment and, additionally, it contains a *mind*, a cognitive component giving the agent the ability to reason logically and make decisions. GOLEM also provides mobility functionalities, which allows that the agent migrates from one container to another. The details on the agent architecture and the mobility features of GOLEM are beyond the scope of this introductory chapter; the interested reader is referred to [21, 19] for more details.

3.2.3 Containers

In an agent environment, agents can be computationally expensive to deploy and this can limit how many agents can be deployed in a single machine. GOLEM tackles the scalability problem with the notion of *container*. A GOLEM container supports the distribution of agents, processes and objects in an agent environment thus enabling them to interact regardless of their location. To define the agent environment, a set of containers is composed at design time to link the entities distributed in different machines and synthesise the environment.

A container has a *state* that acts as a directory of all the present agents and objects in it, including information about their topology and configuration. Interactions between the entities within a container are governed by a set of *physical laws*. These laws specify the possible evolutions of the container, including how these evolutions are perceived by agents and affect objects and processes in the environment.

Every container has its own perceived affordances that include the ways in which an agent can configure itself (or other basic objects, agents, and containers) to became part of the container's internal state. For example, the term below represents the state of a 2 x 2 packet world container showing only one agent, packet, destination and battery:

```
packet_world:c1[
 address ⇒ "container://one@134.219.7.1:13000",
 laws ⇒ physics:pw1,
 type ⇒ open,
 mediation_services ⇒ {communication, discovery, agent_registry},
 entities ⇒ {agent:ag1, packet:p1, dest:d1, battery:b1}]
```

describes a packet world container that has an address which is container://one @134.219.7.1 :13000, has laws that are represented by another object pw1 of class physics, is an open type of container in that any agent can enter it, and has an internal state containing four entities, one agent ag1, ag2, a packet p1, a destination d1 and a battery b1. The representation of the 8 x 8 grid of Figure 3.1 is similar but larger as it contains more agents, packets, destinations, and squares.

The container offers mediation services which are available for all the entities in the system: the discovery service of the container is used by the agents to query the location of other agents, objects or containers in the agent environment; the communication service delivers exchanged messages between agents to the destination; the connector mediation service allows agents to move from one container to another container of the agent environment.

A set of containers connected together defines a *topology* of containers. The topology relates the containers by defining how they are connected which also determines how the containers can query the whole state of the agent environment [2]. The *topology* can be defined in terms of a root container, neighbouring containers and/or super and sub containers. The root container is the ancestor of all the containers of the topology. For every container one or more super, sub or neighbour relationship with the other containers can be defined as shown in Fig.3.3.

Containers that are deployed within other containers are said to be sub-containers. The idea is that the sub-container is logically inside the super container, but possibly physically distributed elsewhere. A container can also be connected to one or more adjacent containers which in GOLEM are called neighbouring containers. In Fig. 3.3, container C0 is a root container connected to three sub-containers C1, C2, C3 that are neighbours between each other. Moreover, also C1 and C3 have sub-containers organised in a neighbourhood.

[2]The details of such mechanisms are explained in Section 3.3.3.

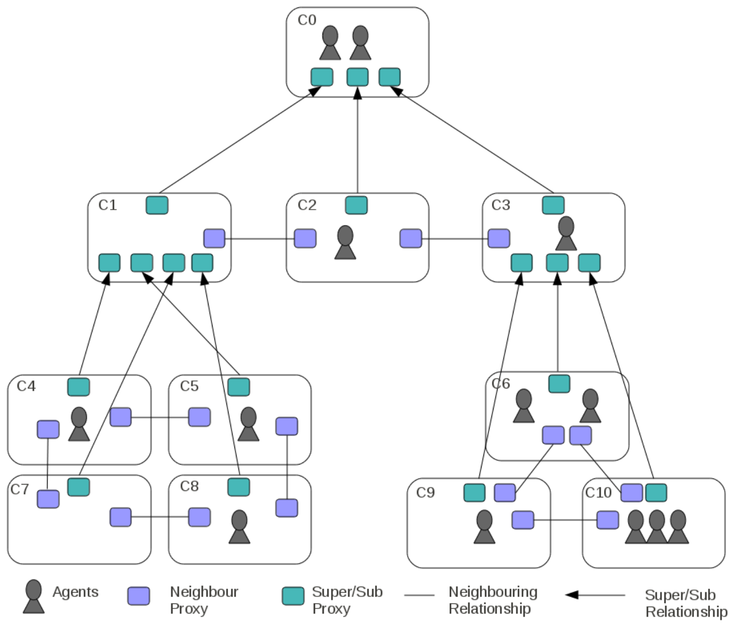


Figure 3.3: Distributed Containers.

In GOLEM neighbour containers and sub/super containers can communicate by means of a proxy object. Usually the communication between two containers is bidirectional. The difference between neighbour containers and sub-containers is that the neighbouring relationship occurs between containers at the same layer which are adjacent in the topology of the agent environment, while in the case of sub-containers these are logically contained in other containers. When containers are organised in multiple hierarchical layers and are composed of neighbours and super/sub-containers, then we say that they form regions to represent the distributed agent environment.

To exemplify the discussion we can think of an agent application where a set of agents are deployed in different buildings across an area. In each building there are many sensors and agents connected to computer machines. The sensors collect some specific data that agents analyse and, if necessary, they can request additional data to sensors or to other agents in the agent environment. In this case, the topology of the application can be deployed as shown in Fig. 3.4. In this example, the developer chooses to have a

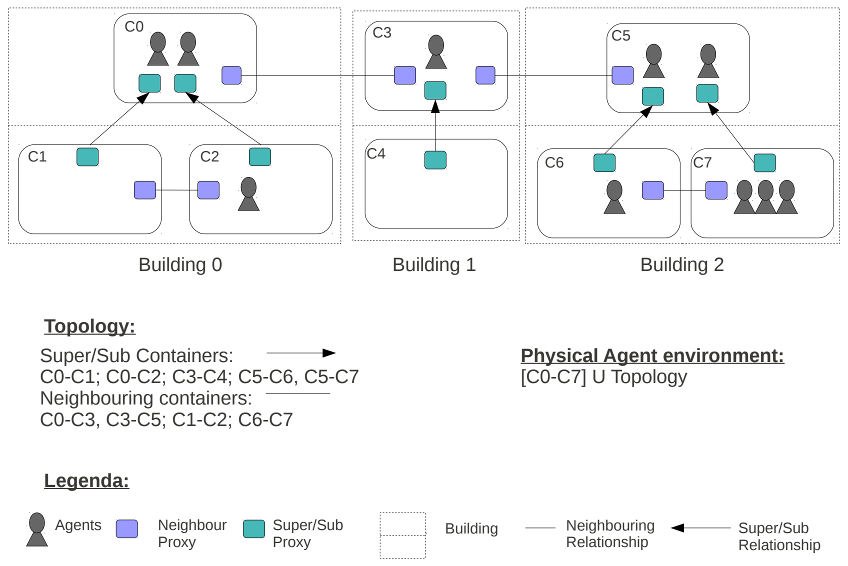


Figure 3.4: Example of how an agent application can be deployed in terms of GOLEM Containers.

container for every machine where agents and sensors are connected (C0-C7). Every building has a super container for all the other containers in the building (C0, C3 and C5 for respectively building 0, 1 and 2). The other containers of the buildings are connected as neighbours (C1-C2 and C6-C7). Finally, the super containers of each building are related to each other as neighbouring containers (C0-C3 and C3-C5) thus, the whole agent environment is connected. Agents within containers can interact in the agent environment independently from their locations. In some applications, it is also useful to define the neighbouring containers in a way that they reflect physical proximity in the real world. For example in Fig. 3.4 C1 and C2 can be containers corresponding to machines located in adjacent rooms. The DPW scenario explained in Section 3.1 can be treated in a similar way. The grid can be partitioned in smaller grids that are contained within GOLEM containers. Whenever these smaller grids are adjacent to another smaller grid, the corresponding containers are connected as neighbouring containers.

3.3 Evolution of the Environment

GOLEM containers are complex objects that contain descriptions of agents, objects and other containers. In section 3.2.3 we outlined how a container can be represented as a set of complex terms. These complex terms evolve over time as agents act on the environment. To define the evolution of the environment in GOLEM we describe the evolution of containers whose states are described as Event Calculus theories.

3.3.1 The Event Calculus

The Event Calculus (EC) was introduced in [77] by Kowalski and Sergot and it is a formalism to specify and reason about actions and their effects. In EC, produced events initiate and terminate properties (called *fluents*). Events happen instantaneously and they are represented in terms of the time when they happen. The axioms below define a simple version of the EC:

Clause EC1 states that a certain property P continues to hold at a particular time T if the system was initialised with that property (at time 0) and nothing happened between then and the time of interest T to change P from holding. Similarly, clause EC2 states that a property holds at a time T if it has been initiated by an event E that happened at time Ts and the holding of that property has not been broken from the starting time Ts and the time of interest T. To decide when a property is broken, we use the clause EC3. This states that a property P is broken between time Ts and T, if another event Estar has happened at a time Tstar whose effect is to terminate P from holding. More sophisticated versions of the Event Calculus are discussed by

```
(EC1) holds_at(P, T)←
              0 ≤ T,
              initially(P),
              not broken(P, 0, T).

(EC2) holds_at(P, T) ←
              happens(E, Ts),
              Ts < T,
              initiates(E, P, Ts),
              not broken(P, Ts, T).

(EC3) broken(P, Ts, T) ←
              happens(Estar, Tstar),
              Ts < Tstar, Tstar < T,
              terminates(Estar, P, Tstar).
```

Shanahan in [110], but these are beyond the scope of this thesis.

3.3.2 Object Event Calculus

To describe how entities change state within a container, GOLEM uses a dialect of the EC called Object Event Calculus. The Object Event Calculus (OEC) was described by Kesim and Sergot in [74] and assumes an object-based data model where instances of objects are represented by unique identifiers and attribute value pairs describing the objects' states at a given time. Given this data model, the OEC formulates how instances of complex terms of this kind evolve over time. OEC is used in GOLEM to describe how entities change state within a GOLEM container.

A subset of the clauses describing the OEC is given below:

Clauses C1-C2 provide the basic formulation of OEC deriving how the value

```
(C1) holds_at(Id, Class, Attr, Val, T)←
        happens(E, Ti), Ti ≤ T,
        initiates(E, Id, Class, Attr, Val),
        not broken(Id, Class, Attr, Val, Ti, T).

(C2) broken(Id, Class, Attr, Val, Ti, Tn)←
        happens(E, Tj), Ti < Tj ≤Tn,
        terminates(E, Id, Class, Attr, Val).

(C3) holds_at(Id, Class, Attr, Val, T)←
        method(Class, Id, Attr, Val, Body),
        solve_at(Body, T).

(C4) attribute_of(Class, X, Type)←
        attribute(Class, X, Type).

(C5) attribute_of(Sub, X, Type)←
        is_a(Sub, Class),
        attribute_of(Class, X, Type).

(C6) instance_of(Id, Class, T)←
        happens(E, Ti), Ti ≤ T,
        assigns(E, Id, Class),
        not removed(Id, Class, Ti, T).

(C7) removed(Id, Class, Ti, Tn)←
        happens(E, Tj), Ti < Tj ≤ Tn,
        destroys(E, Id).

(C8) assigns(E, Id, Class)←
        is_a(Sub, Class),
        assigns(E, Id, Sub).

(C9) terminates(E, Id, Class, Attr, _)←
        attribute_of(Class, Attr, single),
        initiates(E, Id, Class, Attr, _).

(C10) terminates(E, Id, _, Attr, _)←
        destroys(E, Id).

(C11) terminates(E, Id, _, Attr, IdVal)←
        destroys(E, IdVal).
```

of an attribute for a complex term holds at a specific time. Clause C3 describes how to represent derived attributes of objects treated as method calls computed by means of a solve_at/2 meta-interpreter as specified in [75].

C4-C5 support a monotonic inheritance of attributes names for a class limited to the subset relation. In particular, C4 checks if a class has a certain attribute while C5 checks if an attribute belongs to a superclass.

C6-C7 determine how to derive the instance of a class at a specific time. The effects of an event on a class is given by assignment assertions; the clause C8 states how any new instance of a class becomes a new instance of the super-classes.

Finally, deletion of objects is catered for by clauses C9-C11. C9 deletes single valued attributes that have been updated, while C10-C11 delete objects and dangling references.

Event descriptions themselves are specified as complex terms. For example, in the Packet-World, the event description below:

```
do:e14 [actor ⇒ ag1, act ⇒ move:m1 [destination⇒ sq3]]
```

represents a physical action of agent ag1 who tries to move from one location to another. We will see later, how such an action is executed by the agent that causes the event to happen. For the time being, we will assume that the event has happened and we will show next how the affordances of the agent that made the move have changed in the environment as a result of the happening of this event. To do this we need to define domain specific initiates/5 and terminates/5 clauses, as shown below:

```
initiates(E, agent, A, position, Pos) ←
        do:E [actor ⇒ A, act ⇒ move:M [destination⇒ Pos]].
```

In this way the new position of the agent has been initiated as a result of the move. The complete description of the event's effects also requires to terminate the attribute holding the old position of the agent; this is handled in OEC by the general rule described in clause C9.

An example of how the OEC is used in GOLEM can be illustrated using the Packet-World scenario. Suppose that Packet-World is initialised at time 0 so that packet p1 is in square sq1 :

```
happens( assigns:e0[ packet⇒ p1, position ⇒ sq1], 0).
```

Suppose further that the following events happen in the environment as a result of agent ag1 acting on it:

```
happens(do:e1 [actor ⇒ ag1,
                act ⇒ pick:M[packet⇒ p1, position⇒sq1]], 1).
happens(do:e2 [actor ⇒ ag1,
                act ⇒ move:M[position⇒sq2]], 2).
happens(do:e3 [actor ⇒ ag1,
                act ⇒ drop:M[packet⇒ p1, position⇒sq3]], 3).
```

These three events state that at time 1 agent ag1 picks packet p1 from square sq1, at time 2 moves to square sq2 and at time 3 the agent drops packet p1 to square sq3.

To describe this kind of interaction in the environment we need domain specific axioms for the effects of agent actions, such as pick, move, and drop. For example, for the pick action we may specify these effects as:

```
initiates(Ag, agent,holding, PID, T) ←
      do:E [actor ⇒ A, act ⇒ pick:C[packet⇒ PID, position ⇒ Pos]].
```

By describing the rest of the actions similarly, we are now in the position to ask queries about the state of the world and get answers, as shown below:

```
?- holds_at(p1,packet, position, sq1, 1).
Answer: no

?- holds_at(ag1, agent, holding, p1, 1).
Answer: yes.

?- holds_at(ag1,agent, position, sq2, 2).
Answer: yes.
```

3.3.3 The Ambient Event Calculus

The Ambient Event Calculus (AEC) [22] is a formalism that allows the specification of complex events happening in the distributed agent environment represented as a set of interconnected containers.

AEC uses the OEC to query properties of entities within a container. However it extends the OEC with a number of new concepts:

- active perception of events by accessing the environment properties via agent sensors;
- passive perception in terms of notification of events to agent sensors;
- action execution in terms of possible attempts of actions that cause events to happen using agent effectors;
- local queries in a container and all its sub-containers;
- neighbourhood queries over the states of a set of neighbouring containers;
- regional queries over regions of containers including sub-containers, neighbour containers and super containers.

In this thesis we are interested in how an environment composed of containers is queried. To deal with local queries the definition of locally_at/8 below states that the affordances of an entity can be inferred either from the top-level container using clause C18 or from a sub-container using clause C19.

```
(C18) locally_at(C, Path, Path*, Id, Class, Attr, Val, T)←
          holds_at(C, container, entity_of, Id, T),
          holds_at(Id, Class, Attr, Val, T),
          append(Path, [C], Path*).

(C19) locally_at(C, Path, Path*, Id, Class, Attr, Val, T)←
          instance_of(SubC, container, T),
          holds_at(SubC, container, super, C, T),
          append(Path, [C], NewPath),
          locally_at(SubC, NewPath, Path*, Id, Class, Attr,Val,T).
```

Using the above specification we can query whether within the container C exists an object identified as Id, with class Class that has an attribute Attr, whose value is Val at time. In particular, the clause C18 checks whether the object is in the local state of a containerC by using the holds_at/5 to find the object in the container's state. The clause C19 looks for sub-containers. If a new sub-container SubC is found, the same query is asked in the sub-container.

In this way containers can be recursively embedded inside other containers as objects, according to the topology needed, and implemented on different hosts, if necessary. In applications where the topology of the environment requires that containers are next to each other the AEC allows to query neighbouring containers as follows:

```
(C20) neighbouring_at(C, Path, Path*, Max, Id, Cls, Attr, Val, T)←
          Max >= 0,
          locally_at(C, Path, Path*, Id, Cls, Attr, Val, T).
```

```
(C21) neighbouring_at(C, Path, Path*, Max, Id, Cls, Attr, Val, T)←
          holds_at(C, container, neighbour, N, T),
          not member(N, Path),
          Max* is Max -1,
          append(Path, [C], New),
          neighbouring_at(N, New, Path*, Max*, Id, Cls, Attr, Val, T).
```

The rules above specify a query that refers to properties holding in neighbouring containers in the agent environment. Max is the maximum number of containers that form a neighbourhood and Path contains information about which containers have been visited so far with a resulting path Res, while Path* represents the resulting path to the neighbour where the query has succeeded. Clause C20 then looks for the property of the object locally by using the locally_/8 predicates to query in the state of a container C whether an object identified as Id, with class Cls, has an attribute Attr, whose value is Val at time T. Clause C21 looks for the same property in neighbouring containers not already visited. If a new neighbour N is found, this neighbour is asked the query but in the context of a New path and a new Max*.

One of the problems of neighbouring_at/9 as defined before is that when looking at a topology, if the query fails, the super-containers of the container from where we fired the query are not checked. The regionally_at/9 predicate queries big areas of distributed topologies of the agent environment that are referred to as region.

```
(C22) regionally_at(C, Path, Path*, Max, Id, Cls, Attr, Val, T)←
          neighbouring_at(C, Path, Path*, Max, Id, Cls, Attr, Val, T).

(C23) regionally_at(C, Path, Path*, Max, Id, Cls, Attr, Val, T)←
          holds_at(C, container, super, S, T),
          Max* is Max - 1,
          Max* >= 0,
          append(Path, [C], New),
          regionally_at(S, New, Path*, Max*, Id, Cls, Attr, Val, T).
```

Assuming a large Max, clauses C22 and C23 above visit all the nodes of a topology until a solution is found. Complex queries about the attributes of an entity are obtained as a combination of locally_at/8 and neighbouring_at/9 definitions. Similar rules are defined for the generalisation of instance_of/3 that are defined as local_instance_of/6, neighbouring_instance_of/7, and regional_instance_of/7. Their definitions we omit, as these simply rely on calling the predicates of clauses C18...C23 using instance_of/3, instead of holds_at/5. For more details see [19].

3.4 Limitations of GOLEM

The GOLEM agent platform was proposed to address issues related to the concept of agent environment: from the issue of situating agents as physical bodies in a software environment, to the issue of making resources accessible to them by means of objects. The resulting framework, provided an open and distributed agent environment model to mediate the physical interactions amongst agents, objects and containers.

Agents and objects populating the environment are encapsulated into a body which situates these entities in the agent environment. This encapsulation means that GOLEM is open to different entities which are integrated in the system thanks to the body interface. The distribution of the agent environment is another of the features provided by GOLEM. The distribution is defined using the concept of container. Containers maintain distributed portions of the state of the environment and relationship between containers are defined to allow for queries on the state of the agent environment. Agents query the state of the agent environment in order to be able to interact. For this purpose the AEC formalism was proposed by Bromuri and Stathis in [22] to link and mediate the interactions happening in different hosts. Additionally, the distribution of agent environments allows to define systems that can scale up with an increasing number of entities (agents and objects).

The GOLEM framework is a powerful model because allows the definition of agent environment to exhibit the following characteristics:

- enables the deployment of logic-based agents over a network;
- provides a declarative model of agent environment and the representation of the interactions in it;
- mediates interactions within the environment and provides communication mechanisms to send and receive messages;
- proposes a framework and its implementation that supports querying the state of the distributed agent environment.

In general, we want to distinguish the concept of environment into physical environment and social environment. The physical environment defines the physical actions that agents can perform on objects, including the definition of when those actions are possible, and when they are, the effects that they have on the state of the environment. The social environment on the other hand, defines the rules that the agents share to communicate in a common language and coordinate their actions to achieve their goals. It also describes the roles that the agents play in the interactions and their relation to other agents [91].

In applications such as ARGUGRID [3], in order to support the activities of the agents, there is a need to provide both concepts of the physical and the social agent environment. ARGUGRID agents need to collaborate and create agreements which goes beyond the concept of physical possibility offered in GOLEM. For example, a user requests an ARGURID agent to find a specific service in the platform. The agent interacts with other agents that offer the service and creates an agreement on behalf of the user. The physical environment enables the agent to discover service providers that offer the requested service but, it cannot deal with the negotiation protocols that are involved when the agent attempts to reach an agreement over s service. The interaction here requires the involved agents to take specific roles, which in turn make the agents have rights and duties that constrain further the possible actions that the environment can support. As a result, negotiation protocols ala ARGUGRID need a higher social-level that complements the physical one in order to support the required interactions.

To summarise, the development of GOLEM aimed at modeling the physical aspects of an agent environment, but it was not intended to support interactions of a social agent environment. Despite the advantages that it offers as a MAS platform for developing distribution of agents over a network, including their low-level interactions, it does not provide mechanisms for representing the responsibilities of agents according to the roles they play in agent interaction protocols. More specifically, GOLEM does not support normative notions for modeling the social interaction protocols, the communicative actions and their effects in the system. As a result, interactions in GOLEM cannot be defined in terms of are permitted/obligated actions, their validity, and their coordination in more complex and dynamic social settings.

Motivated by these limitations, this thesis seeks to study how to introduce the key aspects of a social environment as part of the GOLEM framework so that it can be used for applications that require social interaction amongst agents. In the following chapters we incorporate in a systematic way the concept of social environment into GOLEM, without interfering with the GOLEM approach.

3.5 Summary

We have presented the GOLEM platform and its building blocks: namely containers, agents and objects. We described how GOLEM uses the concept of affordances to describe entities of an agent environment in terms of what can be perceived of an entity situated in such environment. GOLEM objects are seen as reactive entities used to wrap resources of the external environment. Agents, on the other hand, are defined as cognitive and active entities that interact with other agents and objects in containers. A container is used to represent a portion of the distributed agent environment and can be composed to build more complex configurations.

We then showed how the agent environment can evolve in time and this evolution is captured in the state of every container using the OEC. We also explained how by connecting the containers in a logical structure and using the AEC as the underlying representation mechanism it is possible to query the state of the whole social environment. In fact, agents can query the state of the environment and perceive the affordances of other entities in the environment.

Finally we showed that despite the many advantages of the GOLEM platform, the system does not support social aspects of the agent environment. The current platform only supports the enforcement of rules describing how agents can physically act in the environment. Thus, in the current GOLEM model, it is not possible to model agent to agent interactions other than at a low, physical level.

In the next chapter we introduce the MAGE framework. We present the idea behind MAGE and how we model simple social interactions (protocols) as atomic games. The interactions within games evolve in time and stores the changes to the state of the interaction due to agents making moves within the atomic game. In MAGE agents are enabled to play based on their preferences and strategies.

Chapter 4

Atomic Social Containers

Motivated by the need to model the social interactions of an agent environment and the need to extend the GOLEM platform with social aspects of agent interaction, in this chapter we introduce the Multi-Agent Game Environment (MAGE) framework. MAGE extends the GOLEM container with the notion of social container, thus dividing containers into physical and social. In this way, the agent environment is engineered to separate the social rules of the application from the physical ones [11].

Given the separation between physical and social containers, this chapter studies how to represent interactions in social containers using games framework of Stathis [112]. More precisely the work studies how the idea of atomic games in [113], extended here with a normative interpretation of valid moves, can be used to formulate interactions of the most basic social containers. In MAGE we refer to these containers as atomic social containers. To exemplify the interactions in these type of containers, we are going to use the Open Packet World scenario.

The chapter is organised as follows. In Section 4.1 we present the Open Packet World scenario which is an extended version of the Packet-World scenario presented in [130]. In Section 4.2 we explain the concept of *social* *container* and how this entity relates to the rest of the framework. In Section 4.3 we show how the concept of *game* can be used to define *atomic social containers*. In Section 4.4 we formulate the rules of an atomic game to show how to capture the evolution of the state of the game and how to model the valid acts that agents can perform within it. Finally, in Section 4.5 we summarise and conclude this chapter.

4.1 Open-Packet-World Scenario

For the purpose of this thesis we extend the Packet-World (PW) scenario [130] and its distributed version (presented in Chapter 3) with a variation that makes the world open and competitive. We call this extended version the Open Packet World (OPW).

In the OPW, agents compete to maximise the number of packets they deliver to a destination. We make the agents compete by rewarding them points whenever they deliver packets to appropriate destinations. Agents are now antagonistic and may be developed by different parties. They can reason based on their intentions, strategies and internal preferences. Here, we assume that agents share an ontology which is necessary for both social and physical interaction. Unlike the PW, in this work we are making no assumptions on what types of agents and what reasoning they perform when they choose to act in the environment. Agents can be developed to be norm-unaware in that agents may inadvertently violate norms. Alternatively, agents can be norm-aware in that they may be norm-abiding and comply with the social rules, or 'opportunistic' in that they may violate a norm if, according to the agent's reasoning, the sanction is less 'costly' than the utility gained by violating the norm. For instance an agent may try to deceive other agents by placing a flag in an area that already has packets. As a result of agents being competitive and possibly selfish, we want to encourage agents to follow rules, so we introduce norms. Violation of norms results in sanctions. One type of sanction, in this example, is the reduction of the points that an agent has due to violated norms. These aspects of the OPW scenario have already been discussed in [124, 125].

The Open Packet World (OPW) presents a number of knowledge representation challenges for a norm-governed system. Unlike other practical applications, e.g. electronic markets, it requires both speech acts for collaboration amongst agents but also the simulation of physical actions, which in turn necessitates the representation of physical possibility in the system. Physical possibility requires the representation of a physical environment whose state should be distinct from the state of the social environment. The OPW is also convenient from the point of view of experimentation in that we can make the experimental conditions harder by increasing the size of the grid, the number of agents and the number of packets/destinations.

4.2 Social Environment as Social Containers

We extend the agent environment as discussed in Bromuri's work [19] with the notion of social environment that mediates the social interaction of the agents. In order to model a social environment we introduce the concept of *social container* as opposed to Bromuri's *physical containers.* A *physical container* in Bromuri's work provides strong mediation in that agents situated in it can perform actions successfully only if these actions are physically possible within the container. For example, in the PW, an action of an agent moving in a position where there is another agent will fail. *Social containers*, however, provide a different, weaker concept of mediation. An agent is not physically situated in a social container. Instead, agents relate to a social container logically, through membership. When an agent acts in the physical container in which it is situated, this action is then governed by the rules specified in the social container the agent is a member of. However, an action that is invalid in the social container does not imply that the action will fail, rather that it will probably be sanctioned. For example, in the OPW an agent can flag a region as being explored, even if it is not true; the social container will allow this to happen even if the act is invalid (although the sanctioning mechanism of the social container will eventually punish the agent as it has violated the rules).

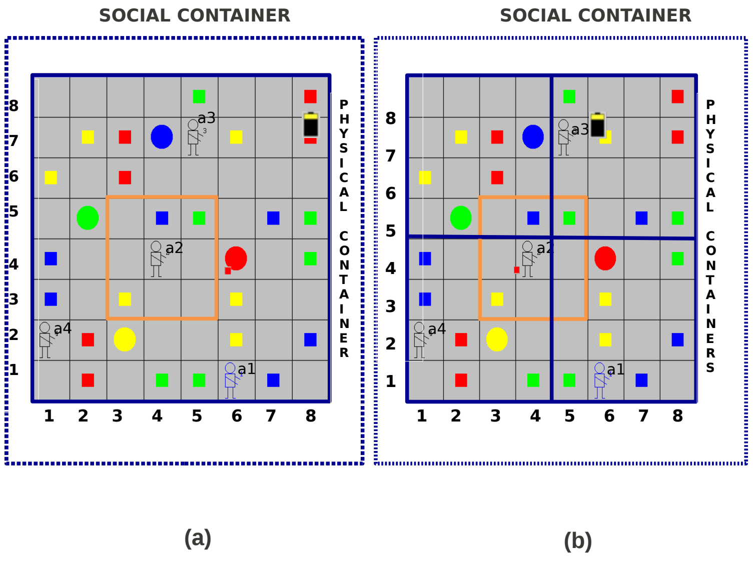


Figure 4.1: The Open Packet-World using Social Containers.

A *social container* is an entity that encapsulates the social rules of the agent environment. The rules of the social container describe how the social state changes as a result of agents undertaking interactions amongst each other. Following our characterisation of social environments in Chapter 2, we view every *social container* to contain:

- a set of member agents;
- a set of social rules;

- a social state;
- a link to the physical environment.

The agents interacting in a social container are members of the social container. The social rules capture how interactions should evolve between interacting agents. The social state is updated based on the social rules and the history of the interaction within the container. The physical environment is linked to the social environment by defining sub-super relationships between social containers and the physical ones, in this way, the social and the physical state are linked together with the AEC.

The link between the physical and the social environment is defined so that the social rules of the social containers can govern acts happening in the physical environment (where the agents are situated). For example, to represent the link between physical and social containers in the OPW scenario, we can represent a social container as a super-container of all the physical containers that are defined in the agent environment. In general, how the social and physical environment are distributed depends on the scale of the application. In OPW, if the number of agents and packets is small, we could have one physical container mediating all the physical interaction and one social container as a super container of the physical container mediating the social interactions of the participating agents.

To exemplify the discussion, in Fig. 4.1 (a) we show a case were a social container is a super-container of one physical container. The figure represents social containers with dotted squares and physical containers with full-line squares. The relationship between containers is represented by including the squares into another square whenever there is a super-sub container relationship amongst containers and by drawing squares next to each other whenever they are neighbour containers.

In a case where the number of entities is big and we have a few social rules, we can distribute the environment in many physical containers but mediate the social interaction from one single social container. We show this representation in figure 4.1 (b) where we show a case where the social container is a super container of four physical ones. There are more complex applications where the concentration of the social rules all in one container might create a bottleneck for the application. We are going to discuss more complex representation of the social environment in the next chapter.

4.3 Social Containers as Games

We use the game metaphor [115, 113, 114] for formulating the interactions amongst agents in a social container. The games metaphor, construes communicative interactions within an agent society abstractly as game interactions [113]. We focus on regulated interaction which produces a result as in dialogues games [80]. The result can be a win/win situation or a draw situation, not necessarily 0-sum [90] where there is always a winner or a looser.

Agents become players in a game and can perceive how the state of the game changes. The game is separated and updated independently from the agent life-cycles, thus, at any time the players of a game have a consistent and shared view of the state of the game. Therefore the game can be seen as a shared memory between the interacting agents. A game that is contained within a social container of the agent environment is perceivable by the players in different locations and allows for the physical distribution of these components.

4.3.1 Link between Social Containers and Games

Using a game metaphor, we can specify the social rules of a social container as rules of games that are played amongst players. The state of a game has (a) a set of properties that are part of the physical environment where the game is played and (b) an additional description that provides a social interpretation of the agent interactions represented separately, as if they were giving meaning to some of the evolution taking place in the physical environment. The state of the game is contained in the social container and is updated as consequence of the moves of players. Agents are the players and the moves they perform are interpreted as events happening in the game state. The roles that agents assume as players determine what valid moves a player can perform. We say that an agent complies to the rules of the social containers if the moves made by this agent are valid game moves.

Within a social container, the rules of a game determine if agents' moves are valid as the interaction progresses. If an agent move is valid, both the physical and social parts of a game's state change according to the prescribed effects the move has on these parts. If the move is not valid, however, it means that the physical state has changed in a way that it is not meaningful (in the sense that the rules of the game have been violated) to carry on playing the game. In this case, we need to change the social part of the state in such a way so that the changes in the social part can support the implementation of the game to bring the physical state at a point where agents can continue playing the game. Often this means that the implementation of the game will need to rearrange the physical state or enforce possible sanctions that the rules of the game may prescribe so that the agents that have violated the rules are punished, so that the game can continue.

4.3.2 Atomic Social Containers

In MAGE, we represent the most basic social containers with atomic games. These games are defined to not contain properties that are games themselves [112]. We refer to these kind of containers as *atomic social containers*. Because of the way these containers are represented in terms of atomic games, they cannot contain social sub-containers, only physical ones.

Fig. 4.2 shows a simple architecture for the social environment. In this

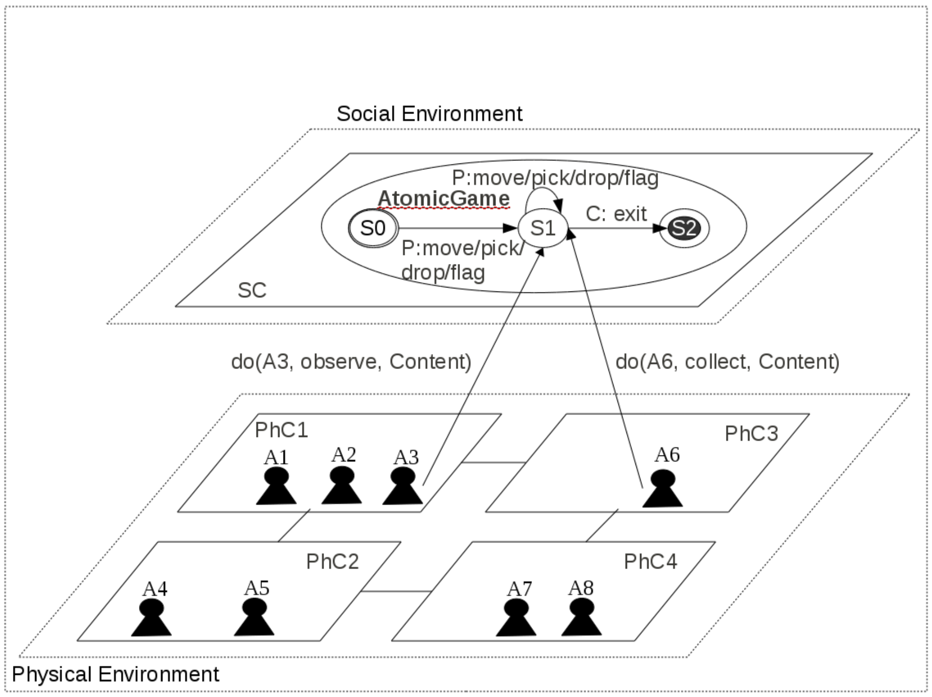


Figure 4.2: A Social Environment as an Atomic Social Container.

example, the social environment is defined as one atomic social container. The physical containers are denoted as PhC_i while the social container is denoted as Sc. The figure shows agents (denoted as A_i) from different physical containers performing moves in the Atomic game contained within the social container. The Atomic game in this case is the OPW game that defines a set of social rules for when the agents perform the acts (pick, drop, move and flag) in the environment. These moves, change the state of the physical environment as well as the social state.

With a move action an agent changes its position in the grid, with a pick action an agent picks a packet from the grid, with a drop action an agent drops packets in the grid, and with a flag action the agent 'announces' that the nearby cells have no packets. These acts can change physical properties of the environment (i.e. in the agent moves, it has a new position in the physical space). To regulate the physical interactions a set of physical rules define what is possible and impossible in the environment. For example, it is impossible for an agent to move more than one square at a time or it is impossible for an agent to pick more than a packet at the same time. Additionally to the physical rules, the social rules specify how the agent is expected to interact from the social perspective.

On the other hand, the OPW game defines the social rules such as it is forbidden for an agent to move too close (adjacent) to another agent, it is forbidden to drop packets anywhere in the grid that it is not a destination point or it is forbidden to flag an area with one or more packets. The main changes in the social state, in this case, happens due to reward policies that apply as a result of agents following or not the social rules. Sanctions are defined to remove points from agents that perform forbidden actions. In particular, we remove 1 point for moving adjacent to another agent, 2 points for dropping packets outside the destination points and 10 points for flagging an area that has packets. Within this game, we also reward 10 points to picker agents that drop a packet in the right destinations.

4.4 Representation of Atomic Games

To formulate an atomic social container as an atomic game we need to decide how to represent the atomic game. In particular, we need to describe the state of the atomic game, its initiating and terminating states, how players make valid moves, and how the effects of these moves change the current state to the next one until the terminating state is reached. We describe these next. The representation that follows assumes the game/2 definition presented in Chapter 2, Section 2.5.5.

4.4.1 The State of Atomic Games

To represent the state of an atomic game we use C-Logic [31] terms identified by a unique identifier describing the attributes of the state's configuration. The rationale behind this kind of representation is that in MAGE we acknowledge the fact that the interaction within a multi-agent system application can become quite complex. To cater for the complexities of practical applications we assume that complex terms have an underlying object-based data-model. Such a state representation will evolve over time as a result of players making moves.

In order to give an example of an atomic game, lets consider the term representing the state of the OPW presented in Sec. 4.1 as an atomic game:

```
opw_game_state:opw1[
    physical_state⇒ packet_world:c1,
    members⇒ {agent:a1 [role ⇒ picker], agent:a2 [role⇒picker]},
    sanctions⇒ {sanction:s1 [agent ⇒ a2, ticket ⇒ 5]},
    result ⇒ nil,
    sub_process ⇒ nil
].
```

The identifier opw1 denotes an instance of an object whose class is the main with one attribute physical_state that refers to the state of the physical environment, an attribute members containing the agents participating in the game and their role, an attribute sanction that identifies agent a2 as sanctioned with a ticket for 5 points, an attribute result which has value nil, and an attribute of class sub_process which states that it has value nil. The last two attributes respectively mean that this particular game has not terminated yet as it has no result and it has no sub-games therefore it is an atomic one.

We will refer to the state of the game using a term of the form G@T to denote the complex term representing the state of the atomic game identified as G. The complex term describes the attributes of the state's configuration for the game G. T is the system's time which uniquely identifies the actual evolutions of the complex term as a result of the interaction.

4.4.2 Valid Moves

The moves of a game are represented by complex terms. The complex term below:

speech_act:m1[actor ⇒ a1, act⇒move, square ⇒ sq1, role⇒ picker],

describes that a picker agent a1 utters a move action to describe its intent to move to the square sq1. Such moves are used as to define the events that happen at a specific time. An assertion of the form happens(m1, 12), states that move m1 has happened at time 12. Such an event changes the state of the game. We then use the Object Event Calculus (OEC) for queries that are local to the social container and we use the Ambient Event Calculus (AEC) for querying the physical state. Both AEC and OEC were presented in Chapter 3.

Before the event of a move being made in the state of the game, we must have a way to check if a move is valid. We check that actions performed by agents are valid with the following specification:

```
(SR1) valid(G@T, E)↔legal(G@T, E).

(SR2) legal(G@T,E) ←
          obligatory(G@T, E) ∨
          (not obligatory(G@T, E),
          permitted(G@T, E)).
```

The clause SR1 defines that an event E is a valid event in G@T if and only if it is a legal event in G@T. The clause SR2 defines an event E as legal if it is either obligatory or not obligatory move but permitted.[1]

Permission, prohibition, obligation and empowerment are specilised rules that differ from one game to another. To specify permission, prohibition, obligation and empowerment we specify when in the state of a game a move is permitted or forbidden or when an agent is empowered or obliged to take an action. For example, to specify permitted actions in the OPW game we define the following rule:

```
(OPW1) permitted(G@T, Action)←
            empowered(G@T, Action),
            not forbidden(G@T, Action).
```

The OPW1 rule specifies that in an OPW game, the act Action is permitted in G@T if it is an empowered and not forbidden act in the state of the OPW game G@T. Other definitions of permitted/2 are possible, for example it is possible to define that an act (i.e exit) is permitted at a certain state of a game, without need to additionally define that the agent is empowered to perform such act [2]. Forbidden actions in the OPW game are specified as

[1] In our definition of legal/2 we do not check if the action is empowered. When defining legal moves, the permission to make an act and the concept of institutionalised power seem very close concepts that makes it difficult to distinguish between the two. However as argued by Jones and Sergot these two notions are not equivalent [72]: agents may be permitted to make an action without being empowered to perform this action (i.e. a priest is permitted to marry a couple that requested to be married but it might not have the power to do so if the couple has not catholic religion) and also agents might be empowered to perform an action but not be permitted to do so (i.e. a priest can marry a couple but it might not be permitted to marry them without both of them having requested to be married). This distinctions between the concept of permission and empowerment lead us to suggest that in order for an action to be legal it has to be permitted and empowered. Often however, it is unpractical to list all the empowered actions in the system thereby, in some cases, we can associate the institutionalised power to the permissions (and obligations) [11].

[2] An example of such specification can be found in Chapter 7

forbidden/2 rules. The rule:

```
(OPW2) forbidden(G@T, Action) ←
            do:Action[actor ⇒ A, act⇒drop, flag ⇒ PosA],
            neighbouring_at(this, [], _, 1, Object, Class, position, PosB, T),
            adjacent(PosA, PosB),
            (Class=destination; Class=packet).
```

specifies that it is forbidden for an agent A to drop a flag in position PosA if there are destinations or packets nearby.

Similarly to prohibition and permission, we can also represent a basic form of empowerment. For example, we can express the fact that every agent in a picker role is empowered to perform the acts move, pick, drop, flag in the OPW game. We express this rule as follows:

```
(OPW3) empowered(G@T,Action)←
            do:Action[actor ⇒ A, act⇒Label, content ⇒ C],
            member(Label,[move, pick, drop, flag]),
            holds_at(A, role, picker, T).
```

The above OPW3 clause states that an agent A, is empowered to perform an act of type move, pick, drop, flag if this agents holds the role of picker. In a similar way we can define obligations. For example, in OPW, we can express that agents that have collected a packet should deliver it to a destination with the same colour. Such obligation is defined as follows:

```
(OPW4) obliged(G@T, Action)←
            do:Action[actor ⇒ A, act⇒drop, obj⇒ObjId, destination⇒ Dest],
            neighbouring_at(this, [], Path, 1, A, agent, holding, ObjId, T),
            neighbouring_at(this, [], Path, 1, ObjId, packet, colour, C, T),
            neighbouring_instance_of(this,[], Path, 1, Dest, destination,T),
        neighbouring_at(this, [], Path, 1, Dest, destination, colour, C, T).
```

The clause above states that an agent A is obliged in G@T to drop a packet identified as ObjId to a destination Dest if the agent A is holding the packet and the packet has a colour C which is the same with the colour of the destination. We use neighbouring_at/9 and neighbouring_instance_of/7 AEC predicates to query the information hold in the physical state of the environment. The container holding this information will be the physical container where the agent A is situated. For our scenario, such rules provide a basic form of obligations that implicitly persist until they are fulfilled. More complex application scenarios will require more sophisticated treatment of obligations. However, this discussion is beyond the scope of this work.

The fulfillment of the obligations is checked using the fulfilled/3 predicate. If the obligation is not fulfilled, the happens predicate generates a violation event which will sanction the agent that did not satisfy the obligation. The fulfilled/3 predicate and the generation of a violation is specified as follows:

```
fulfilled(G@$T_j$, E, T)←
  obliged(G@$T_j$, E),
  happens(E, T),
  $T_j$ ≤T.

happens(violation:E*, T)←
  now(T),
  this(G),
  $T_j$ is T-120,
  not fulfilled(G@$T_j$, E, T).
```

The first predicate defines that an event E is fulfilled at time T if the event E happens at a time T and it was obliged at a time T_j, with T_j being less or equal to T. The second predicate, determines that a violation event E happens at a time T if there has been an unfulfilled obligation in the system for more than 120 seconds.

4.4.3 Effects of Moves

Once a move has been determined as valid, a new state of the game must be brought about due to the effects of the move. As by making moves players cause events to happen, if we assume that the happening of such moves take only one unit of time, we can specify their effects as:

```
effects(G@T, Move, G@NewT) ←
  add(happens(Move, T)),
  NewT is T + 1.
```

In our representation of state, once an event has happened, its effects are added to the state implicitly, via inititiates/4 definitions that initiate new values for attributes of a state term, terminates/4 clauses that remove attribute values from a state term, and assigns/3 definitions for assigning to objects identified as G new instances of terms. An example, of how new values are initiated for attributes for the OPW game is given below:

```
initiates(E, A, role, picker)←
   happens(E, T),
   do:E[actor ⇒ A, act ⇒ Label, content ⇒ C],
   member(Act,[move, pick, drop, flag]),
   not holds_at(A, role, E, T).
```

The above definition initiates the attribute role of an agent A to be stored in the state of the game as a picker if the same agent performs one of the default moves of the OPW game and it does not hold a role in this game.

4.4.4 Enforcement Mechanisms

We can detect when agents perform invalid moves. At design time, depending on the type of application, it is possible to decide whether regimentation mechanisms are more appropriate than enforcement mechanisms. In our OPW scenario we found that defining an enforcement mechanism was more realistic and flexible as it allowed to test the application with different reasoning mechanisms including cheating agents.

In [68] the authors foresee agents that intervene to decide on the type of enforcement in case of violations. It is possible to design games with additional umpire or referee agents that decide how to sanction violations, however, if the application domain requires general sanction/reward mechanisms (i.e. agents that perform forbidden actions should be punished by detracting points from the agent who performed the action), these can become part of the game infrastructure methods with no need of defining external observers for every game that is played in the distributed environment. This has two advantages: first it simplifies the definition of the framework and secondly it provides a feedback mechanism for agents as they can inspect what happens in the environment but they cannot do the same with autonomous agents which have their own reasoning process.

In MAGE we define the following sanction mechanism: an agent is sanctioned when it performs an invalid act. In order to keep a coherent view of the points that every agent has, independently from the interactions it is performing, we store these points as part of the social container. When a violation is detected, a new event is created in the social containers. In turn this produces a sanction event to sanction the agent who performed the forbidden action. The rule below creates a sanction object when an agent performs an invalid action.

```
happens(E,T)←
    happens(E*, T),
    E*[actor⇒A, act⇒ Act, content⇒ Cont],
    not valid(E*,T),
    points_for(Act, Points),
    new(violation:E[sanction: S[points⇒ 5, agent⇒ A]).
```

The container manages the sanctions as follows:

```
assigns(violation:E [sanction:SID], SID, sanction).

initiates(E, A, points, Points)←
   happens(E,T),
   violation:E[sanction:S [points⇒ Sanctions, agent ⇒ A]],
   holds_at(A, agent, points, OldPoints, T),
   Points = OldPoints - Sanctions.
```

The assigns/3 statement above associates a class to a certain object identifier, given an initialisation event, while the initiates/4 statement updates the points attribute of the agent A as a consequence of receiving a sanction S at time T. The rationale behind the creation of a new event is to capture the fact that an event happened in a specific game of the environment has social implications for which we need to create a social structure (a sanction in this case) where it can be visible to all the members of the society.

In OPW we also want to reward some of the agents actions as a way to promote good or utilitarian behaviour from the system's perspective. We handle rewards in the same way as we do for the violations, with the difference that we give points to agents performing certain acts (i.e. in OPW agents gain points for dropping packets to the right destinations).

4.4.5 Initial and final states of a game

For the state of an atomic game to be created, the framework discussed so far requires the assertion of an event that will first create the term via an assigns/3 assertion. The assertion:

```
assigns(E, G, opw)←
    do:E[act ⇒ construct, protocol ⇒ opw, id ⇒ G].
```

will allow the creation of an instance for the OPW game, which can then be queried using the AEC that we described in Chapter 3. To complete the instantiation process we also need to specify the initial values for the attributes of the complex term representing the social state of the OPW game. For this we need to define separately the initiates/4 rules as the one below:

```
initiates(E, G, result, nil)←
   do:E[act ⇒ construct, protocol ⇒ opw].
```

Additional initiates/4 clauses are needed to define the whole of the initial state, one for each attribute value.

The initial state of the game will evolve as a result of moves being made in the state of a game. This state will eventually reach the final state from which we can extract the game's result. We specify this via terminating/2 predicates. For example, the definition:

```
terminating(G@T, Result)←
      instance_of(G, opw, T),
      not hold_at(G, result, nil, T).
```

specifies the conditions under which the OPW game terminates and at the same time returns the result.

4.5 Summary

In this chapter we presented MAGE as a logic based framework that models the interactions of agents with atomic games. The MAGE framework has been already partially discussed in [26, 24, 126, 124, 125]. We introduced an extended Packet World Scenario which we call the Open-Packet-World scenario to make the original scenario more competitive and to be able to show how normative aspects can be included as part of the social environment to regulate agent interactions.

The architecture of MAGE is based on the concept of social container, which is an entity distributed over the network that mediates the social interactions between agents, allowing them to create new collaborations and interpreting their moves using social rules.

We introduced atomic social containers and we showed how to model these with atomic games. We illustrated how we model an atomic game in terms of a state that evolves in time as agents perform valid acts within the game. We have showed a general model for defining atomic games and we have illustrated how to apply it to the OPW scenario.

In the next chapter we are going to show how we can use many social containers in order to distribute the computation in a social environment and how we can coordinate atomic games within complex games. We will define coordination patterns to describe which atomic games will be running at a given time T and at a given state of the interaction.

Chapter 5

Complex Social Containers

In the previous chapter we showed how social environments could be seen as social containers that augmented the physical ones. We defined social containers as games where the state of the container is the state of the game, the rules that govern the agent interactions are the rules of the game, the agents are the players, the actions are the moves and the outcome is the result of the game. The basic component we defined was the atomic game that captures simple indivisible interactions (such as protocols). The atomic game evolves as players make moves in order to change its state, possibly to their advantage.

In this chapter we discuss the distribution of a social environment on top of a physical one. We use complex social containers to link the games and the agent environment in a complex distributed structure that can be inspected by agents and evolve as agents act on it. We specify the notion of meta-game as a mechanism responsible to handle the creation of new interactions amongst agents in the social environment. The distribution allows us to execute the games and compute normative relations at run time. We define compound games from simpler, possibly atomic games and show how to relate the physical and the social structure so that they can evolve in parallel.

The reminder of the chapter is organised as follows: Section 5.1 presents the driving scenario for this chapter. Section 5.2 identifies the need to distribute the social environment as well as to coordinate complex interactions and shows how this can be achieved in MAGE by using complex social containers. Section 5.3 defines a complex social container in terms of a compound game. Section 5.4 shows how the structure of the social environment can be changed by defining a meta-game that manages these changes. Section 5.5 defines the coordination mechanisms required to create compound games. Section 5.6 defines the forwarding mechanism as a way to link the social and physical containers by forwarding the actions of the agents into the games of the social environment. Finally, Section 5.7 summarises and concludes this chapter.

5.1 VOs in the Open Packet World Scenario

The version of OPW presented here, differs from that of Chapter 4 in that it allows agents to be part of more organised competitions and form Virual Organisations (VOs) [83]. Informally, a VO is a mechanism where a new collaboration is created between two or more agents wishing to achieve a common goal. The collaboration assigns each agent participating in the VO with responsibilities in terms of roles and enables these agents to achieve the common goal. In our OPW version, agents form VOs to collaborate with other agents to deliver packets. The possible roles in such VOs are the leader role which leads a set of agents to collect packets or a picker role which can pick packets and bring them to a destination.

A VO collaboration goes through a process of formation, execution and dissolution. In the formation phase of a VO, a set of agents (members of the VO) decide to form a VO with a specific goal. By joining the VO, each member is assigned a role which determines its future interactions within the VO.

We empower agents with a high number of points (i.e. $\geq$50 points) by allowing them to form VOs and be leaders. Giving agents the possibility of becoming a leader is an indication of encouraging good behaviour within the environment and ability in collecting packets. In addition, the VO goal can be quite specific (i.e. collect all red packets). The invited agents can agree or disagree to join the VO. An agent agrees to join a VO and becomes a picker agent if, by collaborating with other agents members of a VO, it will gain better knowledge of where the packets are and consequently gain more points due to collecting more packets as a team. A VO is formed if at least two agents agree to take part in it. Once the formation is agreed, the members of the VO can start collaborating to collect packets.

The execution of the VO specifies the rules on how the members are expected to act within the VO. To deal with the execution of the VO, the norms that specify the powers, permissions and obligations of the agents are based on the roles that the members hold in the VO and the agreements amongst members during the VO formation (i.e. collect red packets). In other words, the norms express how different normative rules apply to the members depending on their role in the VO.

An agent who started the formation of the VO takes a leader role in the system. The leader of the VO is empowered to request agents that are looking for packets to collect a packet by indicating where it is located. It is also empowered to dissolve the VO if the number of members in the VO drops to 1 or if the goal of the VO has been achieved. If the leader is coordinating well, the number of points increases, in which case it is permitted to invite more members to join the VO.

Agents who join a VO become pickers of packets of a specific colour. An agent holding a picker role agrees to collect pro-actively packets of the agreed colour and coordinate with the other VO members for the collection of these packets. To coordinate this process, agents in picker roles are permitted to report to the leader about packets they observe but do not collect (i.e. because it is doing other actions). A picker agent is also obliged to contact the leader when it has delivered a packet and it is looking for more packets and to fulfill the request of the leader to collect a specific packet. If the request of the leader is unfulfilled by the picker, the agent is sanctioned.

Agents are rewarded points (i.e. 10 points) to deliver packets into the destination. When an agent joins a VO, the packets that are delivered by a picker gives some of these points to the picker (i.e. 8 points) and some to the leader (i.e. 2 points). This means that if agents are not members of a VO, they get more points to deliver a single packet. However, in the long term, not having a group of agents that help the coordination amongst them, also means that agents need to search more within the world for finding a packet. This is why, from the agent perspective, being part of an organised VO can be more profitable than working alone.

5.2 Large Social Environments as Complex Social Containers

Large scale multi-agent applications are typically characterised by many distributed processes requiring a high number of computational resources. In such applications, there is a possibly a large number of agents giving rise to different interactions. Thus, it is not feasible to design a system where the social rules are centralised and mediate all the interactions from a single social container (as we illustrated in the previous chapter). This would create an inevitable bottle neck in the system that would make the interactions more difficult rather than facilitate them. This is why, for large scale agent applications, we seek to distribute the computation by extending MAGE in such a way that social rules are distributed amongst many social containers.

Fig. 5.1 exemplifies how we can decentralise the social environment of OPW with three social containers [1]. This setting shows the distribution of the

[1]Due to space limitations, the figure is not showing a large scale application, but a

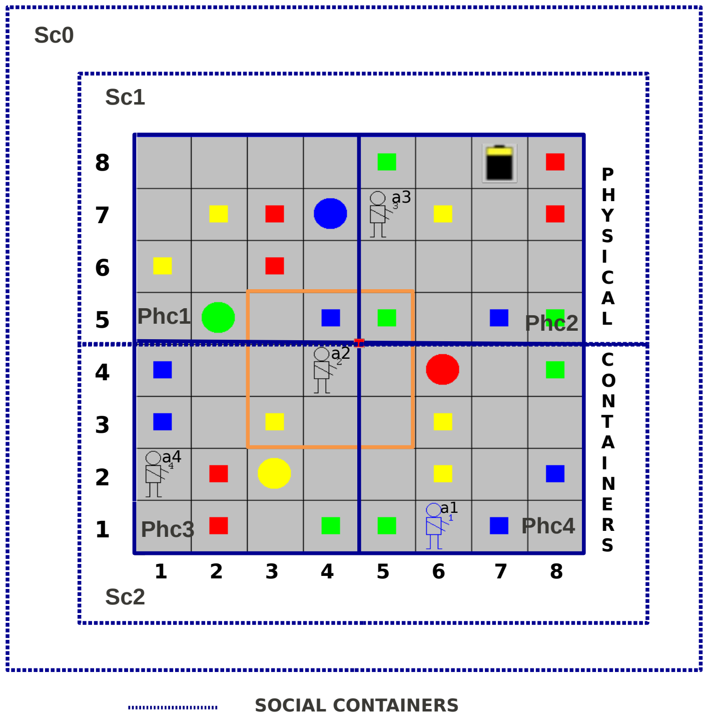


Figure 5.1: OPW regulated by distributed social containers.

social environment in many containers to achieve parallel computation of the social rules. The social containers are represented with a dotted line, while the physical containers are represented with a full line. The physical containers divide the OPW grid into four parts that relate to one another as neighbouring containers. The figure also shows the social containers, containing the physical containers. This is to represent the fact that interactions of the agents situated in the physical containers Phc1 and Phc2 are being mediated by the social container Sc1 and the social interactions of the agents situated in the physical containers Phc3 and Phc4 are being mediated by the

rather small one with only an 8x8 grid with four agents in it. However, the distribution of a bigger grid (i.e. 1000x1000 with more than 1000 agents), would follow the same principles.

container Sc2. To maintain a relationship between the social containers, Sc1 and Sc2 are contained in a super social container Sc0 (or root). The final structure of the social environment is then connected to the structure of the physical environment. Acts produced by agents in the physical containers Phc1 and Phc2 need to be forwarded to the social container Sc1 to evaluate social implications, if any, and update the social state. Similarly the physical containers Phc3 and Phc4 can forward the acts of the agents in the social container Sc2.

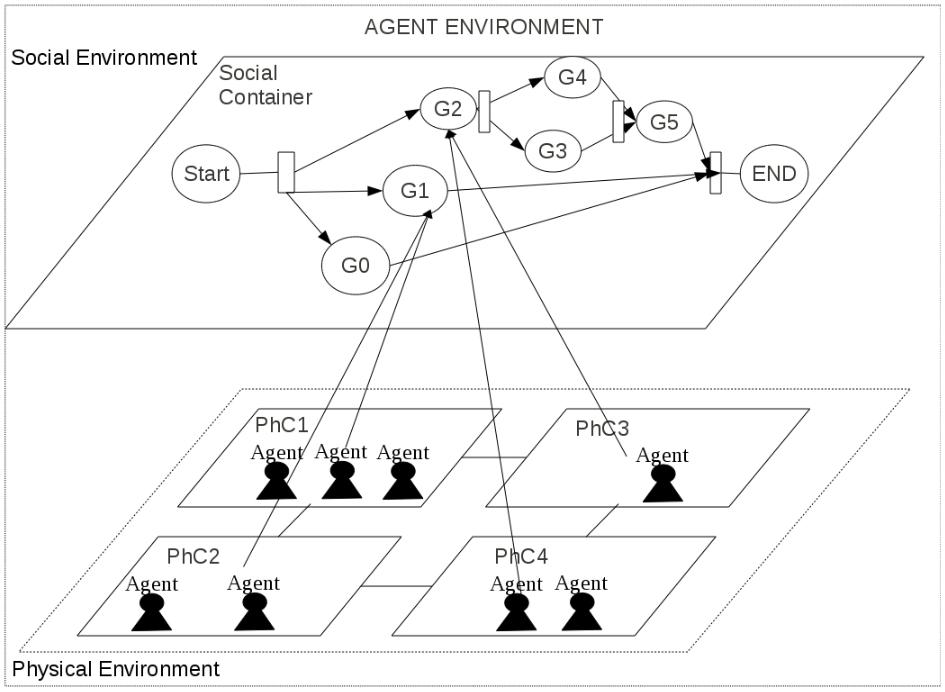


Figure 5.2: Social Container mediating sub-activities of agent interaction.

Another issue to consider in a large social environment is that the social activities in a MAS application can be complex and may need to be structured in terms of dynamic sub-activities. For example, in the OPW the agents create VOs to coordinate amongst each other in the collection of the packets as presented in Section 5.1. Fig. 5.2 shows the social environment as a social container containing a complex activity composed using various atomic games. Within this complex activity, agents can interact in one or more atomic games at the same time. In Fig. 5.2 the physical containers are denoted as PhC_i while the social container is containing many atomic games, denoted as G_i. The figure shows agents from different physical containers playing in different atomic games that are part of a more complex activity. We define a social container able to contain and coordinate many atomic games as a Complex Social Container [2]. To exemplify further, consider the OPW scenario presented in Section 5.1. The complex social container in this scenario can contain a VO interaction among agents. The complex activity refers to the fact that different participants of the VO might be trying to achieve different things, often at the same time. Therefore we can have that some agents are coordinating to collect packets while other agents might be join or leaving the VO.

In summary, in order to extend the MAGE model to scale up also for large social environments, we need to consider the high number of entities sharing the social environment and the complexity of their interactions. A high number of entities sharing the social environment implies that we need to distribute the social environment in many social containers, while to handle the complexity of the interactions, we define complex social containers as entities that enable complex social activities in the environment. Lastly, the interactions we want to support have a dynamic nature. To preserve this property, we need to allow complex social containers to evolve in a life-cycle which eventually terminates at some point in time. In other words, once the agents involved in social interactions terminate their interactions, the social container can be destroyed so that the system can reallocate the resources to other agents requiring them.

Fig 5.3 shows, in more general terms, how we can structure the social envi-

[2]The name Complex Social Container refers to an extended version of the Atomic Social Container. In this chapter we will refer to Complex Social Containers also simply as social containers.

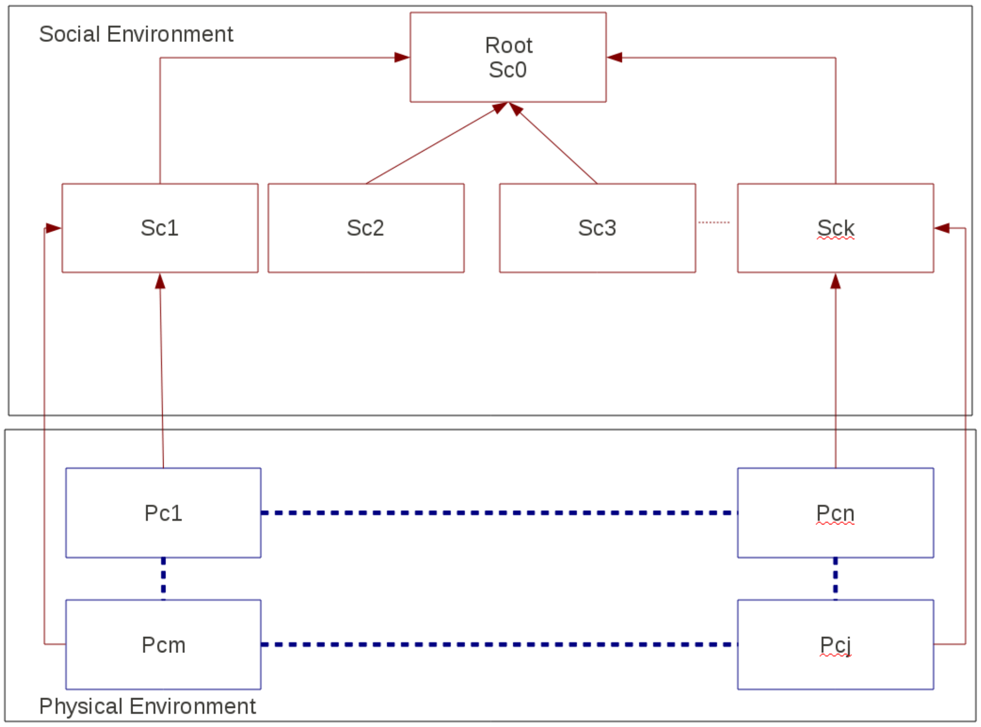


Figure 5.3: Distributing the Social Environment in a Complex Application.

ronment of an application using distributed social containers. The bottom containers are physical containers structured as neighbour containers [3] (represented as Pc1....Pcj). The physical containers are connected between each other with lines to show the neighbourhood relationship between the physical containers of the environment. The structure of the social environment contains a Root container represented as Root/Sc0 and a set of sub-containers Sc1....Sck. The Root container maintains and creates the structure of the social environment in terms of complex social containers. The social environment can start with only the Root container, and then evolve to have many social sub-containers (Sc1....Sck) which contain the ongoing game interac-

[3]Other organisations of the physical containers are possible, which for simplicity we ignore (for more details see Chapter 4 of Bromuri's work [19])

tions in the social environment. In Fig. 5.3 the social containers Sc2 and Sc3 represent containers that are created at run time. They are created from the Root container as its sub-containers. There is no direct relationship with the physical containers. Physical containers however are able to send asynchronous messages to these containers by utilising the connector interface of GOLEM (see [19] for more details). In some applications, there might be a need to define a default game for the interactions of the agents.

For example, in OPW, those social containers that are represented as super containers of the physical ones contain the OPW game (In Fig. 5.3 the social containers Sc1 and Sck are represented as super containers of respectively Pc1, Pcm and Pcn, Pcj.). Thus, from the system's startup, we have a social state that evolves as agents perform their pick, drop, move or flag actions. On agent's demand, the Root container can also create new social containers to contain new VO interactions. These social containers are not defined as super containers of the physical containers because there is no relationship between the VO interactions and the physical location of the agents (In Fig. 5.3 these containers are represented as Sc2 and Sc3).

5.3 Complex Social Containers as Compound Games

In order to specify complex social containers we use the notion of compound games. A compound game is a description of the relationships between atomic games. The compound game allows the definition of different possible interactions between participants that can terminate with possibly different outcomes and allows us to define more complex games in a component base manner. Thus, to model a complex social activity, we can combine the rules of various atomic games. At any time, only a subset of the atomic games might be active in the compound game. This has the advantage that, given a move performed by an agent, only a subset of the all possible rules of the compound game will be checked to determine the social consequences of such move.

There are three main issues that have been identified in representing compound games [112]:

- **Meta-Game:** In the social environment, we want to create and manage the social containers with a set of meta-moves and primitives that allow new game interactions to be created and then subsequently started, suspended, resumed, or terminated. We define a meta-game as a way for agents to use meta-moves to manage their interactions within containers by activating the primitives of the meta-game.
- **Coordination of moves in sub-games:** In the social environment agents interact with other agents through use of atomic games. Atomic games are started and evolve within social containers and describe the social rules of the environment. Often however, atomic games need to be combined to run in different orders: sometimes in parallel or depending on when the conditions arise. For this reason, we define a coordination mechanism for the moves in a sub-game running within a compound game. Such coordination assists the agent interactions by establishing the possible games that agents can play in the container at any time.
- **Forwarding moves:** In order to enable distributed norm checking in large scale applications, the moves that the agents perform in a game should be propagated to the right social containers where they will be evaluated using the rules of the game.

In the reminder of this chapter we are going to explain how we represent these three issues by exemplifying the discussion using the OPW. The representation provided here extends the work described in [112] in two ways: (a) we give an event-based interpretation of compound games using AEC and

(b) we provide a set of coordination primitives based on workflow constructs that can be reused across applications.

5.4 Meta-game aspects of Compound Games

As briefly stated in the previous Sections, a meta-game enables agents to define new social containers to contain their interactions. Usually, these interactions are described as compound games. The meta-moves that an agent performs, affect the interactions by changing the state of the compound game contained in the social container.

The Root container of Fig. 5.3 of the social environment provides the functionalities of a meta-game. Within this container we include meta-game rules to allow agents to create new game interactions, terminate game interactions and, if necessary suspend and resume interaction games as they are being played.

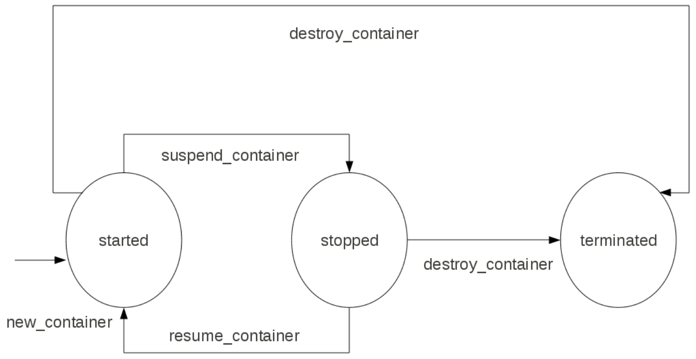


Figure 5.4: Compound-game life-cycle.

Fig. 5.4 shows a description of how the state of a compound game and the state of an atomic game change due to moves of agents. The figure shows that the compound game played in a container is in the started state after a new_container move. From this state it is possible to stop or to terminate the interactions in the container by respectively using the suspend_container and destroy_container moves. Once a game is in a stopped state, it is possible to resume it using the resume_container move. These moves are all meta-moves which are managed within the Root container. [4]

Fig. 5.5 shows an example of how the structure of the social environment evolves as result of agents performing meta-game moves. The figure shows, in four frames, how the social environment evolves in time due to the moves of the agents. The physical environment is distributed in four physical containers (Pc1-Pc4) [5].

- At Time=0 four agents a1-a4 are situated amongst the physical containers and the social environment contains the Root container and two social containers Sc1 and Sc2. The physical containers Pc1 and Pc3 have Sc1 as social super-container and Pc2 and Pc4 have Sc2 as their social super-container. These two containers relate, from the start-up of the system, the physical space (one or more containers) to a set of social rules. The new interactions can then generate new containers, containing other social rules.

- At Time=1 we assume that the agent a1 performs a meta-move. Such

[4]We also can manipulate the life-cycle of an atomic game by directly performing a move within the compound game where the atomic game is played. An agent can start and stop an atomic game by respectively performing a new_game or resume_game move and a suspend_game move or terminate_game move. These moves would follow the same principles of what we define here for describing the meta-game. The life-cycle of the atomic game would follow the same states, just the triggering moves would be different. The only difference here is that these moves are not performed within the Root container but are managed as part of the compound game

[5]Same structure as with the OPW shown in 5.1

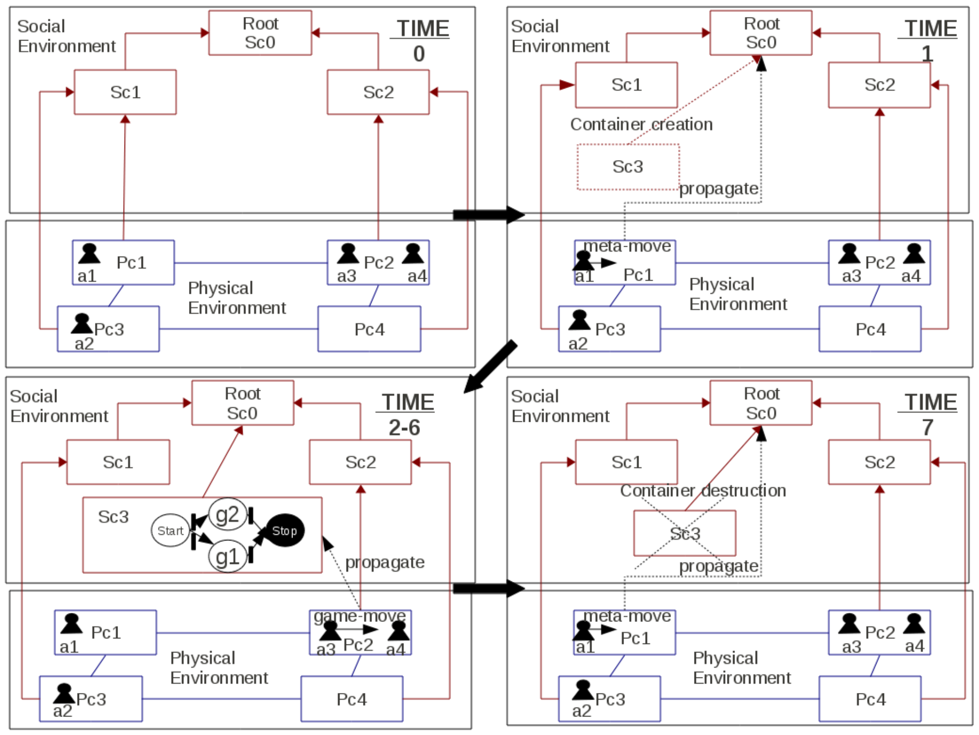


Figure 5.5: Evolving of the Social Environment Structure.

move is propagated from the container Pc1 to the Root container. Assuming that this meta-move was requesting to create a new interaction in terms of a complex game, the Root container generates a new container that uses the definition of the complex game as mediating rules. We show that the container Sc3 is created as a result of this action. The Root container stores these changes in its own state. By dealing with the meta-moves of the agents in such manner, we are able to know how the structure of the social environment changes in time.

- At Time=2-6 we assume that the agents interact following the rules of the compound game. If the agent a3 acts in the complex game contained in Sc3, the act is forwarded from Pc2 to the social container Sc3.

- At Time=7, assuming that the agent a1 performs a move to destroy the complex game, the meta-move is propagated to the Root container which destroys the container Sc3 and the social structure returns to the initial state. Not all the meta-moves succeed to change the environment. In order to perform them correctly, the agent must perform a valid move.

5.4.1 The moves of the Meta-Game

In the meta-game it is possible to define valid moves as in any other game. This is because in practical applications we may want to be able to restrict which agents can perform certain meta-moves. As discussed in the previous chapter, we can use normative concepts to define whether a move is valid in a game. We can distinguish this by defining a specific role (such as an agent with a special role in the system which can perform meta-moves), or by using some general rules that can evaluate at run time which of the agents in the system will have the privilege to perform such actions. For example, in the OPW scenario we can use the empowered/3 predicate to represent which agent in OPW is empowered to create a new VO interaction. We can write an empowerment rule as follows:

```
(Root Pow1) empowered(G@T, E)←
                do:E[actor ⇒ A, act⇒new_container, game ⇒ vo_game],
                holds_at(CID, sender, E, T),
                neighbouring_at(CID, [], _, 1, A, agent, points, Points,T),
                Points≥ 50,
                not holds_at(A, agent, leader, T).
```

The above rule states that all the agents that have not already created a VO and have more than 50 points in the social environment are empowered to create a new vo_game interaction.

5.4.2 Creating a new interaction game

To create a new interaction in the social environment an agent can request the creation of a new social container to contain it. The new interaction can specify an atomic game but we would generally expect that new interactions are described in terms of compound games so that the full functionalities of a complex social container are used. To define the new distributed agent environment, we also extend the happens/2 predicates with happens/3 predicates. The happens/3 predicate defines that an event E happens at time T in a container C of the agent environment. The rule for the creation of new interactions is specified as follows:

```
happens(E[actor⇒Root, act⇒ create,
            container ⇒ CID, game⇒ GameId],Root, T)←
     this(Root),
     happens(E*,Root, T),
     E*[actor⇒A, act⇒ new_container, game⇒ GameId].

assigns(E, container, CID )←
     E[actor⇒Root, act⇒ create, container ⇒ CID, game⇒ GameId].

initiates(E, CID, game, GameId)←
     E[actor⇒Root, act⇒ create, container ⇒ CID, game⇒ GameId].

assigns(E, game, GameId)←
     E[actor⇒Root, act⇒ create, container ⇒ CID, game⇒ GameId].

initiates(E, GameId, cycle, started)←
     E[actor⇒Root, act⇒ create, container ⇒ CID, game⇒ GameId].
```

which states that in the container Root an event E is generated to cause the creation of a new container CID to contain a specific game GameId. The event E can happen only if there is an event E* where an agent A requests the creation of a new interaction regulated by the game GameId (by performing an act labeled as new_container). As a result of the generated event, in accordance with the predicates (C1-C11) presented in Chapter 3, the following assigns/3 predicates create an object container identified as CID and an object of type game identified as GameId. The initiates/4 predicate stores the fact that the container CID has a property game whose value is GameId and it stores the GameId object as having the property cycle with value started.

We perform these operations to store the structure of the social environment within the meta-game, therefore we know which games are being played in every container of the environment. In this way we have centralised the management of how the structure of the social environment changes, however, the control of the moves of the agents within games remains distributed.

5.4.3 Stopping a game

To terminate an existing interaction, an agent must request to stop an existing game. The rule to terminate existing interactions is specified as follows:

```
happens(E[actor⇒Root, act⇒ terminate,
          container ⇒ CID, game⇒ GameId], Root, T)←
    this(Root),
    happens(E*,Root, T),
    E*[actor⇒A, act⇒ stop_container, game⇒ GameId],
    instance_of(CID, container, T),
    holds_at(CID, game, GameId, T).

destroys(E, GameId)←
    E[actor⇒Root, act⇒ terminate, container ⇒ CID, game⇒ GameId].
destroys(E, CID)←
```

```
    E[actor⇒Root, act⇒ terminate, container ⇒ CID, game⇒ GameId],
    not holds_at(CID, game,_,T).
```

similarly to the creation of new game based interactions, the termination requires the agent to specify which game is to be terminated. Consequently, the system uses its local information to locate the container where the game has been created and creates an event E to terminate the the specified game interaction and its container in the case that it does not contains anymore games.

5.4.4 Suspending and resuming game interactions

In particular cases, agents might need to suspend and resume games. We specify the following rules for suspending and resuming games:

```
happens(E[actor⇒Root, act⇒ suspend,
        container ⇒ CID, game⇒ GameId], Root, T)←
    this(Root),
    happens(E*,Root, T),
    E*[actor⇒A, act⇒ suspend_container, game⇒ GameId],
    instance_of(CID, container, T),
    holds_at(CID, game, GameId, T),
    holds_at(GameId, cycle, started, T).

initiates(E, GameId, cycle, suspended)←
    E[actor⇒Root, act⇒ suspend, container ⇒ CID, game⇒ GameId].

happens(E[actor⇒Root, act⇒ resume,
        container ⇒ CID, game⇒ GameId],Root, T)←
    this(Root),
    happens(E*,Root, T),
    E*[actor⇒A, act⇒ resume_container, game⇒ GameId],
    instance_of(CID, container, T),
    holds_at(CID, game, GameId, T),
    holds_at(GameId, cycle, suspended, T).

initiates(E, GameId, cycle, started)←
    E[actor⇒Root, act⇒ resume, container ⇒ CID, game⇒ GameId].
```

The first two predicates state respectively that an event E to suspend an interaction is created in the Root container, if an agent requests to suspend the game GameId with an E*. The game GameId should be in a started state of the life-cycle. As result of the event E created in the container, the second predicate changes the value cycle of the object GameId into suspended. The last two predicates deal with the resumption of a game in a similar way. The happens/3 predicate checks, before creating an event E to resume the game, that the game that was requested to be resumed with a resume_container act and that the game is currently in suspended state.

5.5 Coordination of moves in Sub-Games

With a compound game we want to be able to parallelise, choose or synchronise between different atomic games. To capture these control-flow aspects of compound games we produce a coordination framework that allows us to coordinate complex interactions built from simpler ones. The resulting framework is then applied to support workflow activities.

Compound games in MAGE are defined using coordination mechanisms to determine which games run at a given time of the interactions amongst agents. In order to define the coordination mechanisms we specify workflow patterns to define what games should run at a given time.

In general the term workflow refers to an activity that addresses some business needs by carrying out specified control and data flows amongst sub-activities [111]. A workflow procedure consists in a set of atomic activities and relations between them. Such relations coordinate the participants and the activities they need to perform [4]. The participants of a workflow can be a group which share a set of tasks or a human resource, a software application or a specific hardware with the ability to execute an activity. The link here is that the participants are the agents and the atomic activities are the atomic games.

By specifying the "expected" flow of work, the workflow mechanism supports coordination of games. Given a description of a compound game, we use these mechanisms to support the interactions between agents by coordinating their expected behaviour.

The games are linked together to identify a logic of execution between them. Their execution identifies the active games by using specific control patterns:

- **Sequential:** An activity B is sequential to the activity A when B is executed after A.
- **Parallel:** Two or more tasks are parallel if they are executed in parallel. The management of parallel activities identifies two conditions: and-split between activities which allow them to be concurrent activities or and-join which synchronises two or more parallel flows.
- **Iteration:** One or more activities are executed a certain number of times.
- **Conditional:** One of the alternative activities is executed. The management of conditional activities identify two conditions: xor-join where none of the alternative branches is executed in parallel, xor-split where according to a condition only one branch is chosen.

5.5.1 VO Compound Game in Open-Packet-World

In the OPW scenario, agents move into a grid to collect packets and bring them into destinations. Additionally, in order to be more competitive in the packet collection, we allow agents to dynamically create VOs as we described in Section 5.1. In this section we describe how we can define a VO compound game for OPW. Before defining the compound game, we need to decide the atomic game interactions that are needed in the OPW scenario that supports VOs . To this purpose, we define four additional atomic games to the OPW game presented in the previous chapter.

To allow agents to create VOs, we extend the repertoire of actions given in the OPW game with the following actions: create_vo, invite, accept, reject, collect, observe, leave and dissolve. Agents may decide to use these actions if they believe that it will help them to achieve their goals. Less sophisticated agents, can still be part of the system without being involved in a VO. With a new_game action an agent triggers the creation of a new VO game which might result in a VO. With an invite action an agent can invite new members to join the VO. With an accept or reject action an agent can respectively accept or refuse to join the VO. With an observe action an agent notifies the presence of a packet in the grid. With a collect action an agent asks another agent to collect a packet. With a leave action the agent expresses the will to leave the VO and finally a destroy_game action an agent can dissolve the whole VO.

In figure 5.6 we show how we can define a set of atomic games capturing the interactions of the agents from the social perspective. For illustration purposes we keep the games simple. The states of the game are represented with circles and with arrows we show what moves are possible when the game is in a particular state. With P and L we respectively denote the picker role and the leader role of the agent performing the action and with C we denote events being generated by the containers.

To allow agents to create a new VO, we define the Create_VO game. Using

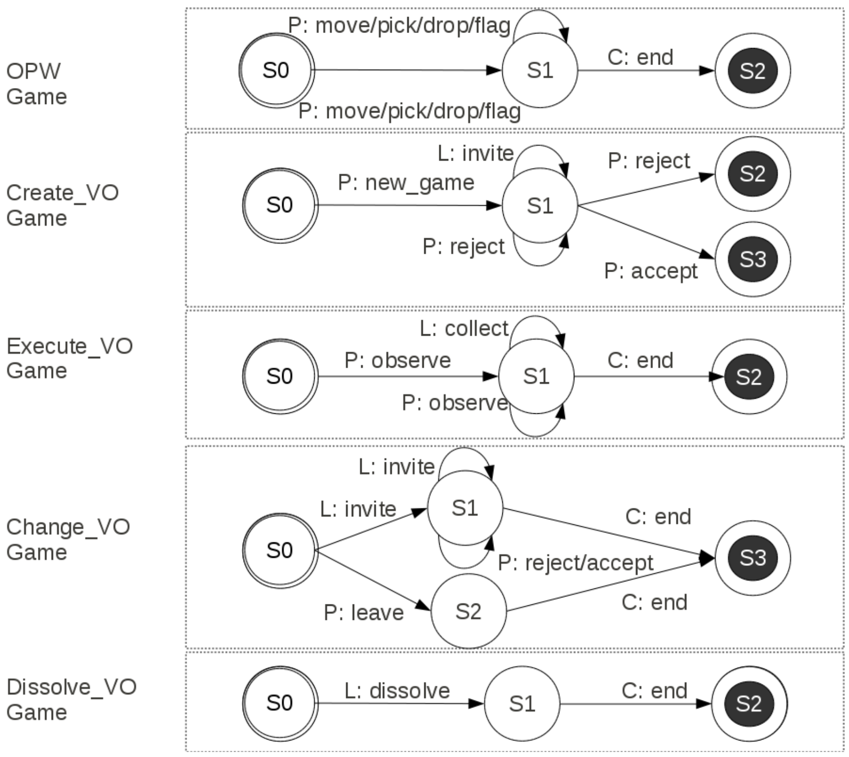


Figure 5.6: Interactions as Games in Open Packet-World.

such game, an agent can initiate interactions with other agents. The game define rules such as: an agent is empowered to create a new VO if it has more than 50 points. If an agent succeeds in creating a VO, it assumes a leader role within the created VO. Afterwards, the Execute_VO is used as a game during which agents can coordinate with one another. Picker agents are permitted to communicate to the leader if it observes packets in the environment which it does not collect. The leader, on the other hand, is empowered to request agents in the VO to collect a packet. In this case, agents are obliged to collect the packet assigned by the leader. The Change_VO game is used to invite new members to join or for the agents who are already members of the VO, to leave the VO. And finally, in the Dissolve_VO game, the agent with a leader role is empowered to dissolve the whole VO.

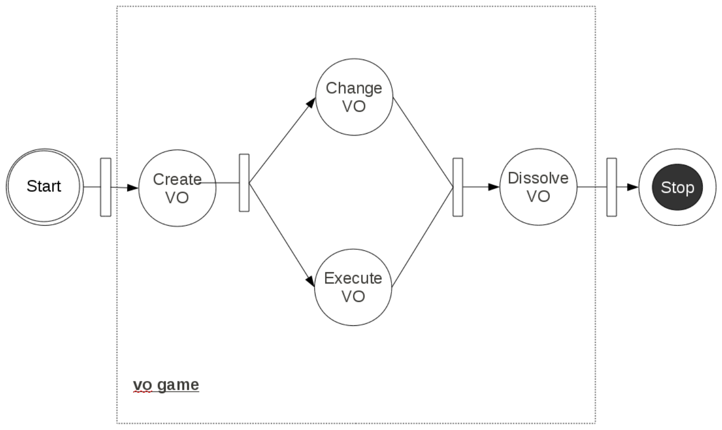


Figure 5.7: The vo_game as a compound game in the Open Packet-World.

In Fig. 5.7 we show a workflow example for the OPW scenario. The figure shows how a complex game is defined in terms of a workflow that models the interactions in a VO. The activities are represented with cycles and the transitions from one activity to the other are described with arrows. As these transitions can be conditional, the vertical bar in the transition denotes that conditions exist for the transition to happen.

With the vo_game example we show a subset of all the possible patterns. However, by varying the compound game using different patterns, the same coordination mechanism allows sub-games to be played in different order thus resulting in different type of interactions. In the following Sections we illustrate how we can interpret these patterns in order to coordinate active sub-games.

After we define these atomic games for the OPW scenario, we can combine these games into a complex game using the workflow coordination patterns. The complex game is started and evolves within a complex social container that mediates the interactions. Using the defined atomic games, agents can change the properties of the social environment by creating temporary persisting entities such as the VO itself, sanctions, rewards or obligations. We specify the compound game using the coordination patterns, in this way, we are able to establish the active sub-games in the compound game. The state of a compound game is described as follows:

```
vo_game: Id [
 members ⇒ {agent:a1, agent:a2, agent:a3},
 sub_process ⇒ Workflow,
 cycle ⇒ started,
 result ⇒ nil
].
```

The sub-games of the vo_game are specified in the Workflow value of the sub_process attribute. The Workflow is specified as a term of the form:

```
seq([
     create_vo:c1,
     and_split((create_vo:c1, []), [change_vo:ch1, execute_vo:ex1]),
     and_join([change_vo:ch1, execute_vo:ex1], ([], dissolve_vo:d1)),
     stop
  ].
```

The vo_game goes through the designed workflow process which specifies that a create_vo game should be played first. After such game terminates the agents can play in parallel the change_vo and the execute_vo games. These two games can join after their termination into the dissolve_vo game termination which is followed by the termination of the whole vo_game.

5.5.2 Active sub-games

The main issue to be considered in compound games is the coordination of moves in active sub-games. We define coordination specifying the predicate active_at/3. Using active sub-games, we can define valid moves in a complex game to include all the valid moves in the active sub-games:

```
valid(CG@T, Move) ←
  active_at(CG, SubG, T),
  valid(SubG@T, Move).
```

The above valid/2 predicate checks if the action Move is valid in an active sub-game identified as SubG@T. The active sub-games are identified using the active_at/3 predicate which is specified as follows:

```
active_at(G, SubG, T)←
    instance_of(G, compound_game, T),
    holds_at(G, process, Workflow, T),
    pattern(Workflow),
    runs(G, Workflow, SubG, T).
```

Patterns in our framework are interpreted by a runs/4 predicate that parses the coordination structure and checks which sub-games are running.

5.5.3 Coordination of Sequence and Interaction Patterns

The sequence pattern allows one activity to be executed after another. In compound game terms, a sequence A_{i+1} can start after the game A_i has terminated as shown in Fig. 5.8.

Sequential games may or may not have conditions in the sequence, if there is no condition we define a seq pattern, in case we have a condition between

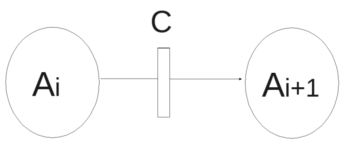


Figure 5.8: The *sequence Workflow Operator*.

sequential activities then we define an if pattern. Similarly to the sequence pattern, the iteration pattern defines a repetition of an activity under a certain condition, we call this pattern a repeat pattern.

```
runs(G, seq([A|_]), A, T)←
    not pattern(A),
    not terminating(A@T,_).

runs(G, seq([A|B]), C, T)←
    not pattern(A),
    terminating(A@T,_),
    runs(G, seq(B), C, T).

runs(G, seq([A|B]), C, T)←
    pattern(A),
    (runs(G, A, C, T);
    runs(G, seq(B), C, T)).

runs(G, if(Conditions, P), C, T)←
    solve_at(Conditions, T),
    (pattern(P) →
    runs(G, P, C, T); C=P).

runs(G, repeat(P, Conditions), A, T)←
    not solve_at(Conditions, T),
    runs(G, P, A, T).
```

```
pattern(P)← sequence(P).
pattern(P)← if_conditional(P).
pattern(P)← repeat_loop(P).

sequence(seq(_)).
if_conditional(if(_,_)).
repeat_loop(repeat(_,_)).
```

The runs/4 predicate checks if in a complex game, described as seq([List]), the head of the list is not a pattern, it checks if the activity identified as the head of the list is still running and if this is the case the game identified in the head of the list is considered as the running game. If the head of the list is not a pattern (this means that it is an atomic game) and it is in a terminating state, then the program calls recursively itself by removing the head of the list from the description of the complex game. If the head of the list is a pattern, then the head and tail of the list are checked calling the runs/4 predicate to determine which games are running.

Note that the top-level game G is required as a parameter in the definition of runs/4 as a reference to the global variables of the interaction. Note also that the definition of the above patterns can be combined to form arbitrary complex structures, which is indicative of the expressive power of the framework.

Optionally, it is possible to define conditions Conditions for a game to become active. The predicate solve_at/2 checks these conditions which are expressed as OEC properties existing in the state of the game. For example, in order to activate a conditional game, we first check if the conditions to activate such game are satisfied. For the iteration on the other hand, the game runs until the running conditions are not true.

The solve_at/2 predicate specifies how given a list of conditions for patterns are checked and it is defined as follows:

```
solve_at([], Time).

solve_at([H|T], Time)←
        H = instance(Id, Class),
        instance_of(Id,Class,Time),
        solve_at(T,Time).

solve_at([H|T], Time)←
        H = object(A,B,C),
        holds_at(A,B,C,Time),
        solve_at(T,Time).

solve_at([H|T], Time)←
        call(H),
        solve_at(T,Time).
```

In this way, we have a flexible mechanism to describe conditions. For example in OPW we might want to activate an operate_vo game only if there is an instance of agreement amongst players.

5.5.4 Coordination of Parallel Patterns

To coordinate parallel games we use and_split and and_join patterns. In an and_split the termination of the game A_i triggers the games $A_{i+1}....A_{i+n}$ to be active at the same time. as shown in Fig. 5.9. A and_split workflow primitive is described as follows:

```
and_split((A, Conditions), Activities)
```

which states that after activity A is completed, if the Conditions is true, then the set of Activities (In Fig. 5.9 denoted as $\{A_{i+1}....A_{i+n}\}$) must be

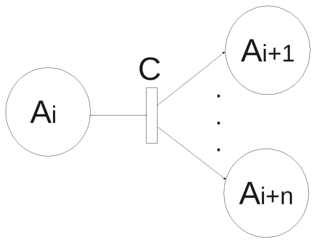


Figure 5.9: The *and_split Workflow Operator*.

carried out in parallel. To support the parallel composition required for this coordination pattern, we define runs/4 as follows:

```
runs(G, and_split((A,_),_), A, T) ←
    not pattern(A),
    not terminating(A@T, _).

runs(G, and_split((A, Conditions), Activities), C, T)←
    terminating(A@T,_),
    solve_at(Conditions,T),
    member(Activity,Activities),
    not terminating(Activity@T,_),
    (pattern(Activity) → runs(G,Activity,C,T); C=Activity).
```

The above specification of runs/4, similarly to the sequence and iteration patterns, checks if the game preceding the split has terminated. If it has, then it checks that the conditions for the split are true. For each new game that should be activated antecedent to the split, it checks if they have terminated. If not, it checks if they are patterns themselves. If these games are patterns, then the runs/4 is called recursively to evaluate the additional patterns, otherwise it activates directly the game.

In a and_join pattern we need to check that the termination of different games $A_i....A_{i+n-1}$ has completed before activating A_{i+n}, as shown in Fig. 5.10.

An and_join workflow primitive is described as follows:

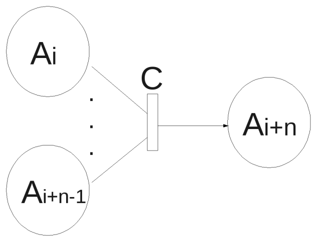


Figure 5.10: The *and_join Workflow Operator.*

and_join(Activities, (Conditions, A))

and states that after the set of activities Activities (In Fig. 5.10 denoted as $\{A_i....A_{i+n-1}\}$) are completed, if the Conditions are true, then the activity A must be carried out (In Fig. 5.10A is denoted with A_{i+n}). To support the synchronisation of parallel activities required for this coordination pattern, we define runs/4 as follows:

```
runs(G, and_join(Activities,(_,_)), A, T) ←
    forall(member(A,Activities), terminating(A@T,_)),
    not pattern(A).

runs(G, and_join(Activities, (Conditions, A)), C, T)←
    forall(member(X,Activities),terminating(X@T,_)),
    solve_at(Conditions,T),
    (not pattern(A),
    C=A;
    runs(G,A,C,T)).
```

For the and_join pattern, the runs/4 predicate checks first that all the games that are defined as precedent to the join have terminated. If this is the case and the activation conditions are true (we can also express cases where there are no condition for the patterns by defining an empty list) we check if the antecedent game is a pattern, if it is, it needs further investigation using the runs/4 predicate, if not, this is the new running game.

5.5.5 Coordination of Conditional Patterns

Similarly to the parallel patterns, we define the conditional patterns namely the xor_split and the xor_join patterns.

In a xor_split the condition on the result after the termination of A_i determines the active game between several $A_{i+1}....A_{i+n}$ games as shown in Fig. 5.11.

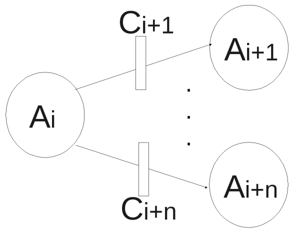


Figure 5.11: The *xor_split Workflow Operator.*

A xor_split workflow primitive is described as follows:

xor_split(Activity,Conditions)

and states that after an activity A is completed, the Conditions set $\{(Condition_i+1,\ A_i+1),\ ....(Condition_n+1,\ A_n+1)\}$ will determine which of the activities $A_i+1,\ ....A_n+1$ from the condition set Conditions must be carried out.

To support the conditional split required for this coordination pattern, we define runs/4 as follows:

```
runs(G, xor_split(A,_), A, T) ←
    not pattern(A),
    not terminating(A@T, _).

runs(G, xor_split(A, Conditions), C, T)←
    terminating(A@T,_),
    member((Condition,Activity),Conditions),
    runs(G, if(Condition,Activity),C,T).
```

The runs/4 predicate checks if the game preceding the split has terminated. If it has, then for every pair (Condition, Activity) it uses the if conditional pattern to check if the game identified as Activity should be activated.

In a xor_join the termination of one of several possible games A_i, A_{i+1} .... A_{i+n-1} activates the next game A_{i+n} as shown in Fig. 5.12.

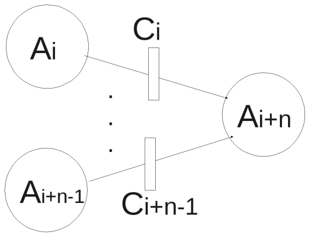


Figure 5.12: The *xor_join Workflow Operator.*

A xor_join workflow primitive is described as follows:

xor_join(Conditions,Activity)

. The set of Conditions is defined as a set of pairs $\{(A_i, Condition_i)$, ...$(A_i + n - 1, Condition_i + n - 1)\}$. After one of the activities A_i, ...$A_i + n - 1$ from the condition set Conditions is completed and the associated $Condition_i$, ...$Condition_i + n - 1$ is true, the activity Activity must be carried out. To support the parallel composition required for this coordination pattern, we define runs/4 as follows:

```
runs(G, xor_join(Conditions, A), C, T)←
    member((Condition,X),Conditions),
    terminating(X@T,_),
    solve_at(Condition,T),
    (not pattern(A),
    C=A;
    runs(G,A,C,T)).
```

The runs/4 predicate for the xor_join pattern checks that at least one game preceding the join has terminated. If it has and the condition associated within such game is true, then the antecedent game in the join pattern is activated.

5.6 Forwarding Agents Moves

Agents are situated in the physical environment. They interact in the environment by producing events in physical containers. An event performed in the physical environment succeeds only when the event is considered physically possible. Afterwards, we propagate the same event in the social environment so that we check if the interaction respects the social rules and possibly if causes changes to the social state.

There is a connection between physical and social containers in the sense that both entities mediate the same actions performed by agents but with different rules. We represent the link between social and physical containers by defining a structure for the agent environment as we illustrated in Fig. 5.3. The agents are capable of perceiving both the social and physical environment by querying the environment and by being notified of the event of interest.

The link between the social and the physical environment is defined using the following principles:

- The social environment is organised as a two layered structure having a Root container and a set of social containers that are defined as sub-containers of the Root container.
- In order to apply a set of default social rules to the agent environment every physical container can be associated with one social-container. This association defines a relationship where the social container is a super-container of one or more physical containers.

- Interactions may create new social containers, where additional social rules apply to a selected subset of agents that are involved in these interactions. These interacting agents can be located in the same physical containers or in different ones.

As agents act in the physical environment they generate events which are propagated in the social environment. The propagation from the physical to the social environment is based on the following happens/3 predicates:

```
happens(E,R,T)←
    this(C),
    happens(E, C, T),
    do:E[actor⇒A, act⇒ Label, content ⇒ Content],
    meta-move(Label),
    instance_of(R, root, T).

happens(E,SC,T)←
    this(C),
    happens(E, C, T),
    do:E[actor⇒A, address ⇒ {game:GID, container:SC},
        act⇒ Label, content ⇒ Content],
    instance_of(SC, social_container, T).
```

The first predicate distinguishes between agent acts that are classified as meta-moves of the social environment. These meta-moves are propagated to the Root container which will check against its own local rules and possibly apply the effects of the act as described in Section 5.4. The second predicate propagates all the other moves to the social container SC where they are addressed.

Agents might perform an act without specifying a game. Physical acts, for example, are acts performed to manipulate objects. These acts might have social consequences which might be applied using a set of default games that are active since the start of the system, as we explained in Section 5.2. In this case agents are playing a game which they have not explicitly created for themselves. Thus, when agents perform one of such acts they do not address this game. To deal with the propagation of the acts, we define the default game as part of the super social container and acts not addressing a game are propagated to this game. We express this rule as follows:

```
happens(E,SC,T)←
    this(C),
    happens(E, C, T),
    do:E[actor⇒A, act⇒ Label, content ⇒ Content],
    instance_of(SC, social_container, T),
    holds_at(C, super_container, SC, T).
```

which states that the moves generating an event E in the container C and which do not specify where the move should be propagated, are propagated to the super social container of the container identified as SC.

5.7 Summary

In this chapter we have further developed the MAGE framework as a component based social infrastructure built on top of the GOLEM platform [21] to support practical applications. We have focused on scenarios requiring the distribution of the social environment in many social containers. We illustrated how we create a link between the social and the physical environment. While the physical environment has its own complex structure, we architect the social environment as a hierarchy of containers with a Root container as the start point which is also in charge for the dynamic changes to the social environment such as creating new containers to contain new game interactions. We explained the meta-game as the mechanism used to define new interactions in complex social containers. We described how by performing a meta-move agents are enabled to change the state of interactions at run time.

In this chapter we also described how to build compound games from simpler sub-games. We looked at a set of coordination patterns that allow us to build compound games as result of coordinating atomic sub-games. We studied how to define coordination by introducing the notion of active sub-games and we defined what counts as valid move in a compound game. We described seven main coordination patterns (sequence, if, return, and_split, and_join, xor_split and xor_join) and defined how we can build compound games based on these seven coordination patterns. We also showed how to define a vo_game as a compound game in the OPW scenario.

Events happening in the environment will be evaluated against the social rules. We distribute the social rules into various containers, for this reason we have presented a propagation mechanism so that events are propagated into the right container to be evaluated as moves within a game. In the next chapter we discuss implementation issues of the MAGE model.

Chapter 6

Implementation

In the previous chapters we presented how to specify atomic games and how to construct complex games within MAGE. In this chapter we present the architecture and implementation of MAGE and how it relates to the GOLEM reference model.

The technologies used for the implementation of MAGE are Prolog and Java. MAGE follows a similar approach to that used in GOLEM. To allow for easy integration between GOLEM and MAGE, a version of Prolog named tuProlog [45] is used to define the rules of social and physical containers and the Java programming language is used to code the rest of the platform.

The chapter is structured as follows: Section 6.1 introduces the GOLEM reference model and Section 6.2 shows how we extended the GOLEM architecture with the MAGE functionality. Section 6.3 gives an overview of the MAGE architecture and the main entities. Section 6.4 focuses on the implementation of the propagation and coordination mechanisms that deal with the management of the Social Containers. Section 6.5 gives implementation details of the social rules, how we can define enforcement policies such as punishments and rewards and how agents can actively perceive what happens in the social environment. Finally, Section 6.6 summarises and concludes this chapter.

6.1 The GOLEM Reference Model

The implementation of the GOLEM framework is represented in the GOLEM reference model shown in Fig. 6.1. The agent environment evolves as `Agents`

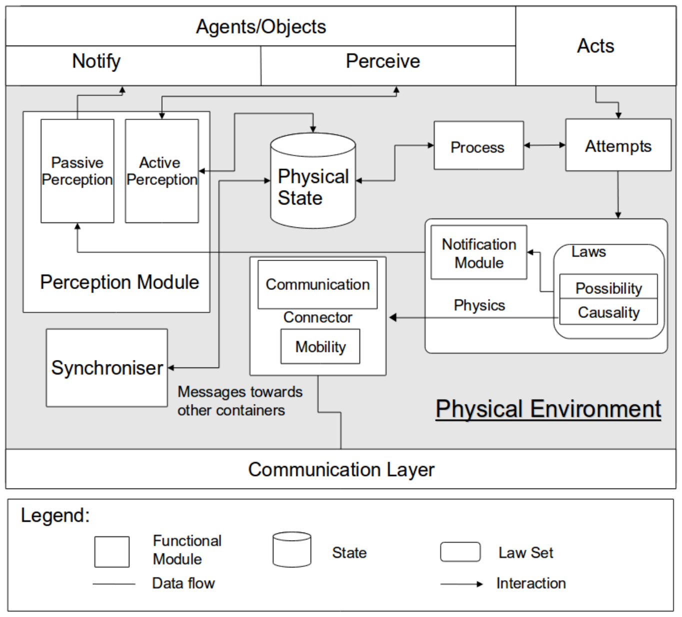


Figure 6.1: GOLEM reference model.

perform `Actions` in the environment. These actions are captured by the `Attempts` module and are treated as attempts regulated by the `Physics` component. The `Physics` describes how and which agents actions cause changes to the physical state of the environment. The `Laws` module of the `Physics` defines a set of physical `Laws` that determine if the actions performed by agents are possible to happen in the environment. Once an action is possible, this is established as an event that happens in the environment and it is directed by the `Notification Module` to the `Passive Perception` module that notifies the sensors of agents and objects. Agents can also actively perceive the environment or objects in it. This is handled by the `Active Perception` module that accesses the state of the agent environment to support agents' requests to perceive the environment.

As containers in GOLEM can be distributed over a network, a `Synchroniser` module is introduced in every container to keep the containers synchronised. GOLEM also provides a `Connector` component as a service in the agent environment that hides from the agents the complexity of interfacing with the `Transportation Layer`. A `Connector` service uses the `Transportation Layer` to send messages from the container where the connector is registered to another container where the message is directed.

6.2 The MAGE Reference Model

MAGE adds an extra layer on top of the GOLEM platform. The reference model of MAGE extends GOLEM as shown in Fig. 6.2. By adding the MAGE framework to the GOLEM implementation, we create a double structure of the agent environment that evolves in time: i) the physical structure where the agents and objects are located and ii) the social structure that checks if the actions of agents conform to the social rules.

In the integrated model of GOLEM and MAGE shown in Fig. 6.2, when an `Action` is performed, this is evaluated in both the social and physical environments: GOLEM checks that the action is possible according to the physical laws and MAGE checks that the action is valid according to the rules of the game.

To link the physical and the social layers of the agent environment we provide a propagation mechanism which propagates the actions performed by agents in a physical container to a social container that can evaluate it.

The acts performed within a social container are first evaluated using the

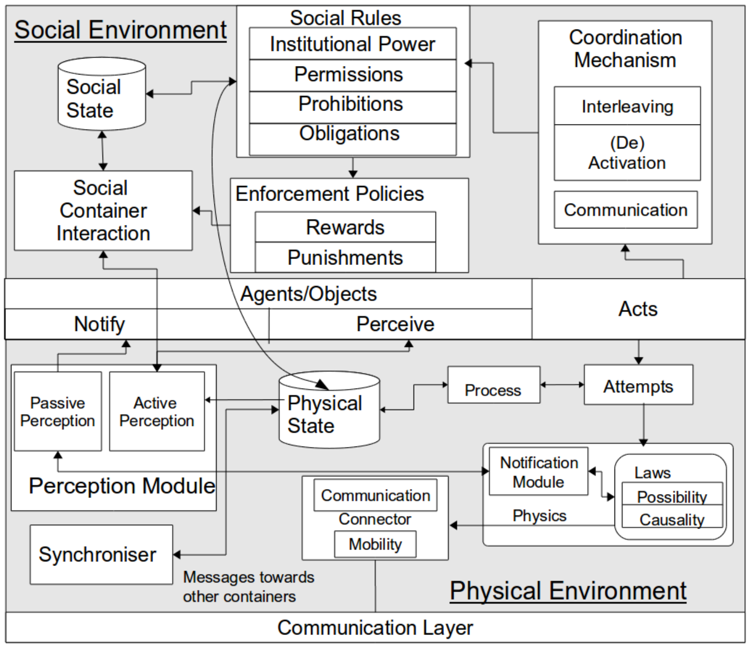


Figure 6.2: GOLEM and MAGE reference model.

`Coordination Mechanisms` offered in the social environment. The coordination mechanism uses a `Communication Mechanism` to exchange information amongst containers, an `Activate/Deactivate` used by containers to activate and deactivate games and an `Interleaving Mechanism` that checks which games are active. The rules of active games are used to evaluate the agents actions.

The validity of an act is checked using a subset of the `Social Rules` defined in the social environment. The `Institutional Power`, `Permissions`, `Prohibitions` and `Obligations` of the agents are defined as domain specific rules which are encapsulated as part of the games that agents play. The social rules may be based on the `Physical State` of the environment or/and the `Social State` of the environment.

The `Enforcement Policies` are included as part of the games whenever there is a need to `Punish` or `Reward` the behaviour of the agents. These rules may affect the social status or the resources of an agent (i.e. in the OPW the points that an agent has), through the use of the `Social Container Interaction Module`. The `Social Container Interaction Module` allows agents to actively perceive the social state of the games.

6.3 MAGE Implementation Environment

We can identify three main components involved when implementing a MAGE based application: agents, social containers and games. Each component has the following requirements:

- **Agents:** Agents in GOLEM are entities with a body, sensors, effectors and a mind [21]. The definition of a new agent (i.e. for OPW) occurs in a similar manner, independently of whether the application includes the MAGE layer or not. What changes when implementing such agents is only the specification of the agent mind, which is a Prolog module that has to be specialised to reason depending on the applications needs. GOLEM supports tuProlog [45], Sicstus [46] and SWI-Prolog [131] mind specifications. We implemented our applications using SWI-Prolog specifications of the agent mind. Agents were defined to perceive what is happening in the social and physical environment, reason about available moves and use their strategy to reach their goals.
- **Social Containers:** Social Containers (complex or atomic) are defined as Java thread processes with a Prolog core. The definition of social containers is similar to the one of physical GOLEM containers. What changes are the affordances of the container and the rules governing it. The container is now perceived as a social container by other entities in the agent environment. The Prolog core of social containers is specified in tuProlog [45] which was chosen for the ability to integrate with Java.

By using this Prolog version, it is possible to define, instantiate and use Java objects whenever necessary in the Prolog theory. Due to this feature of tuProlog, we can define and destroy dynamically the Social Containers at the Prolog and Java level.

- **Games:** We use tuProlog to associate social containers and games (complex or atomic). The games are loaded as the components that regulate agent interactions within containers. Depending on which games are running at a given time, the rules of such games will be activated to check the moves made by the agents in the state of those games.

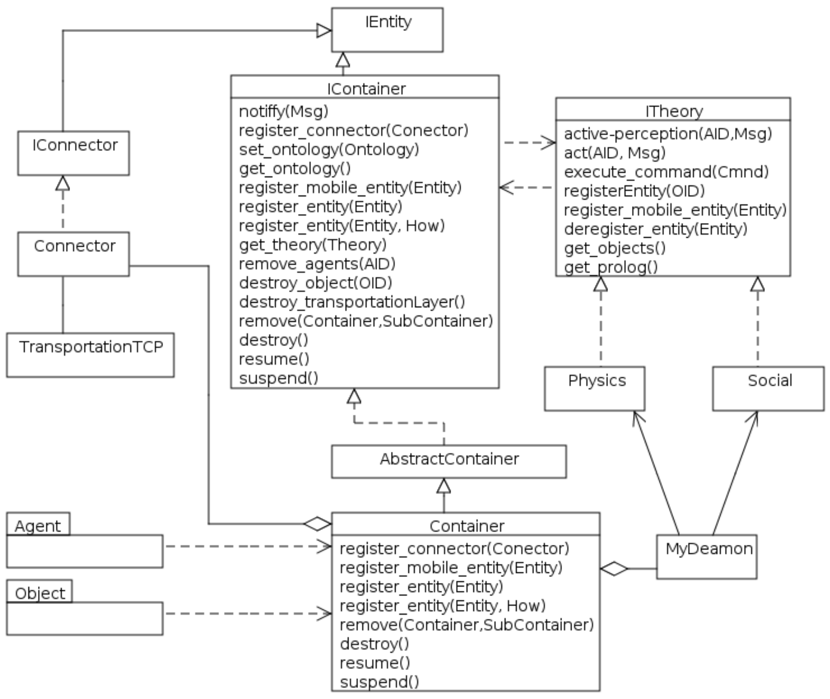


Figure 6.3: GOLEM and MAGE Container Architecture.

In Fig. 6.3 we show the UML diagram of the extended model of GOLEM containers. In this new model, we specialise the containers into social and physical containers. Every Container has a Prolog theory which describes the rules that the container uses to mediate the interactions. The theory ITheory is now specialised to distinguish between Physical and Social Containers. The rules defining physical containers are Physics rules while the rules defining the social containers are Social rules. Every container refers to an entity called MyDeamon which provides an interface for producing remote calls AEC calls in the container.

The Physics and the Social module mediate respectively the physical interaction between the agents and objects registered within a container and the social interactions between the communicating agents. The Physics and the Social component contain a tuProlog engine, thus the interaction is defined in a declarative way. The modules of the reference model in Fig. 6.2 are implemented using Prolog predicates. The details of how we define the internal architecture of the Social module will be shown in the next Section.

Agents and Objects are contained within containers by registering them using the register_entity or register_mobile_entity methods. A Container has one or more connectors that define the agent environment as an interconnected structure. The connectors are registered in containers by using the register_connector method. Connectors provide a transportation layer to the agent environment. GOLEM defines the TransportationTCP class to hide the complexity of communicating in the agent environment. The Container is also extended with new methods that we use during the Coordination Mechanism. This methods deal with the destruction, the suspension and the resuming of containers.

The actions of an agent are generated by default in the physical container where the agent is situated. An agent uses its effectors and sensors to call the following methods of the physical container:

```
public void act(String ActorId, String message);
```

The method is called by an agent effector to produce an action in the agent environment, being the first argument the unique identifier of the actor and the second argument a tuple representing the action.

6.4 Social Container

The implementation of MAGE follows object oriented patterns [60], extending on the patterns that GOLEM already used and introducing two more patterns to handle the social state (Facade and Factory patterns). The full list of patterns utilised in MAGE are:

- **Decorator Pattern**: The Decorator Pattern implies encapsulating an object or a module within another object to extend the functionalities of the previous object. The main advantage of this pattern is that the functionalities of an existing module/object can be extended at runtime without modifying its code. The GOLEM platform utilised the decorator pattern to encapsulate the state of internal objects in GOLEM objects (see [19]). MAGE utilises the decorator pattern to extend the current implementation of the AEC to handle social states, by introducing further predicates to propagate events and to interleave games.

- **Mediator Pattern**: The Mediator Pattern has the purpose to standardise an interface for the interaction with a set of components that only need to be able to interact with the mediator, abstracting away the complexity of dealing with multiple heterogeneous components. GOLEM made use of the mediator pattern to handle the interaction between agents, objects and containers. MAGE extends the utilisation of this pattern by introducing a further level of mediation between the agents and atomic and compound games. In other words, the social containers are mediators of the interaction between the agents and the atomic and compound games.

- **Factory Pattern**: The Factory Pattern consists in a `factory` module that is in charge of instantiating objects at runtime according to the request of a `requester` module. MAGE utilises the factory pattern in two ways: i) a `Root` social container can instantiate social sub-containers when this is required during the social interaction ii) a social container can instantiate atomic games and compound games to handle the interaction amongst a group of agents.

- **Facade Pattern**: The Facade Pattern consists in a module that encapsulate the state of one or more sub-modules to create a simplified and coherent interface of a system. MAGE utilises this pattern when dealing with atomic and compound games, hiding the complexity of dealing with the state of the games behind an `Interleaving Module` that provides the predicates for the coordination of the agents. In other words, thanks to the `Interleaving Module` the state of the games is transparent to the agents, that just see what are the valid moves at a given time.

- **Publish/Suscribe Pattern**: The Publish/Subscribe Pattern consists in an entity, the *subscriber*, subscribing to another entity, the *publisher*, to observe its state changes in terms of notification of events. GOLEM made use of this pattern to create agents able to observe the environment according to the events happening in it, thanks to their sensors and produce events in it, thanks to their effectors. MAGE extends GOLEM use of this pattern by having agents that are subscribed to social containers and can observe their state changes or produce new social events.

In Fig. 6.4 we represent the MAGE architecture. We have abstracted away from the features represented in the physical agent environment.

In MAGE, agents rely on the physical environment to perceive and act in the environment. MAGE provides a `Propagation Mechanism` to connect the

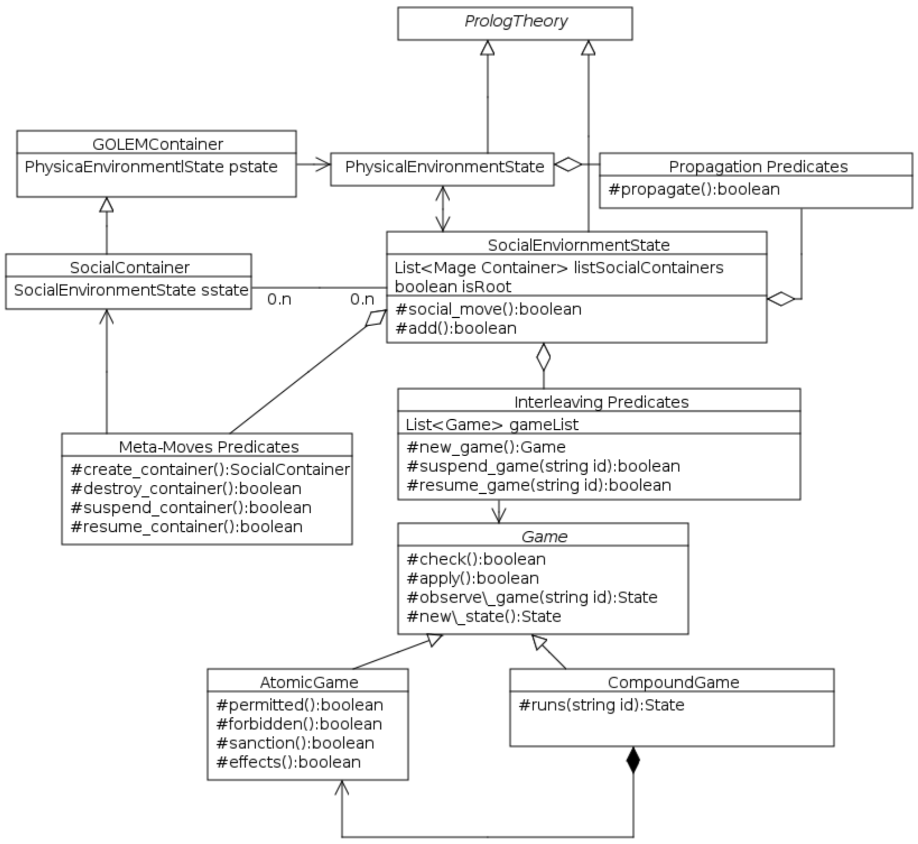


Figure 6.4: MAGE Architecture.

social and the physical environment. The `Propagation Mechanism` propagates the actions that the agents perform in the physical environment to the social environment. The social environment is composed by two types of `Coordination Mechanisms` specified as the `Meta-Move Predicates` which can change the structure of the social environment by creating and deleting social containers, and the `Interleaving-Predicates` which allows agents to decide which atomic games to play. The interleaving predicates act upon the state of `Games`. A `Game` can be a `Compound Game` meaning that it follows the coordination patterns presented in Chapter 5. The state of the games is defined within the social containers and it can be observed by agents that are playing in such games. A `Compound Game` contains one or more `Atomic Games` which are also instances of `Games` and describes normative relations between acts that agents perform.

6.4.1 Social State

As briefly explained in Section 6.3, agent actions cause updates in the state of containers. To update the state of containers (social and physical) we use OEC [76] which is an object-oriented optimised version of EC. Clearly EC can be implemented in other programming languages, such as Java and C. We adopted the logic programming approach partly because EC was originally developed as a logic programming language, and partly because of the declarative semantics and concise representation offered by logic programs.

The implementation of OEC in tuProlog does not support efficient indexing of clauses. To improve the performances of OEC, GOLEM was interfaced with the Berkeley database [93] as extensively explained in [19]. The Berkeley database stores instances of tuples of type:

```
instance(Obj,Class,start(Ev)).
object(Obj,Attr,Val,start(Ev)).
```

The `instance/3` and `object/4` assertions store the state of the environment distinguishing between objects in the container by their identification `Obj`. The `instance/3` assertions are inserted in the Berkeley database when a new instance of an object `Obj` with class `Class` is created. The `object/4` assertions are inserted in the Berkeley database whenever a new event description is added to the container's state. The event usually changes properties of objects that are registered as C-Logic terms in the container. For example, the state of the social container changes whenever the attributes of the C-Logic term representing the state change in the database. Additionally, time intervals are used to store how the properties of objects change their value in time. Time periods are denoted as `start(e1)` and `end(e2)` terms. The event `Ev` initiates the attribute `Attr` with value `Val` of an Object whose identification is `Obj`. Similarly, when the value of the attribute of an object changes due to a new event, the previous value is stored as having been ended as follows: `object(Obj,Attr,Val,end(Ev))`.

For example, in OPW the assertions below:

```
time(e1, 2).
time(e2, 7).
instance(ag1, agent, start(e1)).
object(ag1, position, [3,4], start(e1)).
object(ag1, position, [3,4], end(e2)).
object(ag1, position, [4,4], start(e2)).
```

describe how agent `a1` moved to position `[3,4]` at time `2` and then moved to `[4,4]` at time `7`. We know that the periods in the state of a container are either closed or open intervals which persist into the future. A new event such as `e2` either starts a new period of time (i.e. `start(e2)`) for a conclusion or ends a period of time which was started by another event (i.e. `end(e2)`). The optimisation is obtained now because the new event is either related to the attributes of objects or the class membership, so we do not need to check all the events that have happened, as with the previous OEC version. Our implementation also uses indexing on the arguments of `object/4` assertions, so that if the first three arguments are specified, the time to retrieve the term is $O(1)$.

By using `object/4` assertions we can store and update the necessary information about the ongoing interactions. We use these clauses to build the state of social and physical containers. Every container, independently if it is social or physical, has an associated Berkeley database where the state of the container is kept.

As explained in Chapter 3, GOLEM implements the AEC for the queries performed in the agent environment on top of the OEC which is used internally in every container. The top-level description of the `holds_at/4` predicate implementation of OEC is specified below:

```
holds_at(Obj,Attr,Val,T):-
    object(Obj,Attr,Val,start(E)),
    time(E,T1), T1 =< T,
    not (object(Obj,Attr,Val,end(Evstar)),
         time(Evstar,T2), T2>T1, T2 <T).
```

The main difference between this OEC version and the one discussed in Chapter 3 is that now we add all new properties that are initiated/terminated as `object/4` (`instance/3`) assertions whenever a new event description is added to the container's state. The distribution of the social and physical state is handled by using the AEC features to query the distributed multiple containers in the agent environment, As described in Chapter 3, the AEC uses the OEC, to query C-Logic like objects and their attributes that may be situated in different containers. The implementation of the AEC clauses presented in Chapter 3 has a straightforward Prolog translation. We omit them here to avoid repetition.

To distribute the physical and social state the following `solve_at/6` has the effect of changing all the physical and social rules to work with distributed containers:

```
solve_at(C, Id, Class, Attr, Val, T) :-
    neighbouring_at(C, [], _, 1, Id, Class, Attr, Val, T).
```

The `[]` list above states that the initial path is empty, the underscore '_', that we are not interested in the resulting path, and the number `1` indicates that we should look at all neighbours whose distance is one step from the current container.

6.4.2 The Propagate Mechanism

The propagate mechanism is defined to link the physical environment to the social environment. Both the `Physical Containers` [19], and the `Social Containers` mediate the interactions of agents in the environment.

Agents perform acts which are defined as `attempts` in the agent environment. When an agent performs an act the `attempt/2` predicate queries the physical state to check that the act is physically possible. If the act is impossible, no changes are made to the physical state nor to the social state of the agent environment. If the act is physically possible, then the `add/3` predicate specified below will update the physical state of the container where the act was attempted and the act will be propagated to the social state of the environment. A multi-threaded implementation of `attempt/2` below is used to perform actions mediated in parallel by the social and physical environment:

```
attempt(Act, T):-
    exec(possible(Act, T), true),
    unique_id(Ev),
    add([Act], Ev, T),
    exec(propagate(Act,T), _)).
```

The above program will be called by an agent to perform an action `Act` which generates an event `Ev` in the system. The `unique_id/1` generates a unique identification for the event. The `exec/2` has a result `R` that returns `true` or `false`). If the event is concluded possible in the physical container, it will be given in input to `propagate(E,T)` predicate which propagates the event in the social environment.

The `propagate/2` predicate first checks if the act is a meta-move that is managed by the Root container (i.e. `new_container, destroy_container`, `suspend_container, resume_container`) and, in this case, propagates the act to the Root container. Any other acts are propagated to the game where the act is being addressed. As we specified in the previous chapter, in some cases, some acts do not address a game in which case the propagation of the act is done in the super-social container of the physical container where the act is first performed. The `propagate/2` predicates are defined as follows:

```
propagate(Ev, T):-
   meta_move(Ev), !,
   instance_of(RID, social_container, T),
   this(C),
   holds_at(C, root, RID, T),
   connector <- send(RID, C, Ev).

propagate(Ev, T):-
   Ev=do(AID, Act, Properties),
   member(attribute(address,GID),Properties),
   GID=game(CID,GameName),
   this(C),
   connector<-send(CID, C, Ev).

propagate(Ev, T):-
   Ev=do(AID, Act, Properties),
   not member(attribute(address,GID),Properties),
   this(C),
   instance_of(CID, social_container, T),
   holds_at(C, super_container, CID, T),
   connector<-send(CID, C, Ev).
```

The above predicates state that given an event `Ev` that occurs in a container `C`, if such event can be classified as `meta_move` then the event is propagated to the Root container `RID`, otherwise the second predicate propagates the event `Ev` to the container `CID`. The agent would have to define in its message the game GID where the act is directed. The games are identified in the system by two variables: the container CID where the game is being played and the name of the game GameName which is being played in the container. The third case, defines a case where the agent does not specify a game, this by default propagates the move to the super social container. In order to send the message to another container, the send method of the connector is used. Every container has its own connector, to use when it is necessary to propagate an act. The "<-" sign between the connector and the send method denotes a Java method call to the to the connector which is defined as a Java Object.

The general assumption with the propagation mechanism is that the creation, destruction, suspension and resumption of compound games are managed from the Root container. This allows us to keep a separation between the structure of the social environment and the actual game interactions between agents. All the moves, independently if they are physical, social or meta-moves, can have implications in the social state (i.e. an agent performing a physically possible move such as moving to a new square, might be doing a prohibited act because in the new square the agent is too close to other agents in the environment. If this happens, the agent may be sanctioned).

To insert object/4 and instance/3 assertions in the Berkley database we implement the add/3. The add/3 predicate is used to insert instance/3 assertions when some actions are performed in the environment for which a new object has to be generated in the state of the container (i.e. a new atomic game), or to insert object/4 assertions when the properties of objects maintained within the state of the container change their value. The add/3 predicate is implemented as follows:

```
add([], Ev, T).
add([P1|Pn], Ev, T):-
 P1 = instance(ID, Class, AttributeList),
 berkeleydb<-putEvTime(time(Ev,T)),
 berkeleydb<-putInstanceOf(instance(ID,Class,start(Ev))),
 add_attributes(Ev,ID,AttributeList),
 add(Pn, Ev, T).

add([P1|Pn], Ev, T):-
 P1 =do(ID,Act,AttributeList),
 berkeleydb<-putEvTime(time(Ev,T)),
 change_attributes(Ev,ID, AttributeList),
 add(Pn, Ev, T).

add_attributes(Ev, ID, []).

add_attributes(Ev, ID, [attribute(Name, Value)|Pn]):-
 berkeleydb<-putOTerm(object(ID, Name,Value,start(Ev))),
 add_attributes(Ev,ID,Pn).

change_attributes(Ev, ID, []).

change_attributes(Ev, ID, [attribute(Name, Value)|Pn]):-
 ID<-getAttribute(Name) returns Attribute,
 Attribute = attribute(Name, OldVal),
 berkeleydb<-putOTerm(object(ID,Name,OldVal,end(Ev))),
 berkeleydb<-putOTerm(object(ID,Name,Value,start(Ev))),
 change_attributes(Ev,OID,Pn).
```

At a given time `T` the above `add/3` predicate can initiate a list of new objects `[P1|Pn]` and their properties `AttributeList` in the database or can change existing attributes of an object within the state of the container. The `putEvTime` and `putInstanceOf` are Java methods which are called to insert respectively `time/2` and `instance/3` terms in the database. Similarly to what we previously explained, the "<-" sign between these Java methods and the instance of the database here identified as `berkeleydb` denotes a Java method call to the to the `berkeleydb` which is defined as a Java Object.

The `add_attributes/3` predicate adds new `object/4` assertions to the database of a container. The `object/4` assertions define the value `Value` of one of the properties `Name` of an object identified as `ID` in the database. The `putOTerm` is the Java method which inserts the `object/4` predicates in the database. Similarly the `change_attributes/3` is used to change the value of an existing attribute of an object in the state of the container. The predicate searches in the database the attribute `Name` of an object `ID` by using the `getAttribute` Java method to query the database. Based on the result of the query, the old value `OldVal` of the attribute `Name` of the object `ID` is terminated and the new value `Value` is inserted in the database using the `putOTerm` Java method.

6.4.3 The Coordination Mechanism

The coordination mechanism refers to the rules that define what happens in the social agent environment when the move of the agent is propagated to the social container. Agent moves can be aimed at a specific game, in which case the effects produced by their acts change the state of the game. Other moves can have effects on the state of a compound game or meta-moves can determine changes at the structure of the social environment by deploying or destroying new social containers.

Once the act of an agent is propagated to the social container, this generates events in the social environment. The act is perceived within the social container which activates the `social_move/2` predicate, specified as follows:

```
social_move(E, T):-
   this(C),
   check(C,E,R,T),
   apply(E,R,T)
```

The predicate `social_move/2` is called in the container receiving a move `E` performed by an agent. The predicate uses the `check/4` predicates to determine if the event is valid and the `apply/3` to apply the changes caused by the act. The `check/4` has two definitions to distinguish between agents acts that are meta-moves in the system and acts performed within a game. The specification of `check/4` predicates is defined as follows:

```
% Meta Moves
check(C,E,R,T):-
   meta_move(E),!,
   (valid(C, E, T),
   R=result(valid, in(C)));
   R=result(invalid, in(C)).

%Moves in a game
check(C,E,R,T):-
   E=do(Aid,Act,Properties),
   member(attribute(address,GID),Properties),
   GID=game(C,GameName),
   holds_at(C, compound_game, Wf, T),
   runs(Wf,Games,T ),
   member(GameName, Games),
   holds_at(GameName, player, Aid, T),
   (valid(GameName, E, T),
   R=result(valid, in(GameName)));
   R=result(invalid,in(GameName)).
```

The first `check/4` predicate checks if the performed act `E` is a meta-move then, using the `valid/3` predicate, determines if the move is a valid one. The implementation of `valid/3` is very similar to the specification presented in Chapter 4. The result R states if the move is valid or invalid in a social container in(C). The second check/4, before checking if the act E is valid, uses the member/2 predicate to find in which game GID the act is directed. The game GID is identified by the container C where the game is created and the name of the game GameName. The description of the compound game Wf running in the social container C is read so that the runs/3 predicate can determine which games Games are running in the container. We check that the specified game GameName is running within the container (this also avoids that from the moment the agent perceives the state of the game to the moment it acts on it, the game has terminated).

Every act within the social environment can cause some changes to the social state of the environment. We use the apply/3 predicate to define how the act, depending if it was valid or not, changes the state of the container or the state of the game where it was performed. The apply/3 predicate is implemented as follows:

```
apply(E, R, T):-
   R=result(valid, in(G)),
   effects(G,E,T),
   new_state(G,T).

apply(E, R, T):-
   R=result(invalid, in(G)),
   sanction(G,E,T),
   new_state(G,T).
```

The above predicates define the effects that an act E has within the state of a game G depending on the action E being valid or not. If the act was a valid move in the game (or container in case of meta-moves), the apply/3 predicate calls the predicate effects/3 to apply a set of changes to the state of the game G. If the act was invalid invalid the predicate sanction/3 is called to apply other changes to the state of the game. The two predicates, `effects/3` and `sanction/3`, can change the state of the game `G` by changing some properties of the state or, in some other cases, new events are generated which can update the state of other games or containers. The `new_state/2` predicate is called to maintain the state of a game as an attribute of the game `G`. The use of this predicate becomes explicit in Section 6.5.3.

6.4.4 The effects of Meta-Moves

Using the `effects/3` predicate explained in the earlier section, we can specify how meta-moves change the state of the social environment. Using the meta-moves, agents can choose to create a new complex game. When the interactions within games became obsolete, either due to termination of the game or due to exceptions, the complex game may be suspended to be resumed later or even canceled. The `effects/3` for the creation of a new interaction game within a container is defined as follows:

```
effects(Root, E, T):-
    E=do(A, new_container, [attribute(game,GameName)]),
    create_container(GameName, GID, T),
    unique_id(Ev),
    GID=game(GameName, CID),
    add([instance(CID,container,[attribute(game, GID),
                  attribute(cycle, started)]], Ev, T).

create_container(GameName, GID, T):-
   url_specification(GameName, Url),
   generate_id(Container)
   GID=game(GameName, Container),
   java_object('container.Container',
              [Container,GameName,"Ontology.wsml",Url],
              Container).
```

The first predicate checks that the performed act is the `new_container` meta-move. The predicate `create_container/3` is called to start a new container containing the game specification requested by the agent. The `add/3` predicate is called to create an additional event. Such event triggers the addition of the new created container and the information about the game it contains, as a new object with some properties (such as the game contained in the new social container) in the database. Similar predicates are defined also for interleaving of atomic games. The rest of the `effects/3` predicates defined to capture the changes caused by meta-moves in the structure of the containers and in the atomic games that are played in a social container are defined in Appendix B.

6.5 Implementation of Atomic Games

Once we have the general propagation and coordination mechanism in the social container, we can define the atomic games. The atomic games incorporate the normative rules of the application in the agent environment. They describe rules, procedures or protocols in terms of an atomic interaction which can be followed by the players to achieve some specific results. The combination of the atomic games in a compound game adds flexibility to the normative approach and allows agents to change the rules by deciding the games they play. The specification of the compound game is done in terms of workflow coordination patterns which, once instantiated in a social container, provides the mechanism for changing the interaction rules.

To implement an atomic game we need to determine what are the valid acts (including permissions, prohibitions, obligations and empowerment) that agents can perform and at what state of the interactions these acts are valid, the effects of these acts in the state of the game and what sanctions or rewards (if any) we apply to respectively not valid acts and to valid ones. We also need to describe the initial state of the game and the terminating state.

6.5.1 Social Rules

The implementation of the valid/3 predicate, the permission, prohibition, obliged and empowered rules is similar to their specification which we presented in Chapter 4. In general, these rules are application dependent. For completeness we show an example of implementation of the `permitted/3` and `forbidden/3` predicates:

```
permitted(G, Move, T):-
   empowered(G, Move, T),
   not forbidden(G, Move, T).

forbidden(G, Move, T):-
   Move=do(AgentA, move, [Properties]),
   member(attribute(position, PosA),Properties),
   holds_at(CID, sender, Move, T),
   PosA= [X,Y],
   adjacent([X,Y], [J,K]),
   [J,K] = PosB,
   solve_at(CID, AgentB, agent, position, PosB, T),
   not AgentA = AgentB.
```

The `permitted/3` predicate specifies that all the actions that are not forbidden, are permitted actions. The `forbidden/3` shows an example of a forbidden action. The predicate defines that moving into a position that is adjacent to another agent is a forbidden act. To implement the predicate, we need to be able to identify agents in adjacent positions. This information is held in the physical state of the physical container from where the move `Move` was propagated. When we send a message to a container, we identify also the sender of the message which is temporarily stored in the state of the container receiving the act `Move`. Thus, we can query the state of the physical container by using the `solve_at/6` predicate.

Atomic games define very specific interactions in terms of application dependent rules. The `effects/3` predicate defines the effects a move has on the state of a game and it is specified as follows:

```
effects(G, do(AID, enter, [Properties]),T):-
    member(attribute(game,GID), Properties),
    member(attribute(role,picker),Properties),
    not instance_of(AID,agent,T),
    unique_id(Ev),
    add([instance(AID,agent,[attribute(plays, GID),
         attribute(role, picker)])],Ev,T),
    add_attributes(Ev,GID,[attribute(player, AID)]).
```

The above `effects/3` predicate changes the state of the container due to a valid `enter` act performed by an agent identified as `AID`. The agent `AID` is a new instance for the database of the container. The state of the container is changed by adding the new agent `AID` as a player of the game `GID` in the role of `picker` and by updating the multiple valued attribute `player` of the `GID` game with the new player.

6.5.2 Enforcement Policies

When a not valid action occurs, the effect in the state of the social container is to create sanctions which will then be enforced by changing some other properties in the state of the environment. In OPW we have chosen to define points for the agents which will increase as effect of valid acts that are particularly desirable in the system. For example, acts such as dropping a packet in the destination is desirable in the system specifying OPW because it brings the agents closer to deliver all the packets that are around the grid. We also decrease the points of the agents if they perform actions that do not comply with the defined social rules. An example on how we implement a sanction within the system is specified below:

```
sanction(G, Move, T):-
   Move=do(AID, drop, Properties),
   holds_at(CID, sender, Move, T),
   solve_at(CID,AID, agent, points, Value,T),
   NewValue is Value+3,
   append[(attribute(game, game(CID, reply))], Properties,P1),
   append([attribute(agent, AID)], P1,P2),
   append([attribute(points, NewValue)], P2,P3),
   this(C),
   propagate(do(C, sanctioned, P3),T).
```

the above predicate is called when a not valid `drop` move occurs. A drop move of a packet in a position that is not a destination point is forbidden in OPW. This act is sanctioned by reading the points of the agent `Value` in the physical container where the agent is situated and by removing three points as a sanction for such act. The new points `NewValue` of the agent are added in the same database of the physical container. We use `propagate/2` to send the new event to the container. The container then uses the `add/3` predicate to add the new information in its database.

Rewards can be also defined as part of games. We use the same `effects/2` predicate to reward agents points. This is because `effects/2` is activated when valid acts occur. For example, the following predicate:

```
effects(G,Move,T):-
    Move=do(AID,drop,Properties).
    holds_at(CID, sender, Move, T),
    solve_at(CID, AID, agent, points, Value, T),
    NewValue is Value+10,
    append([attribute(game, game(CID, reply))], Properties,P1),
    append([attribute(agent, AID)], P1,P2),
    append([attribute(points, NewValue)], P2,P3),
    this(C),
    propagate(do(C, rewarded, P3),T).
```

changes the points of an agent `AID` because it has performed a valid `drop` act which means it has dropped a coloured packet into a destination with the same colour.

6.5.3 Social Container Interaction

The social container interaction refers to the procedure that agents can use to query the social state. There are many possible implementations on how agents can query the social environment. Since the request to observe the social state is generated in the physical containers where the agents are located, we reuse the same propagation mechanism to propagate the agents query in the social containers. To query the social state agents activate the `observe_game/3` predicate in the physical container where they are situated. This has as result that the agent act is propagated as a query to the social environment. Within the act the agent performs, it also specifies its identity `AID` and the game that needs to observe `GID`. The `observe_game/3` predicate in a physical container is defined as follows:

```
observe_game(AID,GID,State):-
   current_time(T),
   GID=game(CID, GameName),
   solve_at(CID, GameName, game, player, AID, T),
   solve_at(CID, GameName, game, state, State, T).
```

The above predicate specifies that the `observe_game/3` predicate queries the database of the social container about the state of the game specified as

GameName if it finds in the same database that the agent who is trying to observe the state is a player in this game.

The attribute state [1] of a game identified as GameName is a complex structure, the attributes of which can change as players of a game perform acts in it. In order to maintain a coherent value State of the attribute state we need to recalculate it at every act performed by a player within a game identified as GameName. To do this, in the we call the predicate new_state/2. This predicate can be implemented differently depending on the atomic games implemented and the application needs. A general way to implement the predicate is as follows:

```
new_state(Game, T):-
   holds_at(Game, cycle, C, T),
   findall(F, forbidden(Game, F, T), Forbidden),
   findall(V, valid(Game, V, T), Valid),
   findall(Player, holds_at(Game, payers, Payer,T), Players),
   holds_at(Game, result, R, T),
   State=state(C, Players, Forbidden, Valid, R),
   unique_id(Ev),
   add([do(Game, new_state, [attribute(state,State)])],Ev,T).
```

The new_state/2 predicate calculates all the latest information on the state and asserts it in the database as an attribute of the Game. In particular, it finds in what cycle C the game is (i.e. started, suspended, terminated), calculates the forbidden Forbidden and valid Valid moves within the game Game. It also identifies the result R if any has been reached within the game. Finally, the new sate is added to the database by using the add/3 predicate.

The operation of observing the state of the game we presented can be expensive computationally. With a similar approach we can implement additional

[1]To not confuse with the attribute cycle which identifies if the game is in a started, suspended or terminated life-cycle.

observations for the agents that are less computationally expensive. For example agents can observe only a subset of the properties in the state of games.

6.6 Summary

In this chapter we show how we implement the MAGE platform on top of GOLEM. We start with the GOLEM reference model which shows the GOLEM architecture. We use the GOLEM reference model to extend it with the MAGE framework. In particular GOLEM and MAGE relate to each other through the structure that is created between the physical containers and the social containers. By connecting social containers and physical containers we define an inter connected structure regulating the agents actions using both social and physical rules. Actions performed by the agent that is situated in a particular container of the physical environment, are propagated to the social environment which applies social rules to evaluate the consequences of such acts.

Whenever an agent interacts within a game, the agent identifies the game where the act is aimed to be performed. Due to this implementation choice, the acts that are performed in physical spaces, are directly propagated to the social container where the game is contained. The propagation mechanism is included as a way to deal with the dynamic social structure of the social environment but also to hide the complexity of interactions to agents and to deal with distribution in a scalable way.

Agents are allowed to configure the structure of the environment by creating new containers containing complex games using the a set of meta-moves. The meta-moves are also used to act on the games' cycle. Complex games coordinate many atomic games which can evolve in different ways depending on agents actions. We defined the coordination mechanism as the cycle that is defined within containers to deal with the evaluation of acts and their effects.

We showed how normative concepts are implemented as part of the atomic games. Changes due to valid or invalid moves are added to the database of the container. Some acts change properties of the physical state rather than the social state. In this case, we propagate the effects that the act has to the physical state which is then updated accordingly. Agents can perceive all these changes by observing how the state evolves. Agents can perceive the state of the physical container where they are situated and the state of games where they are players (with some exception). They can consequently decide how to act in the environment based on the perceived information.

In the next chapter we show how the MAGE approach is applied within ARGUGRID. We illustrate this by defining game interactions within the Earth Observation Scenario where agents can negotiate agreements about particular services on the fly.

Chapter 7

The ARGUGRID Platform

In the previous chapter we showed the implementation of the MAGE platform. We illustrated how the GOLEM containers can be extended to deal with a dynamic social environment and how agents can act and observe the social environment.

In this Chapter we show how we used MAGE in the ARGUGRID project. ARGUGRID aimed at providing argumentative agent technology for service selection and negotiation over the GRID. More specifically, we first illustrate how we can specify an atomic game to support agent negotiation and we then describe how such a game can be included as part of the VO formation process among agents. We also show how we can define compound games that agents can use to interact within the VO.

The reminder of this Chapter is organised as follows: In Section 7.1 we introduce the aims and objectives of the ARGUGRID project. In section 7.2 we describe the Earth Observation scenario, used throughout the project to show the functionalities of the platform. In section 7.3 we describe the minimal concession protocol which has been used for the negotiation process amongst ARGUGRID agents. In section 7.4 we then show how, by using MAGE, agents can interact to form VOs. In section 7.5 we explain how the agents negotiate about GRID services for the Earth Observation Scenario. Section 7.6 summarises and concludes the work presented in this chapter.

7.1 The ARGUGRID Project

The ARGUGRID project is an European project [3, 120] aimed at developing argumentative agents [41, 42] to support reasoning and decision making about dynamic composition of GRID based services. ARGUGRID users are provided with a user interface where they can describe the services that they are interested to receive. Users are represented with user agents in the agent environment. When a user agent receives a request by the user, it searches for the set of services that satisfy the requirements of the user. Other agents in the platform represent service providers and the corresponding services they offer to interested users. In this context, user agents contact service provider agents to establish an agreement about the provided services and interact according to the rules that govern these agreements and additional rules specifying the application.

In ARGUGRID, MAGE and GOLEM were used to model the agent environment of the application. On one hand, GOLEM allows to define GRID services as separate resources from agents. On the other hand, the additional MAGE layer allows for flexible agent interactions. Within MAGE agents can simultaneously participate in more than one interaction with other agents via social games. At any time, during a social game, agents can decide the best strategy for their moves. Having MAGE mediating the interactions and GOLEM to support the agents to find and use the resources, we remove responsibilities from the agents to check their moves thus they can reason and make decisions about how to best achieve their goals.

Together with GOLEM and MAGE additional technologies were used to build ARGUGRID:

- Inforsense KDE [62], a commercial tool for integrative data analysis, visualisation, and service composition for decision-making users. The KDE software allows ARGUGRID users to define workflow requirements and translate them to high level goals for agents using an additional module named ARGUbroker [89].

- GRIA is a service-oriented infrastructure designed to support B2B collaborations through service provision across organisational boundaries [5]. GRIA was used in ARGUGRID to store and partially monitor (limited monitoring capabilities were available in GRIA) SLA templates which will result from agents interactions and the confirmation of the user of the service. The SLA would then be stored in GRIA servers and used during service execution.

- Platon [78] a peer-to-peer platform designed to support service discovery and load balancing in ARGUGRID.

- Margo [87] is an engine defined for argumentative reasoning. It was used in ARGUGRID to define the decision making mechanisms of agents. In particular, Margo defines modules for reasoning with what moves to play within a game.

In Fig. 7.1 we show the global picture of the ARGUGRID platform. The components of the ARGUGRID platform are distributed among computers residing in distinct locations and connected to the global Internet. ARGUGRID Agents may act as service requesters or as service providers (or both). Fig. 7.1 shows the platform from the service requester's point of view. The main interactions that are involved when addressing the requests that users perform in the system are the following ones:

1. An ARGUGRID User interacts in the platform by submitting an abstract workflow to the KDE. The abstract workflow is realised through the KDE workflow editing tool and reflects, at a high-level, the user

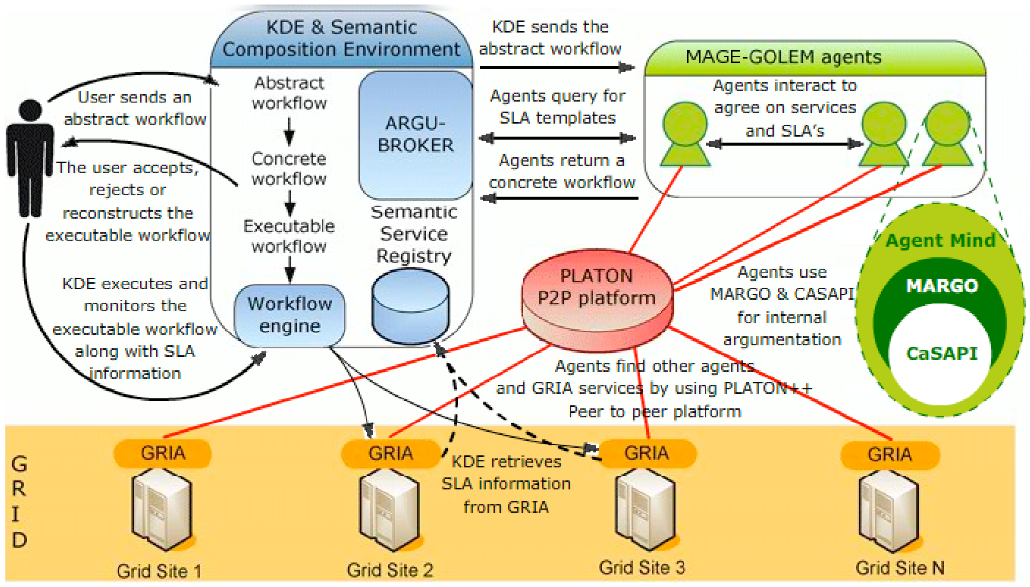

KDE & Semantic Composition Environment
KDE sends the abstract workflow
MAGE-GOLEM agents
User sends an abstract workflow
Abstract workflow
ARGU-BROKER
Agents query for SLA templates
Agents interact to agree on services and SLA's
The user accepts, rejects or reconstructs the executable workflow
Concrete workflow
Agents return a concrete workflow
Executable workflow
Semantic Service Registry
Agent Mind
MARGO
CaSAPI
KDE executes and monitors the executable workflow along with SLA information
Workflow engine
PLATON P2P platform
Agents use MARGO & CASAPI for internal argumentation
Agents find other agents and GRIA services by using PLATON++ Peer to peer platform
GRID
GRIA
GRIA
GRIA
GRIA
KDE retrieves SLA information from GRIA
Grid Site 1
Grid Site 2
Grid Site 3
Grid Site N

Figure 7.1: ARGUGRID global picture.

requirements. The KDE refines the abstract workflow so that its instantiation can be delegated to MAGE-GOLEM agents.

2. After the KDE communicates the abstract workflow to the agent that represents the user within the platform. The agent searches which GRIA services should be used in order to derive a concrete workflow, to be executed on the GRID. The agent mind is composed by a MARGO argumentation engine for decision-making, which in turn uses the CaSAPI general-purpose argumentation engine. The agent mind defines a way for the agent to reason about services and make decisions on how to deal with the abstract workflow. To interface the mind of the agent and the environment, we have defined the Social Interaction Module (SIM). SIM allows the agent to reason about the sequence of acts to perform in order to solve its goals and allows it to perceive the state of MAGE/GOLEM environment and to act on it.

3. Using the SIM, the agent starts the Selection Process where the searches for appropriate service provider agents that can provide the services it needs. A registry is provided inside GOLEM that encapsulates the PLATON++ P2P platform. Agents query this registry to find agents providing the requested GRIA services. The query may produce many agents providing the same service. Argumentation is used to select the preferred agents to start negotiations.

4. The second phase of agent interaction is The Negotiation Process where agents can negotiate with other agents, sign contracts with them and form Virtual Organisations (VOs). In this part of the system, MAGE plays a crucial role to support agent interactions. The user agent creates a new negotiation game and invites the service-provider agents that has selected for negotiation. If the service provider accepts the invitation, the agents exchange messages to first agree on their role, then negotiate about the service and if they reach an agreement they can sign it. These interactions are defined as combination of many dialectical protocols that structure the negotiation game.

5. When an agreement has been reached, the user agent has selected the services that makes the abstract workflow a concrete one. Each concrete service has a SLA template which is stored within the GRIA platform. The relevant SLA templates are completed according to the agreement performed amongst agents.

6. The abstract workflow defined by the user is now a concrete one. The concrete workflow is sent to the KDE. The concrete workflow contains a set of GRIA services to be executed in a certain manner/sequence and the negotiated SLAs of the services within the workflow.

7. The KDE uses the capabilities of the ARGUbroker component to change a concrete workflow into an executable one. The executable workflow with an associated set of SLAs is presented to the user which has the

choice of accepting the executable workflow, rejecting it or deciding to modify the abstract workflow for a better solution. In the latter case, the abstract workflow will be given again as input to the KDE, repeating the previous steps until the user either accepts or rejects the executable workflow solution. In the case of acceptance, the system will follow the step below.

8. The workflow engine within the KDE will use its workflow execution service to send the execution workflow, along with its related SLA information, for execution on the GRID infrastructure, running the GRIA GRID middleware. Upon successful execution of the final executable workflow, the user is informed and the execution results/data are returned back to the user via the KDE interface.

To make the functionalities of ARGUGRID more explicit we focus on the Earth Observation Scenario.

7.2 The Earth Observation Scenario

The ARGUGRID Earth Observation scenario considers a ministry official (the user) requiring data about the detection of an offshore *oil spill* [119]. In this scenario three organisations interact to create a virtual organisation that offers to the user an oceanic oil-spill detection service.

The abstract and high-level goal of the user is provided by the user utilising the Inforsense KDE tool. The ARGUbroker GUI translates the requirements of the user to an agent goal and passes them to the user agent as illustrated in Fig. 7.2. This goal cannot be immediately satisfied by the user agent itself but it requires the help of satellite companies that observe parts of the earth at different days. These companies publicise their services in ARGUGRID using their service provider agents to manage the services. In this scenario, given the abstract request of the official, the user agent tries to instantiate it in a detailed set of services that can be invoked in sequence to provide the requested information.

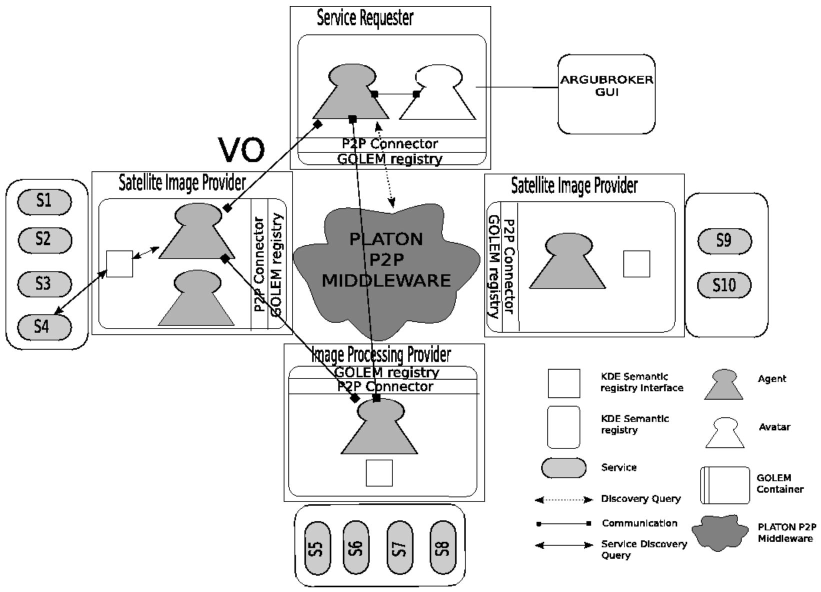


Figure 7.2: Earth observation scenario in ARGUGRID [119].

There are two types of organisations needed in order to offer oil spill detection: image providers and providers of data processing algorithms. The image providers are organisations that control a variety of satellites with various orbits, capabilities and costs where the providers of data processing algorithms are companies that take the raw image data from the satellites and provide a variety of services from simple format transformations to complex high-level identification of oil spills. The agent has to solve three problems: The first problem is the Selection Problem. There are many satellite companies providing different services, each with different capabilities and costs, and one satellite may be more appropriate than another given certain conditions that the ministry sets. The user agent, based on a set of preferences over the service requested, selects the suitable satellite companies and engages in a contract negotiation process with provider agents to create a VO that will instantiate the lower level services required to meet the official's request.

The second problem is the *N*egotiation Problem. Every service has a set of negotiable properties such as price, resolution and delivery time. As an example, possible satellites may be Envisat, ERS-1 and RADARSAT. Each satellite provides different services, each with different capabilities and costs, and one satellite may be more appropriate than another given certain conditions. For example:

- Envisat is an optical satellite with a swath of 1150km and an orbit frequency of 3 days.
- ERS-1 is a satellite with a radar sensor with a swath of 500km and a frequency of 3 to 168 days.
- RADARSAT is a different kind of radar sensor with a swath between 45-500 km and a frequency of 3-24 days.

The third problem is the *C*omposition Problem. Whichever satellite is chosen, the raw images from the satellites are not sufficient for the detection of oil spills. The ministry must find companies that offer post-processing of raw satellite images. The ministry may delegate this post-processing to the satellite company or find a different company to provide this extra service. For example, some of the possible post processing services are:

- A format conversion which does further processing of the satellite image to convert it into one of many formats such as: CEOS, HDF, TIFF, JPEG, etc.
- A re-projection which takes a satellite image file to reproject it into a different coordinate system depending on the needs of the user.
- An oil spill detection which uses different pattern recognition algorithms, to detect ships, buildings, oil spills.

Since the ministry is only interested in oil spill detection, it contacts a company which is able to provide the oil-spill detection as a post-processing service. Once the image is received from the satellite company, the ministry sends the image file and expects the company to return images with the oil spill identified.

The GOLEM affordances and the peer-to-peer platform, combined with argumentative reasoning handle the selection problem. The negotiation and composition is handled using MAGE to support the negotiations between agents and using agent reasoning so that every agent decides the best course of actions to achieve their own goals. To mediate the interactions we define the vo_formation compound game. This game is composed by four main components. The core of the negotiation process is defined by the minimal concession protocol which is defined as an atomic game within MAGE. Three other atomic games are defined as part of the negotiation process, one is the role game for agents to agree about their roles within the VO, the second one is the sign game that is used from the agents to give the final agreement. Finally, the general rules for ARGUGRID interactions are defined within the argugrid game.

7.3 The Minimal Concession Protocol

To illustrate how we define atomic games for ARGUGRID we use a bargaining protocol named minimal concession protocol. The minimal concession protocol (mcp) was defined by Dung et al and it was formally described in [43]. The protocol enables agents to reach agreements about services or products by deviating minimally from their optimal bargaining positions. To do so, in conjuction to the mcp protocol, agents are expected to follow a minimal concession strategy.

A minimal concession strategy requests that the agents start the bargain with the optimal properties for the service/product that they offer or search. Every opponent should concede minimally if the opponent has conceded in the previous step [1], and, stand still if the opponent stands still in previous step. An agent will consider if it can concede on a property of the service/product depending on its bargaining power. Afterwards the agent will expect the other agent to concede as well, thus, allowing the offer to get closer to match the request and vice versa. If the agent decides not to concede, it can stand-still. An important property of this protocol is that if the two agents use the protocol in conjunction with a minimal concession strategy, then every negotiation terminates successfully and the minimal concession strategy is in symmetric Nash equilibrium [43].

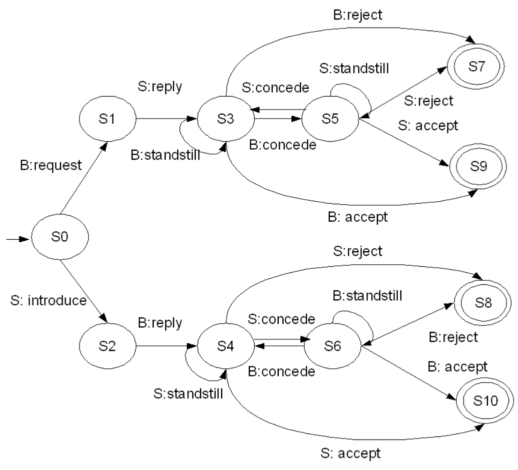


Figure 7.3: The *Minimal Concession* protocol [43].

The mcp is shown in Fig. 7.3. The protocol provides the following set of locutions available to agents: *request, introduce, reply, concede, standstill, accept, reject.* The protocol assumes two agent roles, a *buyer* (B) and a *seller* (S). The avialable locutions change depending on the agent's role and on the state the protocol is in. The protocol starts in *S0* with an *introduce*

[1]Or if it is making the third move in the mcp protocol. This is because in the mcp protocol the first two moves can be either request and reply or offer and reply. The third move, if it is not accept or reject which concludes the agreement, should be concede to follow the minimal concession strategy. This is illustrated also in Fig. 7.3

move made by the seller or with a *request* move made by the buyer. These moves are used to respectively request or introduce an offer e.g. an oil spill detection service with some properties. Afterwards, a *reply* move can be made from the buyer to reply to an *introduce* move (*S2*), or from a seller to reply to a *request* move (*S1*). After this move, *standstill, reject* or *concede* are all moves that can be made by any of the two roles (*S3, S4, S5* and *S6*). The *accept* move terminates successfully the protocol (*S9* and *S10*) and the accepted offer is considered the value of the result of the game. Three consecutive *standstill* moves are considered as a *reject* move, which terminates the protocol with no agreement (*S7* and *S8*).

7.3.1 Starting MCP

To start the mcp game, within MAGE a meta-game is defined which manages the creation and destruction of games within social containers. All the interested agent has to do, is to perform a start_game move and specify the game and the players that will be participating in it. The new game is created as effect of this move using the happens/3 predicate for the creation of a new game [2]. After the creation of the mcp, the moves of the two players will change the state of the mcp game. As described in Chapter 3, the agents moves (such as starting the mcp game) create events in the social container. The start_game move initiates the game via an assigns/3 assertion. The assertion:

```
assigns(Ev, Id, min_concession)←
    Ev[act ⇒ new_game, game ⇒ min_concession, properties ⇒ P].
```

The move allows the creation of an instance for the minimal concession protocol. To complete the instantiation process we also need to specify the initial

[2]As specified in Chapter 5.

values for the attributes of the complex term representing the minimal concession protocol. For this we need to define separately the initiates/4 rules as the one below:

```
initiates(Ev, Id, party_of, Val)←
    Ev[act ⇒ construct, protocol ⇒ min_concession, parties ⇒ agent: Val].
```

Additional initiates/4 clauses are needed to define the initial state. The initial state of the game will evolve as a result of moves been made in the state of a game.

From the implementation perspective, the minimal concession protocol is started using the coordination primitives defined in Chapter 6. The move produces the desired effects when the mcp is considered in the system as a game that satisfies the running conditions (the runs/3 predicate checks within the compound game state when games satisfy the compound game specifications) and that the move is valid. To define the validity of this action, we empower agents holding an initiator role to start the game. The initiator role is given to the user agent when it requests to start the vo_formation game. The argument behind this choice is that the requests start from the user and the user agent is the first agent wanting to form a vo. The same user agent can start a new mcp specifying the service-provider agent with which to negotiate.

7.3.2 The mcp State

The state of the mcp was represented using a complex term of the form:

```
min_concession:mc1 [
   players⇒ {agent:a1 [role ⇒ seller], agent:a2 [role⇒buyer]},
   buyer_position ⇒ offer:o1 [price ⇒80, resolution⇒20, delivery ⇒2],
   seller_position ⇒ offer:o2 [price ⇒100, resolution⇒20, delivery ⇒2],
   standstill_count ⇒ 1,
   result ⇒ nil
].
```

The state is identified by mc1 denoting an instance of an object whose class is the minimal concession protocol with two participating agents a1 and a2, whose role attribute is seller and buyer. The buyer, in the previous round, has made an offer o1 (a complex term), while the seller has made another offer o2 (another complex term). There is one standstill move that has been encountered, and the result of the interaction is still incomplete as the value is still nil.

7.3.3 The mcp Valid Moves and Effects

To specify valid moves, we specify when moves are permitted. For example, we specify when a request move is legal in the minimal concession protocol as:

```
permitted(Id@T, Move) ←
   instance_of(Id, min_concession, T),
   speech_act:Move[actor ⇒ A, act⇒request, offer ⇒ Product, role⇒ buyer],
   holds_at(S, agent_of, A, T),
   holds_at(A, role, buyer, T).
```

Once a move is determined as valid, a new protocol state is brought about due to the move's effects. We are assuming that, to handle the effects of a move in the protocol, we use the definition of effects/3 predicate as presented in Chapter 4. In our representation of state, once an event has happened, its effects are added to the state implicitly, via inititiates/4 definitions that initiate new values for attributes of a state term, terminates/4 clauses that remove attribute values from a state term, and assigns/3 definitions for assigning to objects new instances of terms. An example, of how new values are initiated for attributes for the minimal concession protocol is given below:

```
initiates(Ev, Id, seller_position, Offer)←
    happens(Ev, T),
    instance_of(Id, min_concession, T),
    Ev[act ⇒ Act, actor ⇒ Aid, role ⇒ seller, offer ⇒ Offer],
    changes_seller_position(Act).

changes_seller_position(introduce).
changes_seller_position(concede).
changes_seller_position(reply).
```

The above definition initiates the current position made by a seller to be stored in the state of the game as a result of a request, reply or concede move. The old offer is terminated and substituted by a new request because of the way the OEC is specified.

7.3.4 Final state of mcp game

The state of the mcp game will eventually reach the final state from which we can extract the game's result. We specify this via terminating/2 predicates. For example, the definition:

```
terminating(Id@T, Result)←
      instance_of(Id, min_concession, T),
      holds_at(Id, result, Result, T),
      not Result==nil.
```

specifies the conditions under which the minimal concession protocol terminates and at the same time returns the result. Not every interaction is successful, thus it is possible to define termination of games due to exception, such as there is not a sufficient number of players in the game.

```
terminating(Id@T, Result)←
    instance_of(Id, min_concession, T),
    holds_at(Id, standstill_count, Value, T).
    Value>3.
```

The above predicate specifies that the minimal concession protocol terminates if it holds in the state of the game that the standstill_count value is greater than 3.

7.4 VOs in ARGUGRID

The minimal concession protocol we described in the previous section is only a component of the more complex activities that enable agents in ARGUGRID to participate in VOs. In fact, an interesting feature of our framework is that we can specify the interactions beyond the mcp by defining coordination mechanisms which at run-time change the specification of how the games are coordinated in a plug-and-play style.

7.4.1 The VO Life-Cycle as a Compound Game

The Fig. 7.4 shows the VO life-cycle in ARGUGRID. The VO life-cycle illustrated here is based on a reinterpretation of the existing literature [56, 95] about the VO life-cycle in terms of a compound game.

As defined in Chapter 3, the VO life-cycle can be structured in three main phases: formation, operation, and dissolution. During the formation phase, agents negotiate the terms of their formation within a negotiation game, here we include the mcp protocol for agents to agree on the terms of the services they require.

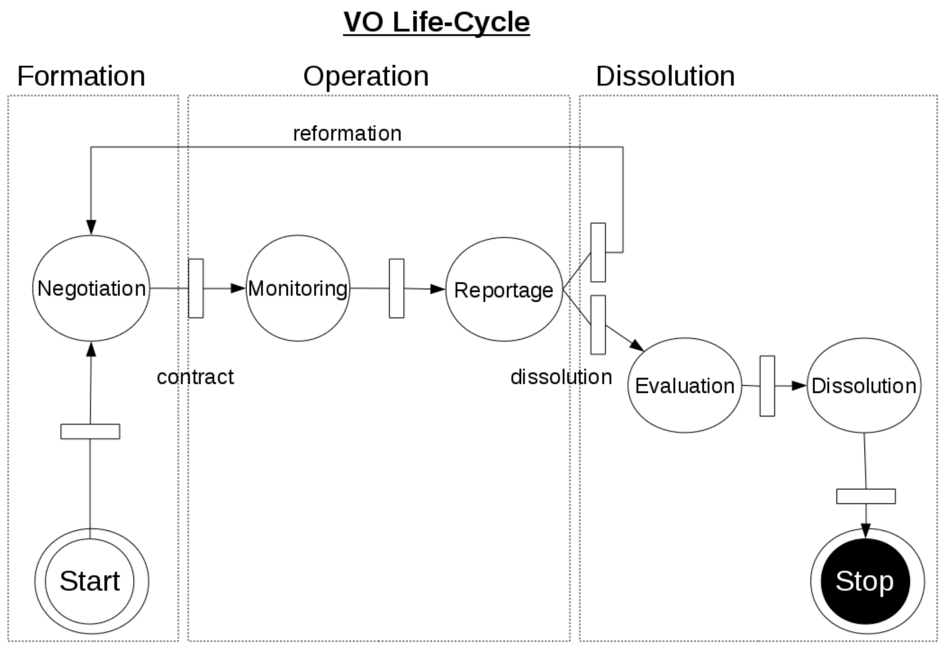


Figure 7.4: The *VO Life-cycle* in ARGUGRID.

The second phase defines monitoring mechanisms as a monitoring game where the agents actions are checked to make sure that the execution of the agreements taken in the formation phase are satisfied. The monitoring game here follows in sequence after the negotiation game. After the monitoring, the operation phase follows a reportage game interaction during which agents can identify problems encountered during the execution of their agreements.

Afterwards, in the dissolution phase, the xor_split primitive allows us to consider two cases: The first case is when it is necessary to go back, and renegotiate the agreements or improve them. The second case is when a dissolution is required to terminate the agreements. In the first case, a reformation can be enacted by restarting the VO from the negotiation game. In the second case, the VO can be terminated after an evaluation game to determine the performances of the agents followed by a dissolution game which results in the termination of the whole VO life-cycle.

The VO life-cycle in ARGUGRID was focused in the formation phase. In principle, it is possible to define monitoring games by defining normative relations in a game whose initial state is defined using the agents agreements from the formation phase. However, in ARGUGRID, the Operation phase of the VOs was handled within the GRIA platform. As a consequence, also the dissolution process becomes trivial as it only requires the user-agent to perform a meta-move to request the destruction of social container containing the VO interaction. The remaining activities of monitoring, execution, reportage, evaluation, and dissolution, can be modeled with similar concepts to the ones we illustrate in this Chapter. In the remainder of this chapter we focus on how to model the control flows of activities as a complex game defining the negotiation game.

In [83], McGinnis et al identify five phases for the VO formation. These phases were integrated within the GOLEM/MAGE platforms and they help us identify the required activities to complete the formation process. The 5 phases are realised as follows:

- In the Initiation Phase given a user's query as an abstract workflow, the user agent identifies the sub-goals it has to achieve. This phase requires the user agent to internally reason about the sub-goals the agent is trying to achieve, find if they can be achieved with the services the same agent offers (if any [3]) and identify what sub-goals cannot be achieved by itself. The agent can search the missing services to solve its goal in the GRID. In ARGUGRID, the agent that starts to interact in a VO will assume an initiator role.

- In the Discovery Phase the agent identifies potential partners. This phase involves querying the physical environment about specific service provider agents (i.e. agents that provide oil spill detection services).

[3]Service providers can become service requesters and vice-versa.

- In the Selection Phase the agent selects which are the most suitable services to satisfy the abstract workflow. Agents advertise their services and a range of properties for these services (i.e price range, delivery time, and other properties relevant to the particular service) within GOLEM. The agents can query the registries from the physical container where they are located. The query can provide many agents therefore the SIM module activates the argumentative module of MARGO. In this module, the various properties of the services are compared to decide for the agents to contact. The goal may require more than one different service to be composed together, therefore the agent can decide for a set of agents to interact with.

- In the Role Establishment Phase the agent starts to interact with other agents. This phase is handled within MAGE with a role game that allows agents to agree on their roles. The agents involved in the VO formation can perceive the state of this game and decide to act accordingly. Initiator agents are permitted to create new vo_formation games where they can invite service provider agents to negotiate about specific services.

- In the Negotiation Phase agents have agreed their roles and can start negotiating about a service. This phase is realised by defining the mcp game for agents to follow in order to find an agreement. In this phase we include an additional sign game for agents to sign their agreement.

7.4.2 VO Activities as Complex Games

To obtain complex interactions, such as the one required for the formation of a VO within ARGUGRID, we specify four atomic games, role, mcp, sign and argugrid, as it is shown in Fig. 7.5. We combine these three games using the workflow primitives explained in Chapter 5. By using these primitives we want to represent the formation process in such a way that, even if the agents fail to conclude one of the VO formation phases successfully, they still can interact within the overall process. Using these games an agent is able to interact with the other agents during the Role Establishment Phase and the Negotiation Phase. The Initiation, Discovery and Selection Phase involve respectively the agent querying the physical containers, the agent performing internal reasoning about potential partners and selecting a partners to negotiate about VOs. These phases do not include agents interacting with one another.

In Fig. 7.5 we show the vo_formation process as a complex game. In this game we combine the four different sub-games: the role sub-game for the agents to determine their roles, the mcp sub-game for the agents to agree on the terms of the contract and finally the sign sub-game where agents sign their agreements. At any point in these three games the agents can reiterate the game starting from the role sub-game. In parallel to these three games there is what we call the argugrid sub-game where we define general rules of interaction such as rules to allow interacting agents to request new agents to join the negotiations or to specify when it is permitted to leave the negotiation process. We can check that the agents respect these rules in parallel to the other games that are being played to form a VO.

The game represented in Fig. 7.5 is the graphical representation of the vo_formation compound game. Compound games can be combined in different ways. We can change the compound game by either defining different patterns and combinations amongst such games or by changing the conditions that allow these games to be activated. The C1-C7 bar-lines represented in Fig. 7.5 define a list of conditions that should hold before a new game can be started.

By defining argugrid, role, mcp and sign sub-games as atomic games we can provide a modular development approach in that we can change the rules of one atomic game without worrying about the combined interactions as these games are independent to each other. We then add basic workflow patterns

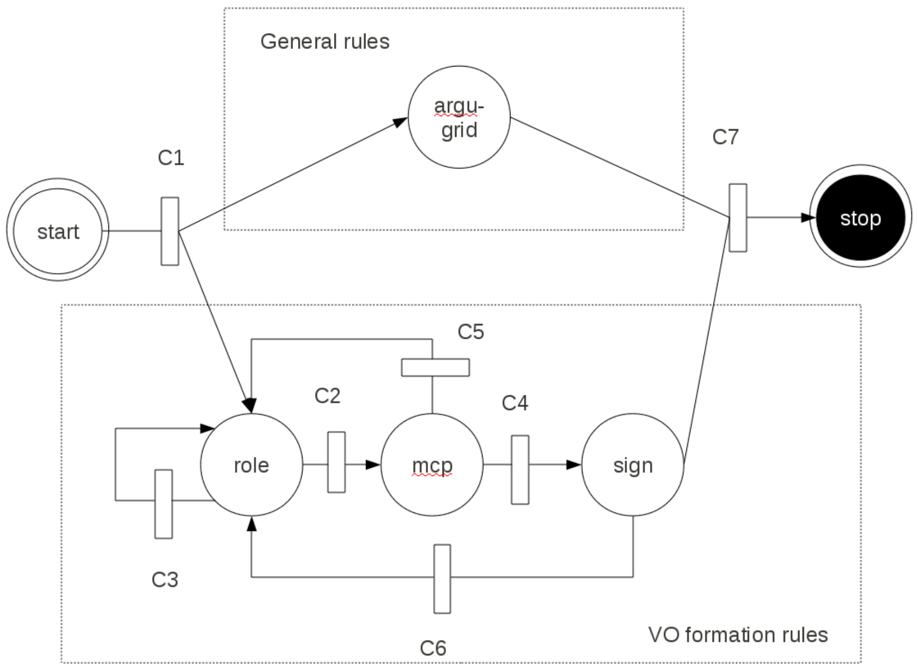


Figure 7.5: The vo_formation game in ARGUGRID.

to enable agents to interact within a more complex interaction mechanism viewed as a compound game. Agents situated in the physical environment can request the creation of such game and its sub-games using the meta-game mechanisms specified in Chapter 5 and implemented in Chapter 6.

7.4.3 Representing the VO Formation State

To give an example of how sub-games will appear in the main game of a practical application, we show next the state of the vo_formation game specified in ARGUGRID and illustrated in Fig. 7.5. The term:

vo_formation: Id [
 members ⇒ {agent:a1, agent:a2, agent:a3},

```
 sub_process ⇒ Workflow,
 cycle ⇒ started,
 result ⇒ nil
].
```

defines the state of the vo_formation game with two attributes, the first is the attribute members which lists the agents taking part in the vo_formation phase and the second attribute sub_process which defines the compound game which is specified in the Workflow value of the sub_process attribute, instantiated to terms of the form:

```
seq([and_split((start:s1, _), [argugrid:a1, P1, P2, P3, P4]),
     and_join([argugrid:a1, sign:s1],([object(s1, result,_)], stop))
  ].
```

The above instance of the Workflow variable states that the process of the vo_formation is a sequence (seq) of sub-games involving first an and_split in two parallel games, then the argurid game with identifier a1 and finally other games which are denoted as P1, P2, P3 and P4.

```
P1= seq(role:r1,if([object(r1,result,_)],
          seq(mcp:m1,if([object(m1,result,_)],sign)))
      P2= repeat([object(r1,exception,_)], role:r1)
      P3= repeat([object(m1,exception,_)], role:r1)
      P4= repeat([object(s1,exception,_)], role:r1)
```

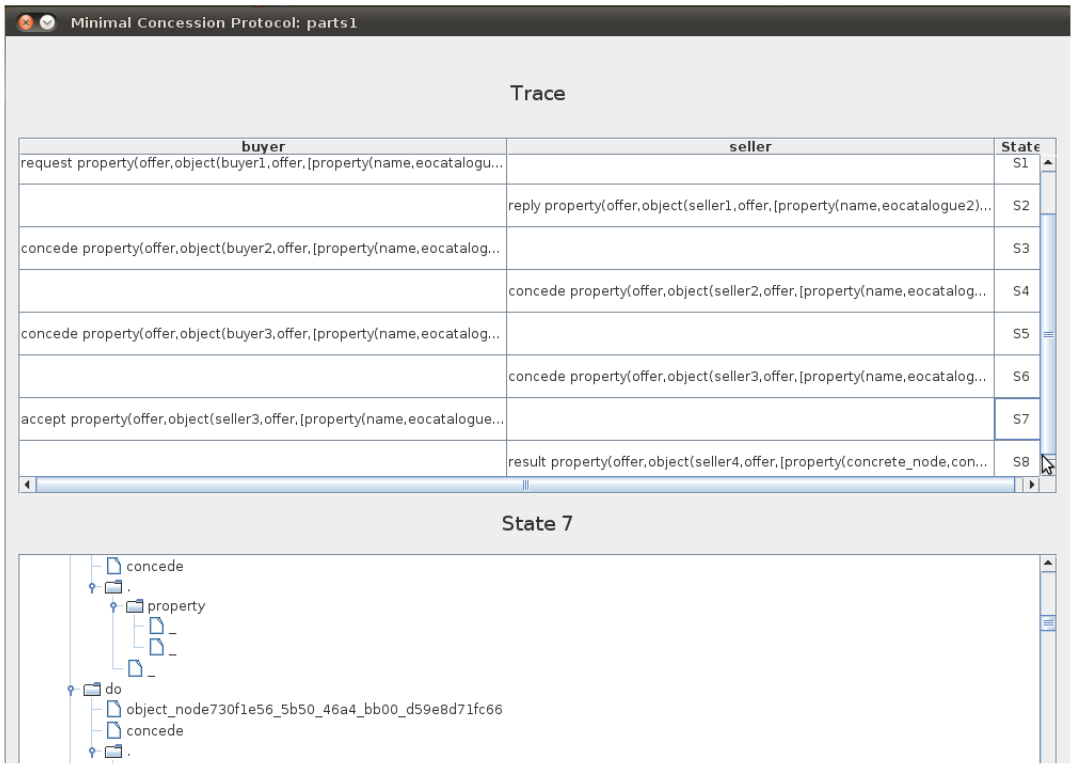
The pattern P1 defines a sequence of the role game with identifier r1. After this game is played, and if there is a result in the role game indicating the successful termination of the game, the roles of the agents in the VO have been agreed. The if pattern following the role game defines that if the game r1 was successful, another sub-game defined as a seq pattern can start. The seq pattern defines that the mcp game, identified as m1, can start and it is followed by another if pattern stating that the successful termination of mcp allows the sign game, identified as s1, to start. Therefore only if the agreement attribute of m1 is set to achieved, the sign game with identifier s1 is started and played to complete the negotiation process. The pattern P2 defines a repeat pattern where if the role game has an attribute exception, the role game is repeated again. Similarly, the pattern P3 defines a repeat pattern where if the mcp game has an attribute exception, the role game is repeated again. The pattern P4 defines the same repeat pattern for the case when the sign game has an attribute exception so that the role game is repeated again. The final and_join defines that the whole compound game stops when all the four atomic games are terminated.

7.5 Demonstration of EOS

In [24] we demonstrated how the Earth Observation Scenario, described earlier in Section 7.2, is handled by a set of negotiating MAGE/GOLEM agents. These tests included three types of runs.

The first run included a buyer and a seller agents that negotiate about the price of an eocatalogue service. The buyer agent has a user specified preference for finding this service with a low price and a high delivery time. Both agents adhere to the reciprocity principle, for this reason, if possible, they concede, otherwise they standstill. In this demo, both agents have internal private constraints on such service. If their constraints on the attribute price are compatible, they will eventually reach an agreement.

In the second run, the two agents were given incompatible constraints on the price of the service. When the agent would play the mcp game, they could not agree by simply negotiating the attribute price. In fact, if there is space for bargaining, the agents can find an agreement, if not they can decide to use a reward mechanism, where the agent concedes on a different attribute


Minimal Concession Protocol: parts1

Trace

buyer	seller	State
request property(offer,object(buyer1,offer,[property(name,eocatalogu...		S1
	reply property(offer,object(seller1,offer,[property(name,eocatalogue2)...	S2
concede property(offer,object(buyer2,offer,[property(name,eocatalog...		S3
	concede property(offer,object(seller2,offer,[property(name,eocatalog...	S4
concede property(offer,object(buyer3,offer,[property(name,eocatalog...		S5
	concede property(offer,object(seller3,offer,[property(name,eocatalog...	S6
accept property(offer,object(seller3,offer,[property(name,eocatalogue...		S7
	result property(offer,object(seller4,offer,[property(concrete_node,con...	S8

State 7

```
concede
.
  property
    _
    _
  _
do
  object_node730f1e56_5b50_46a4_bb00_d59e8d71fc66
  concede
  .
```

Figure 7.6: Negotiation Demo for the Earth Observation scenario.

of the eocatalogue service. When the game comes into a standstill point, the agent can apply the reward by conceding on another attribute. In our run, this was sufficient to allow the two agents to find an agreement.

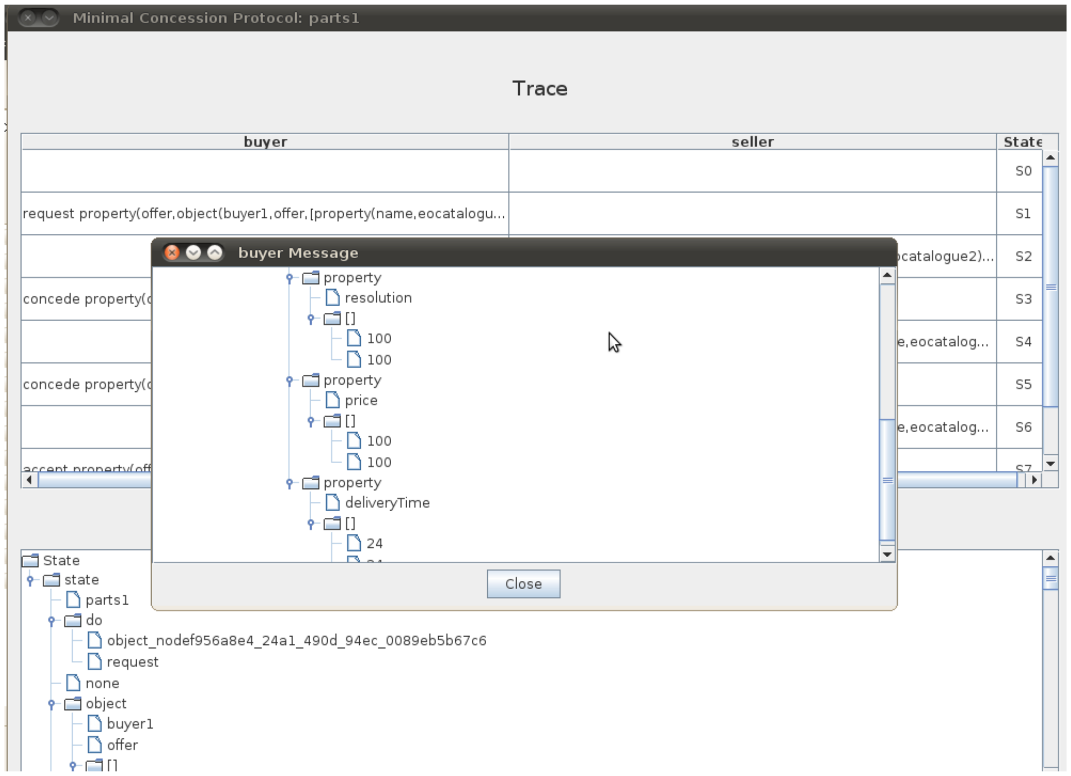
In the third run, the two agents could not find an agreement and the mcp would terminate without success. However, the buyer agent could continue interacting in the same game by selecting a new service provider and starting the negotiation from the role game.

The demonstration relied in a compound game as defined in the previous section, in order to allow the buyer agent to engage in a negotiation process for every service it requires, until the goal has been fulfilled and the agent can provide a concrete service to the user. This mechanism has the advantage that it is possible for the agents to choose the games with which to interact. Also, if as result of their interactions, there is a need to reiterate some of the games or to choose another game, this is still possible within the MAGE framework.

The atomic games allow the interactions to be mediated. This has the advantage that the agents are observing a shared state which is updated consistently, independently from where the agent is situated. In other words, every game the agents play give rise to a shared social environment between the agents. The moves the agent make update the state of the game which is observed consistently by all the player agents. This mechanism has two advantages: the first is that the agents do not have to keep a consistent state of the interaction in their knowledge base which will over complicate them, while the second is that the shared state is updated consistently by all the interacting members.

Fig. 7.6 shows a running demo of the minimal concession protocol used by an agent in a seller role and an agent in a buyer role to negotiate about a satellite service. The figure shows the two agents alternating the following messages: request, reply, concede, concede, concede, concede, accept and result. This sequence of messages can be different depending on the strategies of the agents, their bargaining power and on the properties requested or provided by them. The bargaining power and the peculiarities of the offered service is not known to the opponent agent. In this run, the protocol starts with a buyer agent requesting an eocatalogue service which provides images from a satellite. With the request, the buyer agent states also the required non functional properties of the service. By clicking over the request message the demo visualises the full content of the message, a snapshot of which is shown in Fig. 7.7. In this particular example the buyer agent has a bargaining power on the price between 100 and 140 and a high priority for the attributes of delivery time and price.

The buyer agent, starts the mcp game and requests the eocatalogue service


Minimal Concession Protocol: parts1

Trace

buyer	seller	State
		S0
request property(offer,object(buyer1,offer,[property(name,eocatalogu...		S1
	catalogue2)...	S2
concede property(		S3
	e,eocatalog...	S4
concede property(		S5
	e,eocatalog...	S6
accept property(off		S7

buyer Message

```
property
  resolution
  []
    100
    100
property
  price
  []
    100
    100
property
  deliveryTime
  []
    24
```

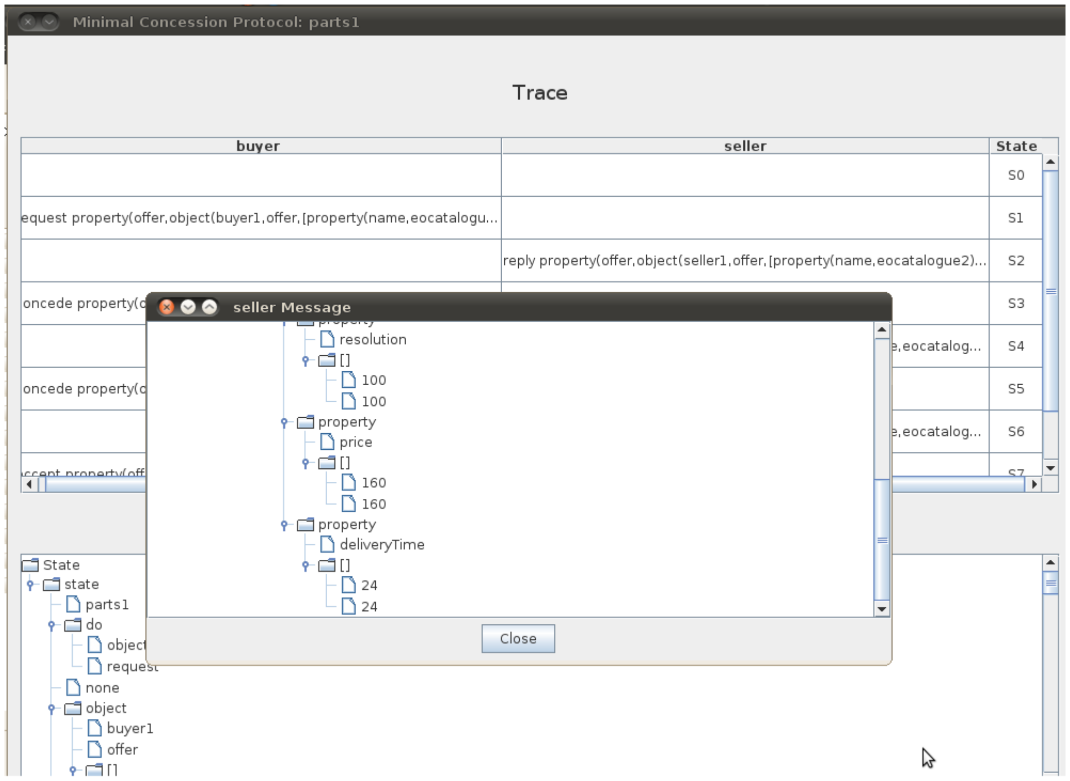
Close

```
State
  state
    parts1
    do
      object_nodef956a8e4_24a1_490d_94ec_0089eb5b67c6
      request
    none
    object
      buyer1
      offer
      []
```

Figure 7.7: Negotiation Demo for the Earth Observation scenario: The Buyer's Request Message

for the price of 100, resolution of 100 and delivery within 24 hours. On the other side, the service provider, who has taken a seller role, has a bargaining power on the price between 120 and 160, and after receiving the request of the buyer agent decides to reply by matching all the other attributes of the service except for the price. The seller offers the same service for the price of 160 as shown in Fig. 7.8.

At any time, the participants that observe the state observe also the valid moves within the game. For example, after the reply move made from the agent having a seller role, the agent in a buyer role can make the following valid moves: accept, reject, concede, standstill. In Fig. 7.6 we can see that the buyer, after considering its valid moves in that state of the game, decides


Minimal Concession Protocol: parts1

Trace

buyer	seller	State
		S0
equest property(offer,object(buyer1,offer,[property(name,eocatalogu...		S1
	reply property(offer,object(seller1,offer,[property(name,eocatalogue2)...	S2
oncede property(c		S3
	e,eocatalog...	S4
oncede property(c		S5
	e,eocatalog...	S6
ccept property(off		S7

seller Message

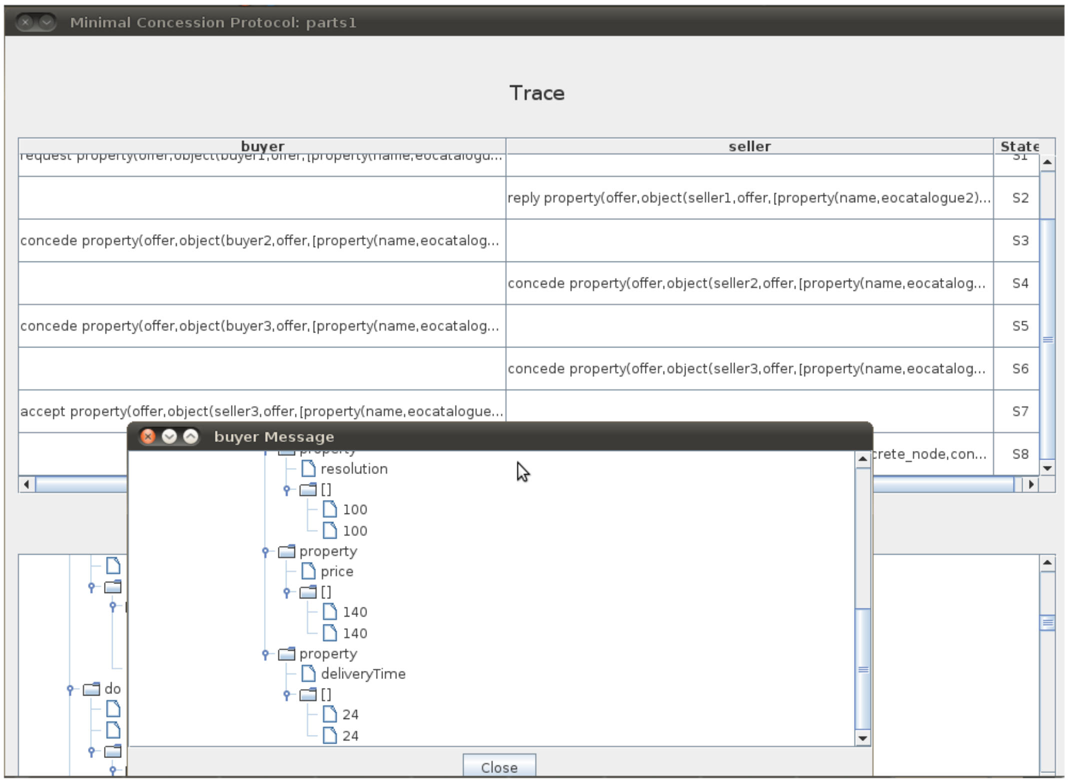
```
property
  resolution
  []
    100
    100
property
  price
  []
    160
    160
property
  deliveryTime
  []
    24
    24
```

```
State
  state
    parts1
    do
      object
      request
    none
    object
      buyer1
      offer
      []
```

Figure 7.8: Negotiation Demo for the Earth Observation scenario: The Seller's Reply Message

to concede. The other agent is unaware what is the reasoning process that makes the other agent decide to concede instead of selecting one of the other moves. What the interested agent can observe however is the move that was made and decide on its own move based on the new state of the interaction.

The number of exchanged messages within the mcp game is not limited. The negotiation can continue until either one of the agents accept the offer or one of the agents decides to reject the offer made by the other agent. If either buyer or seller accepts the last offer made the result of the game is the agreement (given by the last offer made in the game). In Fig. 7.9 we show that the seller and the buyer achieve an agreement when the price offered by the seller for the requested service is in the bargaining power of the buyer


Minimal Concession Protocol: parts1

Trace

buyer	seller	State
request property(offer,object(buyer1,offer,[property(name,eocatalogu...		S1
	reply property(offer,object(seller1,offer,[property(name,eocatalogue2)...	S2
concede property(offer,object(buyer2,offer,[property(name,eocatalog...		S3
	concede property(offer,object(seller2,offer,[property(name,eocatalog...	S4
concede property(offer,object(buyer3,offer,[property(name,eocatalog...		S5
	concede property(offer,object(seller3,offer,[property(name,eocatalog...	S6
accept property(offer,object(seller3,offer,[property(name,eocatalogue...		S7
	crete_node,con...	S8

buyer Message

```
property
  resolution
  []
    100
    100
property
  price
  []
    140
    140
property
  deliveryTime
  []
    24
    24
```

Figure 7.9: Negotiation Demo for the Earth Observation scenario: The Seller's Accept Message

agent. In the case one of the agents rejects the last offer, the mcp game has as a result a missed agreement. To guarantee the termination of the protocol we also define that three consecutive standstill moves means that there is no agreement on the current offer and the fourth consecutive standstill move counts as a reject move and, also, in this case the game has no agreement.

7.6 Summary

In this Chapter we presented the ARGUGRID project. We showed how MAGE supports agent interactions within the ARGUGRID platform. We explained the earth observation scenario and highlighted the steps that an agent has to do before being able to find a solution for a given goal.

Agents interact with other agent using a VO formation mechanism allowing them to negotiate an agreement, We defined a vo_formation compound game to coordinate and support these interactions. The main protocol used in the vo_formation game is the minimal concession protocol. The definition of the minimal concession protocol is such that if agents use it in combination with a minimal concession strategy, it guarantees termination and Nash equilibrium.

We illustrated how we specify the vo_formation compound game using the workflow coordination patterns. The coordination patterns allow agents to play atomic games in different orders. If an agreement cannot be achieved by playing a game, we allow agents to reiterate the same game or choose another game until they find an agreement.

Finally, we applied these ideas to support the implementation of the earth observation scenario. In this implementation we have demonstrated how agents use the game framework to agree on a service which is needed to satisfy the given users requirements. This approach illustrates how MAGE extends GOLEM with a useful way to support social interactions in a practical applications.

Chapter 8

Evaluation

This thesis so far has shown how to model MAGE as a mediation framework acting as a social environment that supports interactions between heterogeneous self-interested agents. We have separated the physical environment from the social environment so that they can evolve separately and with different mediation rules. The physical and the social environments have been connected using a propagation mechanism which makes the division transparent to the agents. We have also exemplified the MAGE approach by showing how to apply it to a practical application.

In this chapter we evaluate two aspects of the MAGE framework. Firstly we define a methodology for constructing complex interaction games between agents. We argue that the game metaphor presented in this thesis is very useful for developing social aspects of MAS applications. Secondly, we perform an experimental evaluation to understand how distribution of the environment improves the performance of the system. We focus on how the increasing number of events generated within the system reflects in the time taken within the containers to evaluate the act of the agent.

The reminder of the chapter is organised as follows: In Section 8.1 we describe how to build an agent application using atomic and compound games in a methodological way. In Section 8.2 we use the Open Packet World to measure how the time to evaluate an action with many social containers is influenced by the number of events in the system. At first we test OPW in a centralised setting where there are no social containers, then we see what happens when we introduce one social container and one physical container and finally we distribute the system and evaluate its performance. Finally, in Section 8.3 we summarise and conclude this chapter.

8.1 MAGE Methodology

In order to define MAGE-based applications we define a methodology which follows the waterfall model of software development [106]. The reason why we have chosen the waterfall model is because we are interested in using MAGE to prototype MAS applications of the kind developed in this thesis. The waterfall model designs a sequential process that follows the phases of *requirements, design, implementation, testing* and *maintenance.*

As shown in Fig. 8.1, in the *requirements* phase the developer analyses the requirements for the specific application that he is modeling. Afterwards, in the *design* phase, the developer identifies and specifies the tasks and behaviour of the agents, what are the possible atomic interaction needed in the system, the roles and the responsibilities of the interacting agents and possible coordination patterns between the various atomic interactions. In the *implementation* phase, the atomic interaction are implemented as atomic games and the coordination patterns will be implemented as compound games. Finally agents that are players with specific strategies are implemented. In the *verification* phase, we define the agent environment as a set of physical and social containers and deploy the agents. We then can specify tests to check the correct behaviour of the application. In the *maintenance* phase, improvements can be identified on how to refine the application.

The waterfall model shown in Fig. 8.1 shows all the steps required to de-

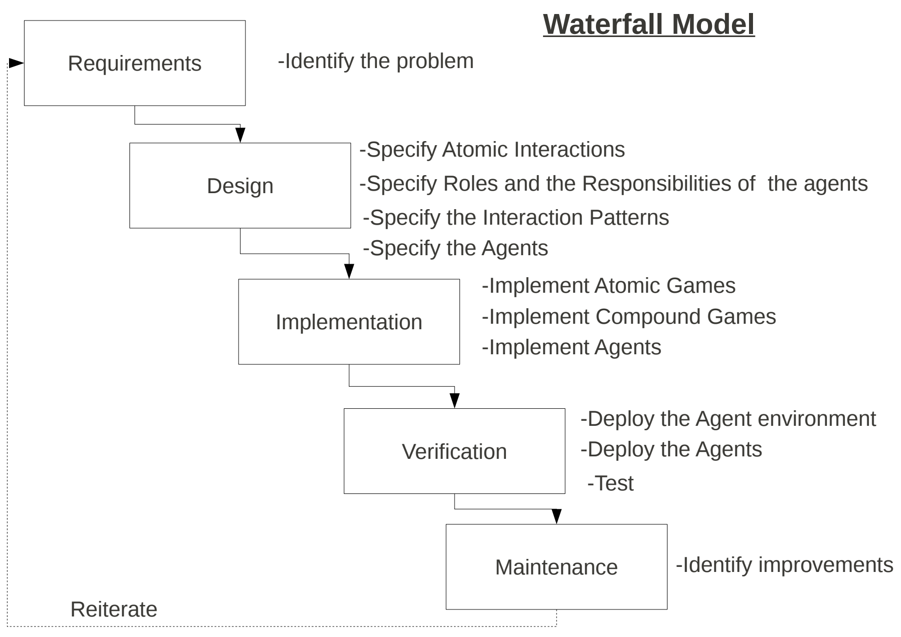


Figure 8.1: Waterfall Model in MAGE.

velop the entire functionalities that prototype a MAS application. Once the prototype is tested, more agents with possibly different designs can be deployed in the system. The MAGE methodology is mainly concerned with the specification of the agent interaction model, which, in MAGE terms. translates to the specification and implementation of the atomic and compound games. Once these are in place, the definition of the agents is based on how to interface the agents with the games which is done by querying the social containers as described in Chapter 6.

8.1.1 Specification of games

In MAGE, the social interactions within a MAS application are specified as games. As we describe throughout this thesis, MAGE consists of atomic and compound games. For every atomic game identified in the *design* phase the developer specifies the following:

1. The state of the game in terms of properties;
2. The initial and the final state of the game;
3. The moves of the game;
4. When these moves are legal;
5. The effects of the moves on the state of the games.

Compound games are games of games. They also have a state, a set of moves which includes all the moves of the sub-games, the meta-moves for starting, stopping, suspending or resuming an atomic game, coordination patterns to specify how atomic games are coordinated at run-time and a set effects describing how the moves change the state of the compound game. To specify a compound game the developer defines:

1. Follow 1-5 for specifying the game;
2. The coordination patterns as part of the state;
3. The meta-moves and their effects.

We have illustrated how atomic games are specified in Chapter 4 where we show how to specify the state of the OPW game, how to define the legal moves in terms of permission, prohibition and obliged, how to specify the effects of the moves and sanctions and rewards. In Chapter 7 we show again a specification of the MCP protocol [43] in terms of atomic game. The specification of the compound game is shown in Chapter 5 where we define the patterns for the OPW scenario and we explain the meta-moves and the forwarding mechanisms of MAGE.

8.1.2 Implementation and testing

Once we have specified the games we can move to the implementation of the application. To implement a new agent environment based on MAGE, the developer can proceed with the following procedure:

1. Deploy the physical environment by:
    - Implementing the specification of the physical rules;
    - Deploying the physical containers with the specification of the physical rules. In this way the acts that the agents perform are mediated by the physical rules.
    - Defining the distribution of the physical containers by specifying how the physical containers relate to one another.
2. Deploy the social environment by:
    - Implementing the specification of the atomic games by following the procedure described in the previous section and the implementation details as described in Sec. 6.5;
    - Implementing the specification of the desired compound games as described in the previous section. In Chapter 5, Sec. 5.5, we defined the runs/4 predicates. These are general predicates to be added to the compound games to be able to coordinate them. Additionally social_move/2, check/4 and apply/3 predicates as shown in Sec 6.4.3 are included to enable the coordination the atomic games;
    - Implementing the specification of the Root meta-game by utilising the meta-moves as specified in Sec. 5.4.
    - Deploy the social containers with the respective specification of the social rules.
    - Define how the deployed social containers relate to one another.

3. Connect physical and social environment by:

    - Adding to the specification of the physical rules defined in point 1 the propagate rules as defined in Chapter 6, Sec. 6.4.2. This connects the physical and the social containers by allowing messages to be propagated in the social environment.
    - Adding to the same specification the add/3 predicates as defined in Chapter 6, Sec. 6.4.2. These rules connect the physical and the social containers and allow for updates to be propagated on the properties of the physical state;
    - Defining how the physical containers relate to the social containers.

4. Implement the agents by:

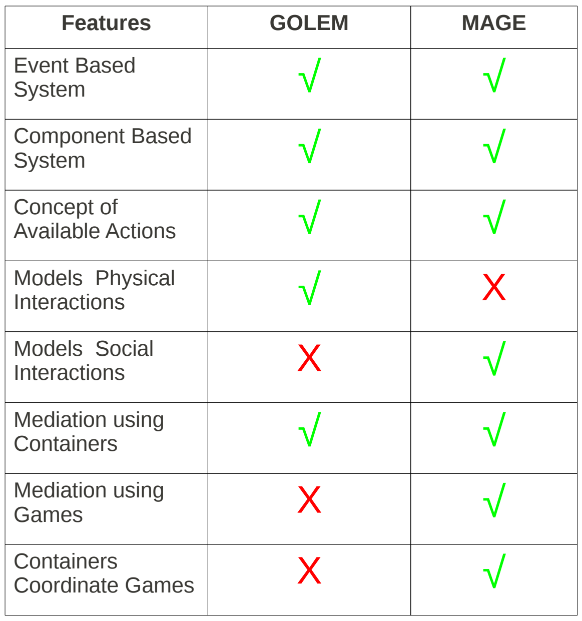
    - Implementing agents that can query the physical and social state;
    - Define their reasoning mechanisms and strategies;
    - Defining how they act in the environment.

Once the steps 1-4 are completed the tests consists on running the platform and observing and testing the behaviour of the agents on one hand and that the games and their state is updated in a correct way on the other. Within this thesis we haven not provided many tools for verification and testing as it was out of the scope of the thesis. Future work however will try to address these issues and provide a set of development tools to support such methodology.

8.1.3 MAGE-GOLEM: A Comparative Evaluation

In this thesis we have extended the GOLEM functionalities with MAGE. The main contribution of MAGE to the GOLEM model is that MAGE complements the concept of a physical agent environment with a model of the social agent environment and relates the two in an agent environment for specifying agent based applications.

In order to develop specific agent based applications we make use of both platforms to complete the steps 1-4 described in the previous section. The step 1 defines the physical agent environment within GOLEM, step 2 and 3 are the ones that define the social agent environment within MAGE and connect it to the physical one. These steps add more functionalities to the agent environment which are summarised in Fig. 8.2.



Features	GOLEM	MAGE
Event Based System	√	√
Component Based System	√	√
Concept of Available Actions	√	√
Models Physical Interactions	√	X
Models Social Interactions	X	√
Mediation using Containers	√	√
Mediation using Games	X	√
Containers Coordinate Games	X	√

Figure 8.2: A Comparison between MAGE and GOLEM features.

In Fig. 8.2 we show a comparative evaluation between MAGE and GOLEM. They are based in similar approaches thereby they are both event based systems. As we have shown throughout the thesis both are based on Object Event Calculus [74] and can reason about properties of the state of the agent environment that dynamically change in time. Additionally, they both are component based and provide the concept of available actions. MAGE provides the concepts of atomic and compound games, social containers and complex social containers, while GOLEM provides the concepts of physical containers, agents, objects and processes. All of these components come together when a developer defines a new application. The concept of available actions, has to do with the fact that actions within the environment change the state (physical and social) and agents can observe within containers (social and physical) what actions they can perform as the system evolves.

GOLEM and MAGE differ when it comes to defining how the system changes as result of agents moves and what rules apply to the actions of the agents. While in GOLEM we find the notion of physically possible and necessary actions, in MAGE we find the notions of permitted, forbidden, obliged and empowered moves. In this sense, we have that while GOLEM models the physical interactions that occur in the environment, MAGE models the social ones. The way this is done, is by mediating the actions that the agent perform using the social and physical containers. This means that both MAGE and GOLEM use containers to mediate the interactions. The difference however, is that MAGE has additional components within containers which are the games. Having games as additional components that mediate the interactions, implies that with an additional coordination component within containers, we can change the games dynamically. In turn this means that the rules of mediation within a social container change in time, while the rules of mediations in physical containers remain unchanged.

After the step 3 of the procedure defined in the previous section, the specification of the agent environment is now complete, therefore, following step 4 of the procedure, agents can be defined to act upon it. The additional benefits that we add with MAGE means that an agent can ask MAGE whether a move is permitted, forbidden, obliged or empowered, what are the properties of the current social state and what are the effects of the actions within the state. Due to the homogeneity between MAGE and GOLEM, the agent's interaction with MAGE is similar to the one with GOLEM. As a result, the agent is now able to check the implications of his actions, both, at a physical and at a social level.

8.2 Experimental Evaluation

One of the questions that are raised by the introduction of MAGE into the development of a MAS using GOLEM is how it affects the overall performance of the systems. For this purpose, we conducted a number of experiments using the OPW. This scenario was chosen as opposed to more realistic scenario, such as ARGUGRID, because it allows us to increase the size of the environment, the number of containers and agents deployed in it in a straight forward manner. In this way we can evaluate the possible configurations of how MAGE/GOLEM can be applied in practical applications.

The first two experiments run in a centralised agent environment. In the first experiment we do not use social containers. We define the OPW rules using the notion of possible/impossible actions defined in GOLEM. In the second experiment, we separate the physical and social state in respectively a single physical container and a single social container. In this experiment the physical and the social rules are evaluated in parallel. The OPW is centralised in the sense that all the agents are situated in one physical container. We then vary the number of agents that are deployed in the environment and measure the performances of the system. The third experiment consists in evaluating the social and physical rules in parallel in a distributed agent environment. Agents are distributed in more than one container and can move in the OPW grid by changing container. We vary the number of containers into two and four containers and, also, vary the number of agents deployed per container to check how the performances change.

In all the experiments we run, we were interested to measure how the time to compute the produced events relates to the total number of events produced in the system. In particular, we measured the time to compute if an action is physically possible and if an action is permitted by measuring the time taken to evaluate possible/2 and forbidden/2 rules against an action performed by an agent in the environment [1]. We then related this time to the total number of events produced by agents in the whole agent environment.

8.2.1 Test 1: Centralised OPW in GOLEM

This test consist in defining the social rules of the OPW using the notion of possible/impossible actions that is offered in GOLEM. For example, one of the OPW rules states that an agent is forbidden to go too close to another agent. We can define this in GOLEM only by specifying this action as impossible to perform.

For this test we defined one physical container deployed in an Intel Centrino Core 2 Duo 2.66GHz with 4GB of RAM. The OPW environment was represented by a 40x40 grid containing 100 packets. We deployed 10 agents to collect the packets and bring them into one of the 8 destinations in the grid. In all of the runs of the test, the agent minds were deployed in a separate machine and were remotely connected with their bodies deployed in the physical container in order to avoid having reasoning process of the agents stealing CPU resources from the agent environment.

Fig. 8.3 shows a linear curve representing the average time to compute a query in one GOLEM container. The time to query a container grows linearly with the number of events produced in the container. In this test, there is no distinction between the physical and the social rules. Given an action performed by an agent, this is evaluated if it is possible and, if it is, then the state of the container will change. As the number of events increases, so does the number of the records in the database that contains the information

[1] With exception to Test 1 where we measure only the time to compute possible/2 predicates.

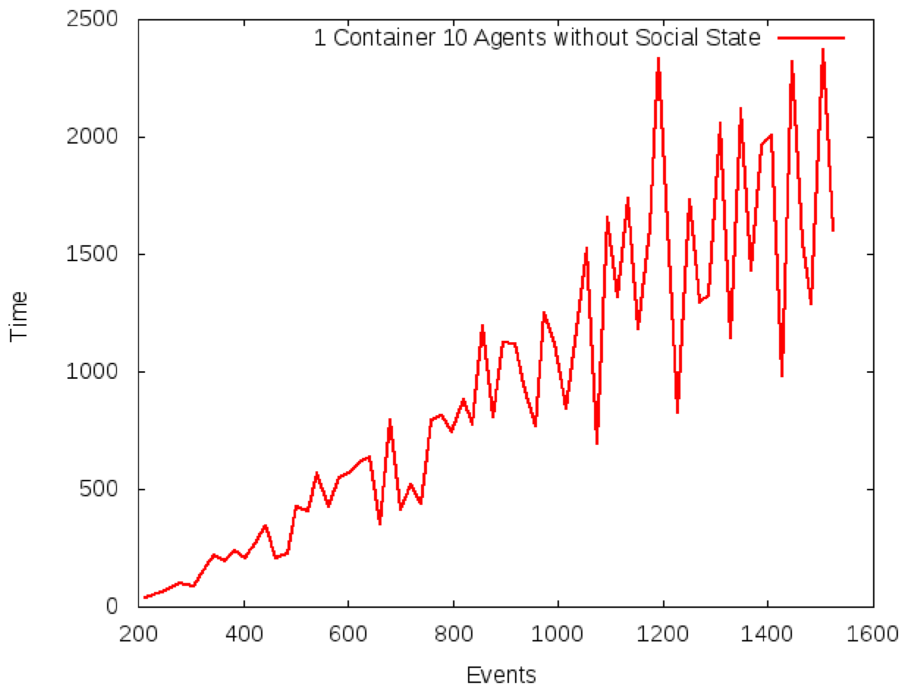


Figure 8.3: Time to query the physical state of a GOLEM container with 10 Agents.

about the state of the container. This means that, as agents perform more acts in the system, new information is stored in the database, thus, the evaluation of a query takes longer.

After 1500 events, the system becomes overloaded and the time to answer a query dramatically increases. In the next section, we distribute the social and physical state and show how this improves considerably the performances of the system.

8.2.2 Test 2: Centralised OPW in MAGE

In the second test, we define OPW using a centralised physical and a centralised social container (MAGE model). The agent environment is again deployed in an Intel Centrino Core 2 Duo 2.66GHz with 4GB of RAM. The OPW grid is the same 40x40 size and it contains 100 packets and 8 destinations. We first run the test with only 10 agents and compare it to the results shown in Fig. 8.3 where the social and physical state was not separated. In all of the runs of this test, the agent minds were deployed in a separate machine and were remotely connected with their bodies deployed in the physical container.

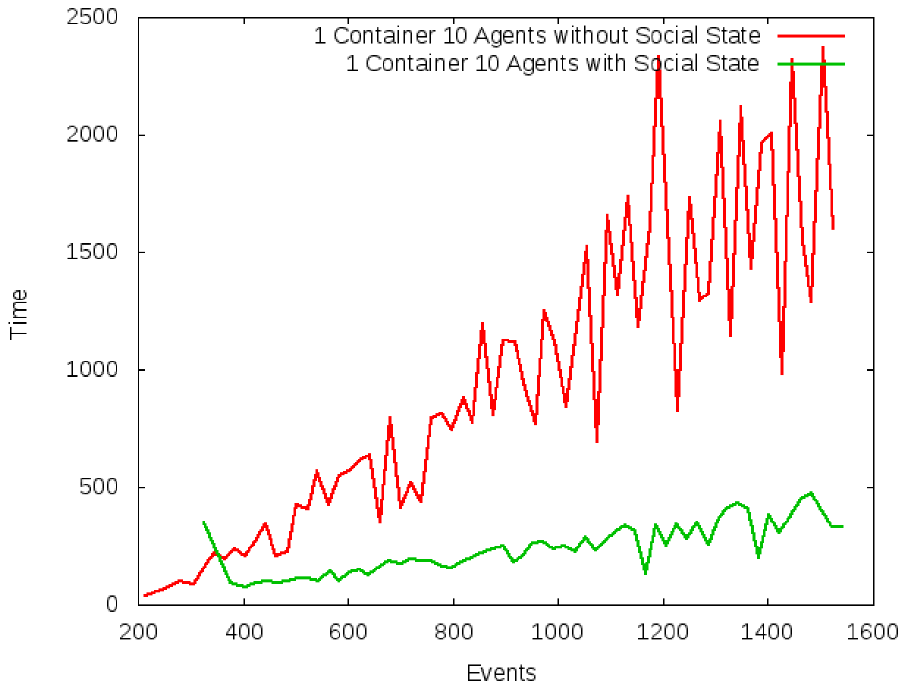


Figure 8.4: Time to query the physical state in GOLEM versus the time to query in parallel the physical and social state in MAGE.

Fig. 8.4 shows two linear curves respectively representing:

- the average time to compute queries performed by 10 agents in a centralised agent environment where there is no separation between the physical and social state and,

- the average time to compute the same queries, in the same settings, except that the physical and the social state run in parallel.

The figure shows that when we introduce MAGE and separate the social and the physical state, the slope of the curve decreases significantly, thus, such separation improves the performances of the system. We also noticed that the system could now run for more than 1500 events.

We run again the same test but increasing the number of agents from 10 to 30 and then 50 to see how the number of agents impacts on the performances of having a system defined using the MAGE model. Fig. 8.5 shows again three linear curves representing the average time to compute a query in an agent environment with respectively 10, 30 and 50 agents. Since the evaluation of the physical and social states is done concurrently, the curves represent the worst case between the computation of the social state and the computation of the physical state.

Also in this case, the three curves follow a linear behaviour suggesting that the time to query a container grows linearly with the number of events produced in the container. The fluctuations in the curves are explained as follows. The high peaks show the worst case where the attempted action was either impossible or invalid or both. As we check possible and valid actions in parallel and we wait for both threads to finish the execution, the time shown is the one that took longer between the two. Alternatively, the lowest peaks show the best case where the attempted action was either possible or valid or both. As before, the one shown is the one that took longer.

The figure shows that with an increasing number of agents acting in the system, the performances improve slightly. This is due to the fact that the same number of events is generated by more agents. These events generate properties in the two states that are often associated to the identification of the agents. When an agent queries these states, the values of the properties are accessed by agent identification. This means that, for a given number of events in the system, the properties to explore in order to answer a query are

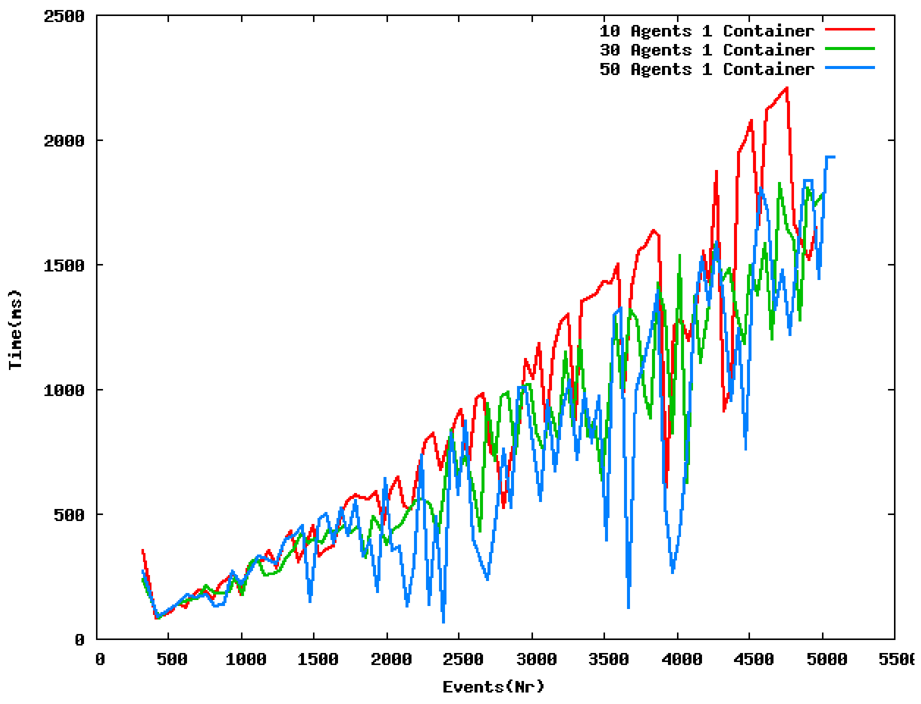


Figure 8.5: Time to query the physical and the social state of a GOLEM container with 10/30/50 Agents.

distributed amongst more agents which translates to shorter times to find the property and answer the query (we further explain this in the Results Section 8.2.4.

8.2.3 Test 3: Distributed OPW in MAGE

In the second series of experiments we distributed the OPW grid (40x40 squares) first into two physical containers (20x40 squares) and then into four (20x20 squares) physical containers. Each physical container was associated to a social container. For the distribution of the containers we used an Intel Centrino Core 2 Duo 2.66GHz with 4GB of RAM and an Intel Centrino Core Duo 1.66Ghz with 1GB of RAM. The agents were deployed between the distributed physical containers and could move from a machine to another by means of the mobility capabilities offered by GOLEM [22]. Fig. 8.6 shows what happens when we distribute the environment in multiple containers and use AEC to link these containers. As shown in Fig. 8.6, with a growing

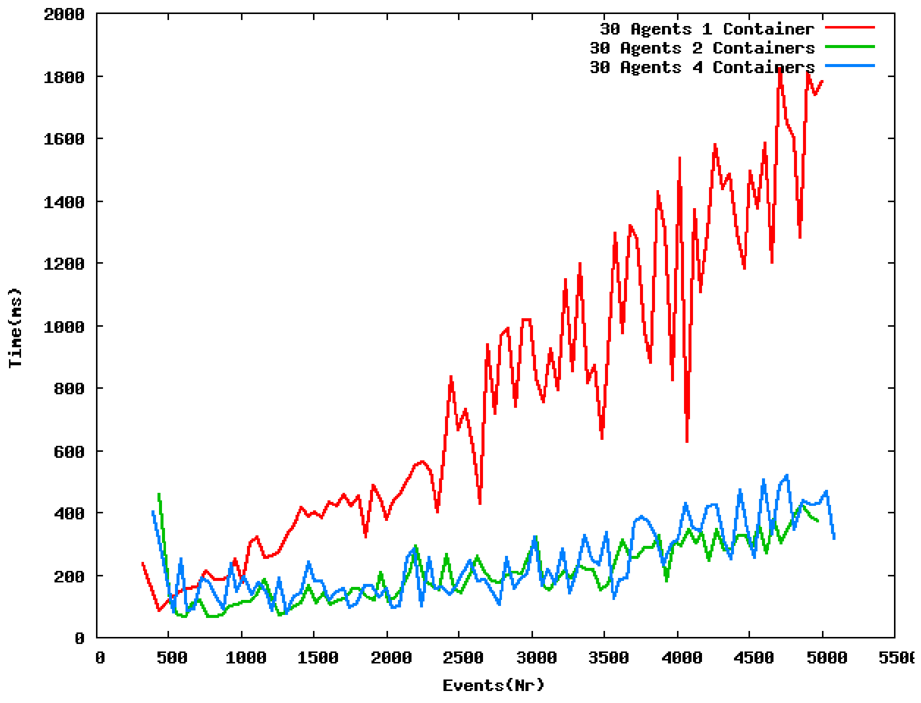


Figure 8.6: The effects of distribution.

number of events if we increase the number of containers, we improve considerably the performance. In Fig. 8.6 we show that in a system with a small number of events (0-500), it is better to compute the physical and social state using a centralised environment. With a bigger number of events, the experiment shows that we can achieve a big difference in performance if we distribute the environment in multiple physical and social containers.

In the distributed version the size of the grid managed by a single physical container becomes smaller and less complex terms (agents, packets and destinations) are registered in a single container. Between 500 to 3500 events, in average, having four or two physical containers and the respective social containers,does not make much difference. However, after 3500 events the performance of the application with two physical containers is better than the performance of the application deployed in four physical containers. This is due to the fact that with less packets on the grid (most of them after 3500 events have been delivered to the destinations), the agents moving on the grid are more likely to change containers in search for packets. The smaller the grid, the bigger the number of times agents try to move from a container to another. This introduces a distribution cost related to the cost of interactions between containers. For this reason, there is no improvement when we change from two to four containers.

8.2.4 Results

We can represent the time T_c to compute the social and physical state for the centralised agent environment with the following equation:

$$T_c = a * E + t0 \text{ with } a \sim Ne/Na$$

where *Ne* is the number of entities in the system, *Na* is the number of active entities performing events, E is the number of events in the system and *t0* is initial time to register the entities in the container. As the number of agents increases, then *Na* increases, which means that a decreases, which results in better performance. This is due to the fact that OEC is optimized to deal with events indexed by the identifiers of entities in the agent environment. For example, if we have 10 agents, 5000 events, and assuming that all agents perform the same number of events, each time that we call a `solve_at/6` predicate (e.g `solve_at(c1, ag1, picker, position, [3,4], 100)`), the search for the value of an agent attribute will evaluate a maximum of 500 entries (5000/10), while when we have 50 agents and the remaining conditions are the same, the search will evaluate a maximum of 100 entries (5000/50). Of course, if we consider an increasing number of agents, this also means that they produce more events in less time, but it also means that given the same number of changes applied to the environment, the environment responds better with an increasing number of agents. Thus, the environment as supported by GOLEM scales up better in situations when there are many agents rather than few.

In general, the time to compute the physical and social state distributed over many containers is defined by the equation:

$$T_d = \frac{T_c}{d} + i \times c \sim \frac{Ne}{d \times Na} \times E + \frac{t0}{d} + i \times c$$

where T_c is the time to compute the same experiment with a centralised test where we deploy only one social and one physical container, d is number of containers used in the decentralised version, i is the number of interactions between containers and c is the cost of container interaction. In other words, when we distribute the agent environment in multiple containers, the time to compute the physical and the social state is inversely proportional to the number of containers, thus improving the performance. However, there is an additional delay to compute the physical and social state which is due to the interactions between the containers.

The experiment shown in Fig. 8.6 suggests that there is a lower bound under which distributing the environment further does not improve the performances. However, once we establish the setting of the environment (grid size and number of containers), we can improve the performances by having more agents performing actions in the environment as in the first experiment (Fig. 8.5).

8.3 Summary

In this chapter we started by defining a methodology for developing MAGE based applications. We summarised the definition of atomic games, presented in Chapter 4, and the definition of compound games, presented in Chapter 5, in terms of the necessary steps to specify such games. We defined the procedure for implementing a whole agent based application where the developer defines and implements a physical and social environment so that agents can interact based on physical and social rules of the agent environment. We then provided a comparative evaluation of the two agent platform we utilise to implement the agent environment. We show how MAGE differs from GOLEM and to what extend they complement each other.

In this chapter we also provide an experimental evaluation of the agent environment where the social and the physical state evolve as parallel structures. We have used the OPW scenario to design three types of tests: the first evaluates what happens to the performances in the absence of social containers, the second is an evaluation of a centralised agent environment having the physical structure and the social structure evolving in parallel; the second type of tests look at the performances in a distributed setting. The results showed that the distribution of the agent environment improves considerably the performances when the number of events generated by the entities populating the environment is very big. We also found out that distribution involves more queries amongst containers which in turn might decrease the performances of the system. This means that there is a trade off on how much we can distribute the agent environment and improve the performances of the system due to the cost introduced by querying the distributed state of the environment in many containers. The results of this work were published in [125, 124].

Chapter 9

Conclusions

In this thesis we presented a game-based approach for modeling interactions in social agent environments. We used the game metaphor to define the main features of a social environment, acting as a container of social interactions. We used this idea to develop MAGE, a logic based framework for building complex agent interactions based on complex games built from simpler, possibly atomic, sub-games. The usefulness of the MAGE approach has been exemplified using two scenarios, one to illustrate the features of the framework and the other to illustrate the potential of the MAGE approach to develop practical applications.

MAGE addresses the social aspects of an agent environment, intended as an abstraction that enables agents to perceive the environment and its rules and to hide the complexity of the social interaction mechanism by using the notion of social containers. MAGE defines social containers as containers of games. Social Containers are given the mechanisms to make the state of games evolve by means of events. The games are given a representation of the normative aspects of the interaction. Using a set of coordination patterns the games can evolve in many different ways thereby providing the flexibility agents need during interaction. The agents can perceive the changes of the game state and interact accordingly within the game framework.

In order to apply the ideas and show the validity of the MAGE framework, we have used the platform in two applications, the OPW scenario and the EOS scenario from the ARGUGRID project. OPW provides many challenging aspects due to agent mobility and the dynamics of the application. We used OPW as a pedagogical application to exemplify the features of MAGE, from how to express normative concepts in a game to how to define compound games so that agents can organise themselves in a VO. We also used the OPW to test experimentally the performance of the system. With EOS we demonstrated the use of games to provide a flexible approach to the negotiation process in ARGUGRID.

The reminder of this chapter is organised as follows: In Section 9.1 we give an overview of the thesis by summarising what we achieved in the previous chapters. In Section 9.2 we identify some of the limitations of our approach. Finally in Section 9.3 we give directions of possible future work.

9.1 Summary

The MAGE platform was motivated by the idea of extending the GOLEM platform with support for social interactions in a systematic way. The main driving application was the ARGUGRID platform where we needed to support agents interactions aimed at the formation of virtual organisations. When designing the framework, we had the issue on how to define an atomic interaction, such as a protocol and then how to provide a mechanism to construct more complex interactions from the atomic ones. The distribution of the interactions in the agent environment was another of the issues that this thesis has tried to address with the concept of social container. Overall we have met the aims and objectives proposed in the introduction in this thesis in the following ways:

- We provided a model and an implementation of a game-based framework defining the concept of social agent environment. Within the framework we defined atomic games incorporating normative rules for agent interactions and compound games as a combination of atomic games and workflow coordination patterns.

- We introduced a clear distinction between the physical aspects of the agent environment and the social interactions in a MAS application. We did this by having two types of concurrent and interconnected evolving structures: the physical and the social container. The physical container that evaluates acts for their physical effects in the environment and the social container that evaluates acts using norms to determine their social effects.

- We provided a runtime framework that can execute normative rules about permissions, institutional empowerments, prohibitions, rewards and sanctions. Using games we were able to organise these rules for the agent thus easing the implementation of the agents which do not have to reason and construct a complete knowledge about the application state.

- We defined workflow coordination patterns for coordinating complex interactions as complex games. This allowed us to define agent interactions such as the VO formation process as a compound game.

- We defined a mediation mechanism that uses the OEC [77, 74] specification and its AEC extension [22] that deals with distribution of the social environment.

- We extended the GOLEM platform [21] by developing social containers as components that store the state of game interactions between agents. We developed an infrastructure for the propagation of moves and the coordination of such interactions.

- We evaluated the proposed framework in two scenarios: (i) the OPW scenario [124, 125] to explain the main functionalities of the framework and to evaluate the performance of the system when we introduce norms on top of the physical rules (ii) the EOS scenario [120] where we showed how we specify the negotiation process for the VO formation phase.

In this thesis we have presented two separate background chapters. In Chapter 2 we introduced the main concepts supporting interaction in social agent environments. We argued that the autonomy model of agency implies the possibility of having agents misbehaving in the system, motivating the need to constrain and manage agents interactions. We also introduced the main approaches and models of social interactions.

In Chapter 3 we introduced the GOLEM agent platform to highlight the architecture and the features of GOLEM on which we based our MAGE framework.

To address the problem that motivated our work, we investigated the use of the game metaphor to model interactions: interaction is viewed as a norm-governed activity thought of as an atomic game. In Chapter 4, we specified a model of atomic interactions through a set of valid moves for the participants. Valid moves incorporate the concepts of permission, prohibition, empowerment and obligation.

The agents' moves produce effects on the state of an atomic game and each game has a set of initiating and terminating conditions. We have used the OEC, and its extension AEC, as the logic programming language for the definition of the framework.

Using Event Calculus to represent the MAGE framework has the following advantages. First, it exhibits a declarative semantics whose advantages, compared to procedural semantics, have been well-documented. Second, the Event Calculus offers a formal representation of the agents' actions and their effects. This is in contrast to semantic web languages that offer limited temporal representation and reasoning. Third, the availability of the full power of logic programming, which is one of the main attractions of employing the Event Calculus as the temporal formalism, allows for the development of very expressive social and physical laws. Fourth, we do not have to know from the outset the domain of each variable. Fifth, the OEC and the AEC versions used here provide an efficient and scalable reasoning mechanism, offering the kind of runtime support that is required for norm-governed multi-agent systems.

In Chapter 5 we defined compound games as the mechanism for providing more complex interactions in MAGE. We identified a set of workflow patterns that we use to specify the compound games. Using the workflow patterns we can define control flow mechanisms between sub-games. The workflow description of a compound game was interpreted to determine the active sub-games in the system. We described seven main coordination patterns which are the main control patterns identified from a control-flow perspective to capture aspects related to control-flow dependencies between various games.

In Chapter 6 we showed the implementation of the main functionalities of the social environment. We first illustrated how we extend the GOLEM architecture with the games metaphor. We implemented both a propagation mechanism so that agents acts are propagated from the physical to the social agent environment and a coordination mechanism that can evaluate if such acts comply with a subset of the social rules defined in the social environment. We also showed how we implemented games and how agents can observe their state.

In Chapter 7 we evaluated the game-based approach by testing the functionalities of MAGE in the ARGUGRID earth observation scenario. We specified an atomic game for the negotiation process in ARGUGRID and illustrated how to combine this by specifying a workflow for a compound game. The compound game in the earth observation scenario represents the formation phase of the VO life-cycle. We acknowledged that the benefits MAGE brought to ARGUGRID were to allow agents to interact in a flexible manner to create a VO that satisfies the users' goal and the service provider's goal. Having a social environment mediating the interactions, allowed us to remove complexity from the agent reasoning. Agents would perceive consistently the state of the game and act accordingly, thus we could focus on defining sophisticated reasoning for the agents.

In Chapter 8 we defined the MAGE methodology which summarised the steps that a developer needs to take to specify and implement game-base interactions. We defined a comparative evaluation between MAGE and GOLEM. And, we also evaluated the performances of the framework using the Open Packet World scenario. With a series of tests we showed that, due to the distribution of the agent environment, we can support applications with a large number of events. Tests verified that with a large number of events, the system has better performances in terms of time to respond to a query performed in a distributed agent environment rather than in a centralised one.

9.2 Limitations of the Approach

Evaluating the MAGE framework we have identified some limitations to our approach that can be tackled as part of future work and can be seen as future directions to explore. We can summarise the limitation of MAGE as follows:

- In MAGE, the social rules themselves are not first class abstractions. The implication of this is that agents cannot observe the rules of the atomic games and cannot change them at runtime, they can only perceive how the state of the games changes due to these rules. Having the social rules as first class abstractions would allow to rewrite, substitute or remove these rules at run time whenever these rules become aimless or an obstacle for the right functioning of the system.

- In MAGE the class of a game cannot evolve in time despite the fact that the OEC, and as a consequence the AEC, supports schema evolution (see [74] for more details). Having games that can evolve their class in time would allow to model normative systems where agents decide to join and form a different institution with new rules as well as allowing to consider institutions that are born due to bigger institutions splitting.

- In MAGE we took a bird eye perspective of the interactions and we did not provide a specific agent cognitive model to play games. We partially tackled this problem in ARGUGRID where we utilised argumentative agents to play MAGE games. Agents were enabled to decide their next act based on the state of the game they were observing. However, we did not provide a general reasoning mechanism representing how agents that are aware of the game environment can plan/act accordingly.
- In MAGE we allow for agents joining games at runtime, but we did not investigate the problem of discovering games in the distributed agent environment. This is important as in a distributed agent environment modeled as open, agents may join and leave at runtime. When agents join the open agent environment it could be beneficial for them to be able to discover other agents as well as existing games at runtime. The current implementation of MAGE relies on the AEC for the discovery of games in the agent environment, but as already discussed, the AEC is not suitable to perform discovery in large networks.

As far as it concerns the definition of rules as a first class abstraction, this extension would require us to consider rules as C-logic complex structures and as attributes of a game, which would imply that we could modify the behaviour of the game as well as adding and removing normative rules at runtime.

For what it concerns the schema evolution of games, we could tackle this problem by further extending the AEC to deal with the schema evolution of games in distributed settings.

The definition of an agent model that can play games in MAGE can be tackled by: a) adapting existing agent models such as the Knowledge Goals

Plans model (see [73] for more details) to handle the evolution of games; b) defining a new agent model that plans according to the known game state.

Finally, the limitation concerning the discovery of games in distributed and open agent environment could be tackled by publishing MAGE games as resources in a P2P network, as it already happens for agents in ARGUGRID (see [19] for more details).

9.3 Future Work

The work developed in this thesis provides a broad basis for future research. We outline here some of the directions we aim to further investigate in further supporting the use of games as a model for agent's social interaction models.

9.3.1 A GUI for Programming MAGE Games

An extension of MAGE would be to define a GUI to assist developers with the creation of games and the definition of complex games. Atomic games have specific components that include the valid moves, the state, the initiate and terminate rules and the effects that moves have in the state. We can define a cycle which can verify that the game has in place all the components of an atomic game thus assisting the developer to define these games. To allow developers to define compound games we need the GUI application to be able to interpret visual representation of workflow patterns into the workflow coordination language we used to specify compound games.

9.3.2 Monitoring VOs

As explained in Chapter 7, the experiments that we carried out within the ARGUGRID scenario did not include the support for the monitoring of agreements. We also suggested that the monitoring of the agreement can be handled with the game mechanisms we presented in this thesis. Input from the formation phase, instead of feeding the SLA templates as we did in ARGU-GRID, could be given to a monitoring game that deals with the execution of the agreement. To do so, the initial state of the monitoring game has to contain the agreement so that the valid moves specifying the normative positions of the agents depend on the agreement negotiated and signed with the previous interactions. The rules of the monitoring game would be defined as contract templates [32] which once the initial state given in input from the agreement between agents, the game is able to monitor the interactions.

9.3.3 Exception Handling

In MAGE, the compound games deal with interactions by using workflow coordination primitives. The possibility to start a game or not is given by the conditions defined within the workflow primitives. We would like to investigate what happens if the compound coordination mechanism does not work properly because none of the conditions for the workflow primitives are satisfied (i.e. due to external exceptions such as agents disconnecting from the agent environment or agents suspending games thereby leaving the compound game without the possibility to progress with a new game). The control patterns we introduced are only a few of all the possible patterns designed in the literature (see [127]). In the current stage of the MAGE implementation, the only way to deal with undesirable events encountered during execution, is by giving control to agents in deciding if a game runs, is terminated, suspended or resumed. However, to fully address the problem we need to extend the workflow primitives with exception handling patterns [107] that will be defined to deal with deviations from normal execution of the compound game. These patterns would allow us to define a set of actions as primitives that allow to remove or add a game to the compound game pattern, force a game to restart out of the normal control flow, suspend all the games within a compound game or rollback a game to a particular moment in time. These primitives and possibly other similar ones would allow to recover part of the ongoing interactions in the system.

9.3.4 Model Checking in MAGE

Another possible future direction, implies applying model checking algorithms [102] to MAGE games. This would allow us to discover if MAGE games respect a set of properties such as demonstrating that a compound game is deadlock free, its states are reachable, the games composing it can run in parallel without inconsistency use of resources and that the compound games terminate correctly. Model checking approaches could be used to compare agent strategies. In particular this would allow the agents to understand what are the strategies that allow them to reach a desired state of the world in finite number of steps.

9.3.5 Applications

We would like to demonstrate the generality of the MAGE approach by applying to more applications. One example is the pervasive health-care [129] is quickly becoming a hot topic also for the MAS community, as it provides countless opportunities to apply agent technology to perform assisted living for chronic illnesses such as Alzheimer, diabetes, cardiovascular diseases and so on. So far, systems like the one proposed in [20] propose the diagnosis of conditions related to an illness by means of a single personal agent associated to a particular patient. As discussed in [34], the coordination of multiple heterogeneous expert agents to diagnose a condition can improve the current practice by combining the knowledge of multiple agents. In order to coordinate the differential diagnosis process, we may apply dynamic MAGE games created and dissolved according to the particular needs of the agents monitoring one or more patients.

Appendices

Appendix A

A Basic Introduction to C-logic

C-logic, first presented in [31], is a logical framework for the representation of complex objects. C-logic provides support for the basic features of object oriented programming such as object identities, single-valued and multi-valued attributes and object types.

The reason to choose C-logic as a formalism to describe objects both in GOLEM and in MAGE (see Chapter 2 for more details) resides in the fact that C-logic complex terms can be represented as a conjunction of atomic properties that can be then translated directly to first-order logic formulas.

In more detail, in C-logic terms of the following form:

```
person:john
person:bob
```

specifies that john and bob are two identifiers of the person type, where the class is considered an unary predicate. To specify the attributes of the instances, C-logic makes use of labels as follows:

```
person:frank[
            name ⇒ 'Frank',
            address ⇒ 'Paris',
            hobbies ⇒ { basket,piano },
            children ⇒ {person:samuel, person:simon }
]
```

The complex term above is of type person, it has an identifier frank and a set of labels name, address, hobbies, children. In C-logic each label is a binary predicate and represents an attribute or a property of a complex term. Labels may have single or multiple values. A notation of the form:

name ⇒ 'Frank'

means that the complex term has an attribute name with a single value 'Frank', while a notation of the form:

hobbies ⇒ { basket, piano }

means that the complex term has an attribute hobbies with two values associated that are basket and piano. The main advantage of C-logic is that these terms have a direct translation to first-order logic. A complex term as the following one

person:frank[name ⇒ 'Frank', address ⇒ 'Paris']

can be represented as

person:frank ∧ frank[name ⇒ 'Frank'] ∧ frank[address ⇒ 'Paris'] which can be translated to

```
person(frank) ∧ name(frank, 'Frank') ∧ address(frank, 'Paris').
```

Since C-logic allows to model complex terms as a conjunction of atomic formulas, one object attribute can be modified independently from the others, facilitating the updating of the state of the object in the agent environment. Similarly, the C-logic syntax is a convenient way to represent the events that take place in the agent environment. For instance, the following Open-Packet-World event (see Chapter 4 for a description of the Open-Packet-World) performed by an agent to move from one position to another of the environment:

```
event(e100).
act(e100, move).
actor(e100,a1).
position(e100, [3,4]),
happens(e100, 100).
```

can be specified by means of the following complex term

```
move:e100[actor⇒a1, position ⇒ [3,4]].
happens(e100,100).
```

As Kesim points out in [74], the fact that a complex term in C-logic has a direct translation to a conjunction of atomic formulas allows us to mix C-logic with first-order logic formulas. As a consequence, rather than writing forbidden/2 rules using a first-order logic syntax as shown below

```
forbidden(G@T, Action) ←
  move(Act),
  actor(Act, A),
  act_label(Act, drop),
  flag(Act, PosA),
  neighbouring_at(this, [], _, 1, Object, Class, position, PosB, T),
  adjacent(PosA, PosB),
  (Class=destination; Class=packet).
```

we can write such rules using the syntax of C-logic.

```
forbidden(G@T, Action) ←
  do:Action[actor ⇒ A, act⇒drop, flag ⇒ PosA],
  neighbouring_at(this, [], _, 1, Object, Class, position, PosB, T),
  adjacent(PosA, PosB),
  (Class=destination; Class=packet).
```

which makes C-logic a convenient formalism to specify the rules of the agent environment in a compact form. See Chapter 4 for the meaning of these rules.

Appendix B

Implementation Appendix

In this appendix we show some further details on how we implement the effects of moves that change the structure of the system. We first show how we deal with meta-moves after they are propagated to the Root container. We show how the state of the root container changes as effect of such moves. Then we show how interleaving moves that are managed in every social container by changing the state of the game. The implementation of these predicates is quite similar, we however, for completeness show these effects into detail

B.1 The Effects of Meta-Moves

The coordination mechanism provides the rules on how to manage changes that happens to the state of the social environment when a move is performed by an agent. Meta-Moves will change considerably the structure of the containers or the structure of the games being played by agents. We first consider the moves for creating, destroying, suspending and resuming social containers. We want to allow agents to create and destroy containers because we expect social containers to be employed mostly to contain complex game interactions (such as VOs).

Using the `effects/3` predicate, we can specify how meta-moves change the state of the social environment. Using the meta-moves agents can choose to create a new complex game. When the interactions within games became obsolete, either due to termination of the game or due to exceptions, the complex game may be suspended to be resumed later or even canceled. The `effects/3` for the creation of a new interaction game within a container is defined as follows:

```
effects(Root, E, T):-
    E=do(A, new_container, [attribute(game,GameName)]),
    create_container(GameName, GID, T),
    GID=game(GameName, CID),
    unique_id(Ev),
    add([instance(CID,container,[attribute(game, GID),
                  attribute(cycle,started)]], Ev, T).

create_container(GameName, GID, T):-
   url_specification(GameName, Url),
   generate_id(Container)
   GID=game(GameName, Container),
   java_object('container.Container',
              [Container,GameName,"Ontology.wsml",Url],
              Container).
```

The first predicate checks that the performed act is the `new_container` meta-move. The predicate `create_container/3` is called to start a new container containing the game specification requested by the agent. The `add/3` predicate is called to create an additional event. Such event triggers the addition of the new created container and the information about the game it contains, as a new object with some properties (such as the game contained in the new social container) in the database of the root container. Similar predicates are defined also for interleaving of atomic games.

The second predicate implements the actual creation of a new container. As we showed in Chapter 6, containers are defined using the Java programming language. They contain Prolog theories that define how the containers manage the interactions. When we need to create a new container, we specify a new instance of the class `Container`. We also specify the Prolog based theory that the container uses and the affordances of the container. The affordances defines how the containers are perceived in the agent environment. In our case, the affordances describe a social container. Given a game description identified as `GameName`, the `url_specification/2` calculates the url `Url` of the game specification theory. The `generate_id/1` generates a unique identification for the container. With this information, and the description of the affordancies of the container (which is the same for all the social containers) a new instance of the class `Container` is generated. The syntax shown above uses the default Prolog to Java integration mechanisms that are provided by tuProlog (we refer the interested reader to [45] for more details).

The destruction of a social container is defined in a similar way:

```
effects(Root, E, T):-
    E=do(A,stop_container,[attribute(game,GameId)]),
    destroy_container(GameId, CID, T),
    unique_id(Ev),
    add([do(CID,terminated,[attribute(cycle,destroyed)])],Ev,T).

destroy_container(GameId, CID, T):-
   GameId=game(GameName, CID),
   holds_ at(CID, game, GameId, T),
   Container ← destroy.
```

The first predicate states that the effects of a `stop_container` move are to call the `destroy_container/3` predicate to destroy the container. A new event `E` is then created to update the changes made in the system in the database of the root container. The `destroy_container/3` predicate implements the actual destruction of the container that contains the game `GameId` specified by the agent. The predicate first checks in the data-base that there is an attribute `GameId` for the same container `CID` as identified by the move. In other words, the root container makes sure that the container contains the specified game. Afterwards, a Java call evokes the destruction of the container.

The suspension and resumption of social containers requires similar implementation to the one adopted above:

```
effects(Root, E, T):-
    E=do(A,suspend_container,[attribute(game,GameId)]),
    suspend_container(GameId, CID, T),
    unique_id(Ev),
    add([do(CID,suspended,[attribute(cycle,suspended)])], Ev, T).

suspend_container(GameId, Container, T):-
   GameId=game(GameName, CID),
   holds_ at(CID, game, GameId, T),
   holds_at(CID, cycle, started, T),
   Property=[attribute(game,GameId)]
   propagate(do(CID, suspended, Property),T).

effects(Root, E, T):-
    E=do(A,resume_container,[attribute(game,GameId)]),
    resume_container(GameId, CID, T),
    unique_id(Ev),
    add([do(CID,resumed,[attribute(cycle,started)])],Ev,T).

resume_container(GameId, Container, T):-
   holds_ at(CID, game, GameId, T),
   holds_at(CID, cycle, suspended, T),
   Property=[attribute(game, GameId)]
   propagate(do(CID, resumed, Property),T).
```

Similarly to the creation and destruction of social containers, the coordination primitives for suspending and resuming a social container call the respective `suspend_container` or `resume_container` predicates and finally create an event which updates the new changes in the state of the root container. The `suspend_container` predicate suspends the container `CID` by propagating the `suspended` event to the suspended social container (identified as `CID`). In this case, the notified container will change the attribute `cycle` to `suspended`. Acts of agents do not create changes in the state of a suspended container. When the container is in a `started` state changes in the state of such container will resume. The `resume` method is defined in a similar way.

B.2 Game Coordination

Atomic games contained within a social container, need to be coordinated as well. Agents can directly act over the life-cycle of an atomic game by starting, terminating, suspending and resuming atomic games. However, this process has an additional coordination component provided as part of the complex game description. For example, before agents are in a position where they can start an atomic game, the game should be available to the players. The availability of the game is checked using the `runs/3` coordination predicates explained in Chapter 5.

To start a new atomic game we define the following predicates in the social containers:

```
effects(CID, E, T):-
```

```
    E=do(A,new_game,[Properties]
    member(attribute(game,GameId), Properties),
    GameId=game(GameName, CID),
    new_game(GameId, CID, T),
    append(P,attribute(cycle,started), NewP),
    unique_id(Ev),
    add([do(CID, new_game, [attribute(game, GameId)]),
         instance(GameId,atomic_game,[NewP]],Ev,T).

new_game(GameId, G, T):-
    GameId=game(GameName, CID),
    url_specification(GameName, Url),
    consult(Url).
```

The first predicate deals with the case when an agent performs a `new_game` act. The `new_game/3` is called to start the atomic game. The predicate adds the rules of the game in the social container. The `add/3` predicate changes the state of the container to be updated with the new game.

The termination of an atomic game is defined in terms of conditions that bring a game to an end. Therefore we allow agents only to suspend and resume atomic games but not to terminate them as they are running. The definition of `effects/3` for suspending and resuming an atomic game is defined in a similar way:

```
effects(CID, E, T):-
    E=do(A,suspend_game,[attribute(game,GameId)]),
    GameId=game(GameName, CID),
    holds_at(GameId, cycle, started, T),
    unique_id(Ev),
    add([do(GameId,suspended,[attribute(cycle,suspended)]],Ev,T).
effects(CID, E, T):-
    E=do(A,resume_game,[attribute(game,GameId)]),
    GameId=game(GameName, CID),
    holds_at(GameId, cycle, suspended, T),
    unique_id(Ev),
    add([do(GameId,resumed,[attribute(cycle,started)]],Ev,T).
```

Similarly to the creation of atomic games, the coordination primitives for suspending and resuming a game changes the attribute `cycle` of the game `GameId` respectively to `suspended` and to `stared` .

Bibliography

[1] *Engineering MAS Environment with Artifacts, Second International Workshop, E4MAS 2005, Utrecht, The Netherlands, July 25, 2005, Selected Revised and Invited Papers*, volume 3830 of *Lecture Notes in Computer Science*. Springer, 2006.

[2] The Foundation for Intelligent Physical Agents (FIPA). http://www.fipa.org/, 2008.

[3] ARGUmentantion as a foundation for the semantic GRID (ARGUGRID). http://www.argugrid.eu/, 2009.

[4] Workflow Management Coalition. http://www.wfmc.org/, 2009.

[5] GRIA secure dynamic service-oriented collaborations. http://www.gria.org/, 2010.

[6] M. Alberti, M. Gavanelli, E. Lamma, P. Mello, F. Chesani, and P. Torroni. Compliance Verification of Agent Interaction: a Logic based Software Tool. In *In Proc. Intl. Workshop on Declarative Agent Languages and Technologies (DALT), LNAI 2990*, pages 1–24. Deis technical report, LIA series no 71, 2004.

[7] M. Alberti, M. Gavanelli, E. Lamma, P. Mello, and P. Torroni. Specification and verification of agent interactions using social integrity con-

straints. In *Electronic Notes in Theoretical Computer Science*, page 2003. Elsevier, 2003.

[8] M. Alberti, M. Gavanelli, E. Lamma, P. Mello, and P. Torroni. Modeling interactions using social integrity constraints: a resource sharing case study. In *In Proc. Intl. Workshop on Declarative Agent Languages and Technologies (DALT), LNAI 2990*, pages 243–262. Springer-Verlag, 2004.

[9] A. Artikis. *Executable Specification of Open Norm-Governed Computational Systems.* PhD thesis, Imperial College London, 2003.

[10] A. Artikis, J. Pitt, and M. Sergot. Animated specifications of computational societies. In *AAMAS '02: Proceedings of the first international joint conference on Autonomous agents and multiagent systems*, pages 1053–1061, New York, NY, USA, 2002. ACM.

[11] A. Artikis and M. J. Sergot. Executable specification of open multi-agent systems. *Logic Journal of the IGPL*, 18(1):31–65, 2010.

[12] A. Artikis, M. J. Sergot, and J. Pitt. An executable specification of a formal argumentation protocol. *Artif. Intell.*, 171(10-15):776–804, 2007.

[13] A. Artikis, M. J. Sergot, and J. V. Pitt. Specifying Norm-Governed Computational Societies. *ACM Trans. Comput. Log.*, 10(1), 2009.

[14] F. T. B. FIPA ACL Message Structure Specification. Component, Foundation for Intelligent Physical Agents, 6-12 2002. `http://fipa.org/specs/fipa00061/`.

[15] C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving.* Cambridge University Press, 2003.

[16] M. Barbuceanu, T. Gray, Serge, and S. Mankovski. Coordinating with obligations. In *Proceedings of the 2nd International Conference on Autonomous Agents (Agents98*, pages 62–69. ACM Press, 1998.

[17] M. Benerecetti, M. Panti, L. Spalazzi, and S. Tacconi. Verification of payment protocols via multiagent model checking, 2002.

[18] G. Boella and L. W. N. van der Torre. Regulative and constitutive norms in normative multiagent systems. In D. Dubois, C. A. Welty, and M.-A. Williams, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Ninth International Conference*, pages 255–266. AAAI Press, 2004.

[19] S. Bromuri. *Generalised Onto-Logical Environments for Multi-agent Systems*. PhD thesis, Royal Holloway University of London, 2009.

[20] S. Bromuri, M. I. Schumacher, and K. Stathis. Towards distributed agent environments for pervasive healthcare. In *Proceedings of the Eighth German Conference on Multi Agents System Technologies (MATES '10)*, 2010.

[21] S. Bromuri and K. Stathis. Situating Cognitive Agents in GOLEM. In *Engineering Environment-Mediated Multi-Agent Systems, EEMMAS 2007*, volume 5049/2008 of *Lecture Notes in Computer Science*, pages 115–134. Springer, 2007.

[22] S. Bromuri and K. Stathis. Distributed Agent Environments in the Ambient Event Calculus. In *DEBS '09: Proceedings of the third international conference on Distributed event-based systems*, New York, NY, USA, 2009. ACM.

[23] S. Bromuri, V. Urovi, P. Conteras, and K. Stathis. A Virtual E-retailing Environment in GOLEM. In *Proceedings of Intelligent Environments,*

(IE08), Seattle, US, July 2008.

[24] S. Bromuri, V. Urovi, M. Morge, F. Toni, and K. Stathis. A multi-agent system for service discovery, selection and negotiation. In *Procedings of The Eighth International Conference In Autonomous Agents and Multi Agent Systems AAMAS09*, Budapest, Hungary, May 2009. Demonstration.

[25] S. Bromuri, V. Urovi, and K. Stathis. Game-based e-retailing in golem agent environments. *Pervasive and Mobile Computing*, 5(5):623–638, 2009.

[26] S. Bromuri, V. Urovi, and K. Stathis. iCampus: A Connected Campus in the Ambient Event Calculus. *International Journal of Ambient Computing and Intelligence*, 2(1):59–65, 2010.

[27] F. T. C. FIPA ACL Communicative Act Library Specification. Component, Foundation for Intelligent Physical Agents, 6-12 2002. `http://fipa.org/specs/fipa00037/`.

[28] F. T. C. FIPA Contract Net Interaction Protocol Specification. Component, Foundation for Intelligent Physical Agents, 12-03 2003. `http://fipa.org/specs/fipa00029/`.

[29] F. T. C. FIPA Interaction Protocol Library Specification. Component, Foundation for Intelligent Physical Agents, 2-10 2003. `http://fipa.org/specs/fipa00025/`.

[30] C. Castelfranchi. Commitments: From individual intentions to groups and organizations. In V. R. Lesser and L. Gasser, editors, *Proceedings of the First International Conference on Multiagent Systems, June 12-14, 1995, San Francisco, California, USA*, pages 41–48. The MIT Press, 1995.

[31] W. Chen and D. S. Warren. C-logic of Complex Objects. In *PODS '89: Proceedings of the eighth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 369–378, New York, NY, USA, 1989. ACM Press.

[32] S. C. Cheung, P. C. K. Hung, and D. K. W. Chiu. A meta-model for e-contract template variable dependencies facilitating e-negotiation. In *Proceedings of the 21st International Conference on Conceptual Modeling*, 2002.

[33] A. Chopra and M. Singh. Contextualizing commitment protocols. In *Proceedings of Conference on Autonous Agents and Multi-Agent Systems (AAMAS)*, pages 1345–1352. ACM, 2006.

[34] A. Ciampolini, P. Mello, and S. Storari. Distributed medical diagnosis with abductive logic agents. In *ECAI2002 workshop on Agents in Healthcare, Lyon*, 2002.

[35] O. Cliffe, M. D. Vos, and J. A. Padget. Answer set programming for representing and reasoning about virtual institutions. In K. Inoue, K. Satoh, and F. Toni, editors, *Computational Logic in Multi-Agent Systems, 7th International Workshop, CLIMA VII, Hakodate, Japan, May 8-9, 2006, Revised Selected and Invited Papers*, volume 4371 of *Lecture Notes in Computer Science*, pages 60–79. Springer, 2007.

[36] R. Conte, R. Falcone, and G. Sartor. Introduction: Agents and norms: How to fill the gap? *Artif. Intell. Law*, 7(1):1–15, 1999.

[37] M. Dastani, D. Grossi, J.-J. C. Meyer, and N. A. M. Tinnemeier. Normative multi-agent programs and their logics. In J.-J. C. Meyer and J. Broersen, editors, *Knowledge Representation for Agents and Multi-Agent Systems, First International Workshop*, volume 5605 of *Lecture Notes in Computer Science*, pages 16–31. Springer, 2009.

[38] J. M. David, D. Robertson, and C. Walton. Protocol synthesis with dialogue structure theory. In *Argumentation in Multi-Agent Systems: Second International Workshop, LNAI*, pages 1329–1330. Springer-Verlag, 2006.

[39] P. Davidson. Categories of Artificial Societies. In *Engineering Societies in the Agents World II*, volume 2203 of *Lecture Notes in Computer Science*, pages 1–9. Springer, 2001.

[40] E. W. Dijkstra. *On the role of scientific thought.* Springer-Verlag New York.

[41] P. M. Dung, R. A. Kowalski, and F. Toni. Dialectic proof procedures for assumption-based, admissible argumentation, 2005.

[42] P. M. Dung, P. Mancarella, and F. Toni. A dialectic procedure for sceptical, assumption-based argumentation.

[43] P. M. Dung, P. M. Thang, and F. Toni. Argument-based Decision Making and Negotiation in E-business: Contracting a Land Lease for a Computer Assembly Plant. In *9th International Workshop on Computational Logic in Multi-Agent Systems (CLIMA)*, Dresden, 2008.

[44] E. H. Durfee and V. R. Lesser. Negotiating task decomposition and allocation using partial global planning. *Morgan Kaufmann Series In Research Notes In Artificial Intelligence*, pages 229 – 243, 1990.

[45] D. Enrico, O. Andrea, and R. Alessandro. Multi-paradigm Java-Prolog integration in Tuprolog. *Science of Computer Programming*, (2):217–250, Aug.

[46] J. Eskilson and M. Carlsson. SICStus MT–A Multithreaded Execution Environment for SICStus Prolog. In C. Palamidessi, H. Glaser, and

K. Meinke, editors, *Programming Languages: Implementations, Logics, and Programs*, volume 1490 of *Lecture Notes in Computer Science*, pages 36–53. Springer-Verlag, 1998.

[47] M. Esteva, D. de la Cruz, and C. Sierra. Islander: an electronic institutions editor. In *AAMAS '02: Proceedings of the first international joint conference on Autonomous agents and multiagent systems*, pages 1045–1052, New York, NY, USA, 2002. ACM.

[48] M. Esteva, J. A. Rodríguez-Aguilar, C. Sierra, P. Garcia, and J. L. Arcos. On the formal specifications of electronic institutions. In *Agent Mediated Electronic Commerce, The European AgentLink Perspective*, pages 126–147, London, UK, 2001. Springer-Verlag.

[49] M. Esteva, B. Rosell, J. A. Rodriguez-Aguilar, and J. L. Arcos. Ameli: An agent-based middleware for electronic institutions. In *AAMAS '04: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 236–243, Washington, DC, USA, 2004. IEEE Computer Society.

[50] N. Faci, S. Modgil, N. Oren, F. Meneguzzi, S. Miles, and M. Luck. Towards a monitoring framework for agent-based contract systems. In *CIA '08: Proceedings of the 12th international workshop on Cooperative Information Agents XII*, pages 292–305, Berlin, Heidelberg, 2008. Springer-Verlag.

[51] J. Ferber. *Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence*. Addison-Wesley, 1999.

[52] J. Ferber and O. Gutknecht. A meta-model for the analysis and design of organizations in multi-agent systems. In *ICMAS '98: Proceedings of the 3rd International Conference on Multi Agent Systems*, page 128, Washington, DC, USA, 1998. IEEE Computer Society.

[53] N. Fornara and M. Colombetti. Agent communication and artificial institutions. *Journal of Autonomous Agents and Multi-Agent Systems*, 2007:14–121, 2006.

[54] N. Fornara and M. Colombetti. *Multi-Agent Systems: Semantics and Dynamics of Organizational Models*, chapter Formal specification of artificial institutions using the event calculus. IGI Global, 2009.

[55] N. Fornara, F. Viganò, M. Verdicchio, and M. Colombetti. Artificial institutions: a model of institutional reality for open multiagent systems. *Artif. Intell. Law*, 16(1):89–105, 2008.

[56] I. Foster and C. Kesselman, editors. *The grid: blueprint for a new computing infrastructure*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[57] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *Int. J. High Perform. Comput. Appl.*, 15:200–222, August 2001.

[58] M. Fox, M. Barbuceanu, M. Grüninger, and J. Lin. An organizational ontology for enterprise modeling. In M. Prietula, K. Carley, and L. Gasser, editors, *Simulating Organizations: Computational Models for Institutions and Groups*, pages 131–152. AAAI Press/The MIT Press, 1998.

[59] D. Gaertner, A. Garcia-Camino, P. Noriega, J. A. Rodriguez-Aguilar, and W. Vasconcelos. Distributed norm management in regulated multiagent systems. 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.

[60] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns:*

elements of reusable object-oriented software. Addison-Wesley Professional, 1995.

[61] A. García-Camino, J. A. Rodríguez-Aguilar, and W. Vasconcelos. A distributed architecture for norm management in multi-agent systems. In *COIN'07: Proceedings of the 2007 international conference on Coordination, organizations, institutions, and norms in agent systems III*, pages 275–286, Berlin, Heidelberg, 2008. Springer-Verlag.

[62] M. Ghanem, N. Azam, M. J. Boniface, and J. Ferris. Grid-enabled workflows for industrial product design. In *Second International Conference on e-Science and Grid Technologies (e-Science)*, page 96. IEEE Computer Society, 2006.

[63] J. J. Gibson. *The Ecological Approach to Visual Perception*. Lawrence Erlbaum Associates, 1979.

[64] E. Giunchiglia and V. Lifschitz. An action language based on causal explanation: Preliminary report. In *In Proc. AAAI-98*, pages 623–630. AAAI Press, 1998.

[65] G. Governatori and A. Rotolo. Changing legal systems: legal abrogations and annulments in defeasible logic. *Logic Journal of the IGPL*, 18(1):157–194, 2010.

[66] F. Guerin and J. Pitt. Agent communication frameworks and verificatione. pages 98–112, 2003.

[67] J. F. Hübner, J. S. Sichman, and O. Boissier. Developing organised multi-agent systems using the moise+ model: Programming issues at the system and agent levels. *International Journal of Agent-Oriented Software Engineering*, 2007.

[68] J. F. Hübner, L. Vercouter, and O. Boissier. Instrumenting multi-agent organisations with artifacts to support reputation processes. pages 96–110, 2009.

[69] JADE. Java Agent DEvelopment framework. Home Page: http://jade.tilab.com.

[70] N. R. Jennings, K. Sycara, and M. Wooldridge. A roadmap of agent research and development. *Autonomous Agents and Multi-Agent Systems*, 1(1):7–38, 1998.

[71] A. M. Jeremy, J. Pitt, and K. Stathis. Connected communities from the standpoint of multi-agent systems. *New Generation Computing*, 17:381–393, 1999.

[72] A. Jones and M. Sergot. A formal characterisation of institutionalised power, 1996.

[73] A. C. Kakas, P. Mancarella, F. Sadri, K. Stathis, and F. Toni. The KGP model of Agency. In *Proceedings of the 16th European Conference of Artificial Intelligence*, pages 33–37, Valencia, 2004.

[74] F. N. Kesim and M. Sergot. A Logic Programming Framework for Modeling Temporal Objects. *IEEE Transactions on Knowledge and Data Engineering*, 8(5):724–741, 1996.

[75] N. Kesim. *Temporal Objects in Deductive Databases.* PhD thesis, Imperial College, 1993.

[76] N. Kesim and M. Sergot. Implementing an object-oriented deductive database using temporal reasoning. *J. Database Manage.*, 7(4):21–34, 1996.

[77] R. Kowalski and M. Sergot. A logic-based calculus of events. *New Gen.*

Comput., 4(1):67–95, 1986.

[78] L. Lymberopoulos, S. Papavassiliou, and V. Maglaris. A novel load balancing mechanism for P2P networking. In *Proc. of ACM sponsored Conference GridNets*, Lyon, France, 2007.

[79] H. Mazouzi, A. E. F. Seghrouchni, and S. Haddad. Open protocol design for complex interactions in multi-agent systems. In *AAMAS '02: Proceedings of the first international joint conference on Autonomous agents and multiagent systems*, pages 517–526, New York, NY, USA, 2002. ACM.

[80] P. McBurney and S. Parsons. Games that agents play: A formal framework for dialogues between autonomous agents. *Journal of Logic, Language and Information*, 11(3):315–334, 2002.

[81] J. McGinnis, S. Bromuri, V. Urovi, and K. Stathis. Automated workflows using dialectical argumentation. In *GES07*. German e-Science Conference, 2007.

[82] J. Mcginnis and D. Robertson. Dynamic and distributed interaction protocols. In *In Proceedings of the AISB 2004 Convention*, pages 45–54, 2004.

[83] J. McGinnis, K. Stathis, and F. Toni. A formal framework of virtual organisations as agent societies. *CoRR*, abs/1001.4405, 2010.

[84] C. Menard. Markets as institutions versus organizations as markets? disentangling some fundamental concepts. *Journal of Economic Behavior & Organization*, 28(2):161–182, October 1995.

[85] N. Minsky and V. Ungureanu. Law-governed interaction: a coordination and control mechanism for heterogeneous distributed sys-

tems. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 9(3):273–305, 2000.

[86] S. Modgil, N. Faci, F. R. Meneguzzi, N. Oren, S. Miles, and M. Luck. A framework for monitoring agent-based normative systems. In C. Sierra, C. Castelfranchi, K. S. Decker, and J. S. Sichman, editors, *8th International Joint Conference on Autonomous Agents and Multiagent Systems, AAMAS (1)*, pages 153–160. IFAAMAS, 2009.

[87] M. Morge and P. Mancarella. The hedgehog and the fox. an argumentation-based decision support system. In *Proc. of the Quatrième Journées Francophones Modèles Formels de l'Interaction*, pages 357–364, 2007.

[88] Y. Moses and M. Tennenholtz. Artificial social systems. *Computers And Artificial Intelligence*, 14:533–562, 1995.

[89] W. X. Moustafa Gahem, Li Guo. Deliverable D5.3, Design of service broker and interface to agent technology argugrid project. Technical report, 2008.

[90] R. B. Myerson. *Game Theory: Analysis of Conflict*. Harvard University Press, September 1997.

[91] J. Odell, H. V. D. Parunak, M. Fleischer, and S. Brueckner. Modeling agents and their environment. In F. Giunchiglia, J. Odell, and G. Weiß, editors, *Agent-Oriented Software Engineering III, Third International Workshop, AOSE 2002, Bologna, Italy, July 15, 2002, Revised Papers and Invited Contributions*, volume 2585 of *Lecture Notes in Computer Science*, pages 16–31. Springer, 2003.

[92] A. Omicini and E. Denti. From Tuple Spaces to Tuple Centres. *Science of Computer Programming*, 41(3):277–294, nov 2001.

[93] Oracle. Berkeley Database Java Edition (visited in April 2009). `http://www.oracle.com/technology/products/berkeley-db/index.html`.

[94] A. Paschke and M. Bichler. SLA Representation, Management and Enforcement. In *EEE '05: Proceedings of the 2005 IEEE International Conference on e-Technology, e-Commerce and e-Service (EEE'05) on e-Technology, e-Commerce and e-Service*, pages 158–163, Washington, DC, USA, 2005. IEEE Computer Society.

[95] J. Patel, W. T. L. Teacy, N. R. Jennings, M. Luck, S. Chalmers, N. Oren, T. J. Norman, A. Preece, P. M. D. Gray, G. Shercliff, P. J. Stockreisser, J. Shao, W. A. Gray, N. J. Fiddian, and S. Thompson. Agent-based virtual organisations for the grid. *Multiagent Grid Syst.*, 1(4):237–249, 2005.

[96] J. Pitt. The open agent society as a platform for the user-friendly information society. *AI Soc.*, 19(2):123–158, 2005.

[97] J. Pitt, M. Anderton, and J. Cunningham. Normalized interactions between autonomous agents: A case study in inter-organizational project management. In *Interorganizational Project Management, International Workshop on the Design of Cooperative Systems (COOP'95)*, pages 76–95, 1994.

[98] J. Pitt, L. Kamara, M. Sergot, and A. Artikis. Formalization of a voting protocol for virtual organizations. In *AAMAS '05: Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*, pages 373–380, New York, NY, USA, 2005. ACM.

[99] J. Pitt, A. Mamdani, and P. Charlton. The open agent society and its enemies: a position statement and research programme. *Telematics and Informatics*, 18(1):67–87, 2001.

[100] J. Pitt and E. H. Mamdani. A protocol-based semantics for an agent communication language. In *IJCAI*, pages 486–491, 1999.

[101] J. Pitt, D. Ramirez-Cano, L. Kamara, and B. Neville. Alternative dispute resolution in virtual organizations. In A. Artikis, G. M. P. O'Hare, K. Stathis, and G. A. Vouros, editors, *Engineering Societies in the Agents World VIII, 8th International Workshop, ESAW 2007, Athens, Greece, October 22-24, 2007, Revised Selected Papers*, volume 4995 of *Lecture Notes in Computer Science*, pages 72–89. Springer, 2007.

[102] F. Raimondi and A. Lomuscio. Automatic verification of multi-agent systems by model checking via ordered binary decision diagrams. *J. Applied Logic*, 5(2):235–251, 2007.

[103] A. S. Rao and M. P. Georgeff. Modeling Rational Agents within a BDI-Architecture. In *Proceedings of the 2nd Intl. Conference on Principles of Knowledge Representation and Reasoning (KR'91)*, pages 473–484, 1991.

[104] A. Ricci, M. Viroli, and A. Omicini. Programming MAS with artifacts. In R. P. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors, *Programming Multi-Agent Systems*, volume 3862 of *LNAI*, pages 206–221. Springer, Mar. 2006. 3rd International Workshop (PROMAS 2005), AAMAS 2005, Utrecht, The Netherlands, 26 July 2005. Revised and Invited Papers.

[105] A. Ricci, M. Viroli, and A. Omicini. CArtAgO: A framework for prototyping artifact-based environments in MAS. In D. Weyns, H. V. D. Parunak, and F. Michel, editors, *Environments for MultiAgent Systems III*, volume 4389 of *LNAI*, pages 67–86. Springer, May 2007. 3rd International Workshop (E4MAS 2006), Hakodate, Japan, 8 May 2006. Selected Revised and Invited Papers.

[106] W. W. Royce. Managing the development of large software systems: concepts and techniques. In *Proceedings of the 9th international conference on Software Engineering*, ICSE '87, pages 328–338, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.

[107] N. Russell, W. M. P. van der Aalst, and A. H. M. ter Hofstede. Workflow exception patterns. In E. Dubois and K. Pohl, editors, *Advanced Information Systems Engineering, 18th International Conference, CAiSE*, volume 4001 of *Lecture Notes in Computer Science*, pages 288–302. Springer, 2006.

[108] J. R. Searle. *The Construction of Social Reality.* The Free Press, 1997.

[109] C. Serban and N. Minsky. In vivo evolution of policies that govern a distributed system. In *POLICY '09: Proceedings of the 2009 IEEE International Symposium on Policies for Distributed Systems and Networks*, pages 134–141, Washington, DC, USA, 2009. IEEE Computer Society.

[110] M. Shanahan. The event calculus explained. In *Artificial Intelligence Today*, pages 409–430. 1999.

[111] M. P. Singh and M. N. Huhns. *Service Oriented Computing.* John Willey and Sons, England, 2005.

[112] K. Stathis. *Game-Based Development of Interactive Systems.* PhD thesis, Imperial College, 1996.

[113] K. Stathis. A Game-based Architecture for Developing Interactive Components in Computational Logic. *Journal of Functional and Logic Programming*, 2000(5), 2000.

[114] K. Stathis, G. Lekeas, and C. Kloukinas. Competence Checking for the

Global E-Service Society Using Games. In *Engineering Societies in the Agents World VII*, volume 4457/2007 of *Lecture Notes in Computer Science*, pages 384–400. Springer, 2006.

[115] K. Stathis and M. J. Sergot. Games as a Metaphor for Interactive Systems. In *In HCI'96, People and Computers XI.*, pages 19–33. Springer-Verlag, 1996.

[116] T. J. Strader, F.-R. Lin, and M. J. Shaw. Information infrastructure for electronic virtual organization management. *Decis. Support Syst.*, 23:75–94, May 1998.

[117] S. Sunkle, M. Rosenmller, N. Siegmund, S. S. U. Rahman, G. Saake, and S. Apel. Features as first-class entities toward a better representation of features, 2008.

[118] N. A. M. Tinnemeier, M. Dastani, and J.-J. C. Meyer. Programming norm change. In W. van der Hoek, G. A. Kaminka, Y. Lespérance, M. Luck, and S. Sen, editors, *9th International Conference on Autonomous Agents and Multiagent Systems*, pages 957–964. IFAAMAS, 2010.

[119] F. Toni. E-business in ArguGRID. In J. Altmann and D. Veit, editors, *Grid Economics and Business Models, 4th International Workshop, GECON 2007*, volume 4685 of *Lecture Notes in Computer Science*, pages 164–169. Springer, 2007.

[120] F. Toni, M. Grammatikou, S. Kafetzoglou, L. Lymberopoulos, S. Papavassiliou, D. Gaertner, M. Morge, S. Bromuri, J. McGinnis, K. Stathis, V. Curcin, M. Ghanem, and L. Guo. The ARGUGRID Platform: An Overview. In *GECON*, pages 217–225, 2008.

[121] G. Tonti, J. Bradshaw, R. Jeffers, R. Montanari, N. Suri, and A. Uszok.

Semantic web languages for policy representation and reasoning: A comparison of KAoS, rei and ponder. In D. Fensel, K. Sycara, and J. Mylopoulos, editors, *Proceedings of Second International Semantic Web Conference*, volume LNCS 2870, pages 419–437. Springer, 2003.

[122] V. Urovi, S. Bromuri, J. McGinnis, K. Stathis, and A. Omicini. Experiences in automated workflows using dialectical argumentation. In *IADIS International Conference "Intelligent Systems and Agents" (ISA 2007)*, Computer Science and Information Systems, pages 3–8, MCCSIS 2007, Lisbon, Portugal, July 2007. IADIS Press.

[123] V. Urovi, S. Bromuri, J. Mcginnis, K. Stathis, and A. Omicini. Automating workflows using dialetical argumentation. *IADIS International Journal on Computer Science and Information System*, 3(2):110–125, 2008.

[124] V. Urovi, S. Bromuri, K. Stathis, and A. Artikis. Initial steps towards run-time support for norm-governed systems. In *Coordination, Organization, Institutions and Norms in Agent Systems (COIN@AAMAS10)*, Toronto, Canada, May 2010.

[125] V. Urovi, S. Bromuri, K. Stathis, and A. Artikis. Towards run-time support for norm-governed systems. In *Twelfth International Conference on Principles of Knowledge Representation and Reasoning (KR10)*, volume toapear, Toronto, Canada, May 2010.

[126] V. Urovi and K. Stathis. Playing with agent coordination patterns in MAGE. In *Coordination, Organization, Institutions and Norms in Agent Systems (COIN@AAMAS09)*, Budapest, Hungary, May 2009.

[127] W. M. P. van der Aalst, A. P. Barros, A. H. M. ter Hofstede, and B. Kiepuszewski. Advanced workflow patterns. In *Cooperative Information Systems*, volume 1901/2000 of *Lecture Notes in Computer*

Science, pages 18–29. Springer, 2000.

[128] W. M. P. van der Aalst, A. ter Hofstede, B. Kiepuszewski, and A. Barros. Workflow patterns home page. http://www.workflowpatterns.com/, 2009.

[129] U. Varshney. *Pervasive Healthcare Computing: EMR/EHR, Wireless and Health Monitoring.* Springer Publishing Company, Incorporated, 2009.

[130] D. Weyns, A. Helleboogh, and T. Holvoet. The packet-world: A testbed for investigating situated multiagent systems. In *Software Agent-Based Applications, Platforms, and Development Kits*, pages 383–408. Birkhauser Verlag, Basel - Boston - Berlin, 2005.

[131] J. Wielemaker and A. Anjewierden. An architecture for making object-oriented systems available from prolog. In *WLPE*, pages 97–110, 2002.

[132] M. Wooldridge and N. R. Jennings. Intelligent agents: Theory and practice. *Knowledge Engineering Review*, 10:115–152, 1995.

[133] M. Wooldrige. *An Introduction to Multi-Agent Systems.* John Willey, 2009.

[134] P. Yolum and M. P. Singh. Reasoning about commitments in the event calculus: An approach for specifying and executing protocols. In *Annals of Mathematics and Artificial Intelligence*, page 2004, 2004.