City University London
Department of Computing

Regulating competence-based access to agent societies

George K. Lekeas

A thesis submitted for the degree of
Doctor of Philosophy of City University London

December 13, 2011

Στους γονείς μου
Κωνσταντίνο και Αθηνά

Contents

List of Figures

List of Tables

List of Listings

List of Equations

Glossary

Notation	**Description**
AA	Authority Agent of an artificial agent society
bbe	Branching Bisimulation Equivalence
EPO	Electronic Purchase Order
FSP	Finite State Processes
LTS	Labelled Transition System
LTSA	Labelled Transition System Analyser
mCRL2	micro Common Representation Language 2
SOA	Service-Oriented Architectures

Acknowledgements

This work would not have been possible without the invaluable help of a number of people over the years. First, I would like to thank my supervisors, Dr. *Christos Kloukinas* and Dr. *Kostas Stathis* for their help and support throughout my degree. They were always available for inspiring conversations, always willing to read drafts and suggest improvements; really the best a student can hope for.

I would also like to thank my colleagues at City University for all their moral support and the enjoyable, long nights (and weekends) that we spent together in the office - namely *Gilberto Cysneiros*, *Tshiamo Motshegwa*, *Ping Zhu*, *George Katopodis*, *Waraporn Jirapathong*, *Stelios Papakonstantinou*, *Nazanin Khalili* and *Aravin Naren*. Special thanks should go to the academics of the Department of Computing for unveiling the wonderful mysteries of academic life and offering an enjoying environment for research. The Department of Computing kindly offered a three-year full-time scholarship under the Professional Pathway scheme, without which this study would have been impossible. My gratitude goes to *Muck van Weerdenburg* and *Simona Orzan* for helping me understand better the code for μCRL2 and *Stefano Bromuri* from Royal Holloway who offered helpful advice and feedback on reading the contents of a file into a Prolog list. The comments of the examiners, Dr. *Alessandra Russo* and Dr. *Andrew Tuson* helped me to significantly improve the presentation and contents of the thesis.

Finally, I would like to thank my family for all their moral and financial support throughout these years. None of this would have been remotely possible without their continuous love and support. It is my dad, *Konstantinos*, mother, *Athina*, sister, *Joanne*, and niece, *Elpida*, deserving all the credit for keeping the dream alive and finally making it a reality.

Declaration

I hereby declare that:

- my submission as a whole is not substantially the same as any that I have previously made or am currently making, whether in published or unpublished form, for a degree, diploma, or similar qualification at any university or similar institution
- the following parts of the work or works now submitted have previously been submitted for a qualification at a university or similar institution (only brief details required):

  ....
  ....
  ....

- until the outcome of the current application to this University is known, the work or works submitted will not be submitted for any qualification at another university or similar institution.

Date: .... Signature: ....

Print Name: ....

Preface

Advances in ubiquitous computing have resulted in changes to the way we access and use everyday applications, e.g. reading mail and booking tickets. At the same time, users interact with these applications in a variety of ways, each with different characteristics, e.g., different degrees of bandwidth, different payment schemes supported and so on. These are highly dynamic interactions, as some of the applications might become unavailable (either temporarily or permanently) or their behaviour may change. As the user has to deal with a large number of proactive and dynamic applications every day, he will need a personal assistant that possesses similar characteristics. The agent paradigm meets this requirement, since it exhibits the necessary features. As a result, the user will provide its personal agent assistant with a goal, e.g. I need a smartphone which costs less than three hundred pounds, and the agent will have to use a number of applications offering information on smartphones so that it finds the requested one. This, in turn, raises a number of issues regarding the organisation and the degrees of access to these services as well as the correctness of their descriptions.

In this work, we propose the organisation of applications around the concept of *artificial agent societies*, to which access would be possible only by a positive evaluation of an agent's application. The agent will provide the *Authority Agent* with the role it is applying for and its competencies in the context of a protocol, i.e., the messages that it can utter/understand. The *Authority Agent* will then check to see if the applicant agent is a *competent* user of the protocols; if yes, entry is granted.

Assuming that access is granted, the next issue is to decide on the protocol(s) that agent receives. As providing the full protocol will cause security and overload problems, we only need to provide the part required for the agent to play its role. We show how this can be done and how we can repair certain protocols so that they are indeed *enactable* once this role decomposition is performed.

Chapter 1

Introduction

1.1 Motivation

The advent of ubiquitous computing opens up new possibilities for applications and changes the way that users interact with a variety of devices to perform everyday tasks [115, 116]. Bulky computing devices have now turned into minuscule ones that users can take with them and use in their day to day life. As a result, small devices - e.g. mobile phones, PDAs and watches - acquired capabilities that only powerful desktop computers had in the not so distant past like surfing the Internet at high speeds and accessing e-mails. The final aim of ubiquitous computing is to seamlessly integrate all these devices into people's everyday life so they should be able to perform tasks like making bookings or participating in auctions without going through the burden of understanding the technicalities involved.

In the tasks of accessing the Internet or sending an e-mail, location and/or configuration do not play a role - the services that the user is accessing are *static*. In some tasks, however, location and/or configuration can play a crucial role. As an example, think of a tourist who is visiting a place for the first time and he is interested in finding out the nearest restaurant or tourist centre. Another example might be a purchase application that accepts card payments among other ways of payment, but there is a last minute problem with the component that deals with that aspect and a replacement has to be found or the application has to be reconfigured and the card payments taken off as an option.

Furthermore, the time span that the application will be needed for could be really small [109] and the device will have a number of providers to choose from. In addition, the applications should exhibit a goal-oriented and proactive behaviour (in the case that reconfiguration is needed or a choice between providers has to be made).

These observations raise the issue of how this kind of applications will be implemented and organised as well as how access to them will be regulated. In terms of implementation, a number of paradigms exist such as distributed objects [107] and network programming. More recent is that of Service-Oriented Architectures (SOA) [39, 94] in many cases applied through web services [63, 77].

On the other hand, web services do not address the issues of pro-active and goal-oriented behaviour; rather, the interaction is limited to pre-defined courses of interaction as specified in each party's configuration. The agent paradigm [19, 78] is well suited for this purpose as proactive and goal-oriented behaviour are amongst the fundamental notions of agency. Moreover, in complex services each agent can implement a part of the protocol to add flexibility to the approach (e.g. one agent could implement the purchase service, another the payment one and a third one could implement the shipping service).

Artificial agent societies [82, 90] can be used to model such applications as individual agents offering various services. The services will be offered to society members, so if an agent needs to make use of a service offered by one of the society member agents, it will have to join the society. Each service of the society will be represented through a set of *protocols* that the agent will have to participate in. The protocols will specify the behaviour of the participants (expressed in the form of *roles*) and co-ordinate the moves that they can make in the context of executing the protocol.

When representing a protocol, a familiar metaphor would provide the advantage of abstracting away from the technicalities and presenting the user as well as the designer with something that they can easily understand and use without problems [104]. In this context, we are treating protocols as *games* and every move made by a participant in a protocol is treated as a move made in a game. This is the case as both games and protocols are rule-governed and additional components (e.g. protocol variations) could be easily added.

1.2 Admittance of agents to semi-open societies

Agents typically have a set of goals they are trying to fulfill. In order to fulfill these goals, an agent may need to get access to a set of resources relevant to this goal and collaborate with other agents to achieve sub-goals [109]. However, these resources will have to be accessed from a variety of locations, in a variety of ways and each one of them will exhibit different properties - varying degrees of bandwidth on the connection they use for example or varying accuracy, price and so on. We choose to represent the agents providing these resources as an artificial society. Such a society may be identified by a set of network locations, will be governed by a set of rules and will be associated with a degree of openness. Agents requiring a resource will need to join a society that holds the resource and conforms to the rules of that society if admitted to it.

We look at *openness* from a membership viewpoint [27], not from one considering agent architectures, standards or ontologies. In this context, new agents can be accepted to the society but they will have to go through an application process first. The application will be assessing the agent's *competence* to interact with the other agents in that society; informally, the agent will be allowed entry to the society if it can *make conversation* with the other member agents. We will be defining *competence* formally later in Section 3.9, as well as describing the interactions between an agent, on one hand, and a society on the other. The latter will be represented in the interactions by its *Authority Agents*; these are agents that are assumed to know all the societal rules regarding what protocols the society makes available and what roles participate in each of them.

We will be using UML [41] to represent the activities of the interaction between agents and societies in the form of activity diagrams. The use of the extended set of UML diagrams as the modelling language for agent-based systems [52] has been proposed as the result of agents having richer properties than objects [79] and, also, some UML diagrams (e.g. class diagrams) lacking formal semantics [88].

Our interest at this stage, however, is to represent the various interactions between the applicant agent and the authority agent of the society the applicant wants to join. Put another way, our representation is intended to be at a high-level of abstraction rather than describing all the details. As a result, we are trying to show the flow of interactions by grouping interactions serving similar purposes into activities; activity diagrams are very useful in this context, due to their processing-thread semantics [79] - this will give us the flexibility of describing activities that happen in parallel. We should note here that *AUML* [12] could have been used for the same purpose as activity diagrams in both *UML* and *AUML* are based on Petri Nets; in fact, most of the enhancements suggested by the *AUML* project are now part of UML 2.0.

In general, for societies where membership is conditional the lifetime of the agent in an artificial society can be divided into three phases [66]:

(i) application phase, where the *competence* of the agent is assessed with regards to the role(s) that it is applying for (*application phase*);

(ii) the phase where the assignment of roles to the agent happens and it pursues the goal(s) for which it entered the society (*societal phase*) - this is where the interaction with the other agents happens;

(iii) the phase where the agent's goal(s) was (were) achieved or it is expelled from the society (*exit phase*).

In this context, this thesis is primarily concerned with the study of phase (i) with the additional focus on the communication capabilities of agents at the time of application.

1.2.1 Joining a semi-open society

The agent will be motivated for joining a society, because it needs the resources available therein to accomplish its goal. The application process is described in Figure 1.1.

The process starts when the Authority Agent of a society receives an application from an individual agent to occupy a *role* (or *roles*) in the society (as an example, the role of the auctioneer in an auction house). When the application is made, we assume that is judged on two grounds:

- a set of *social qualifications*; these represent anything that is not related to the societal protocols, but might be a prerequisite for using them. In the auction example, an agent might have to go

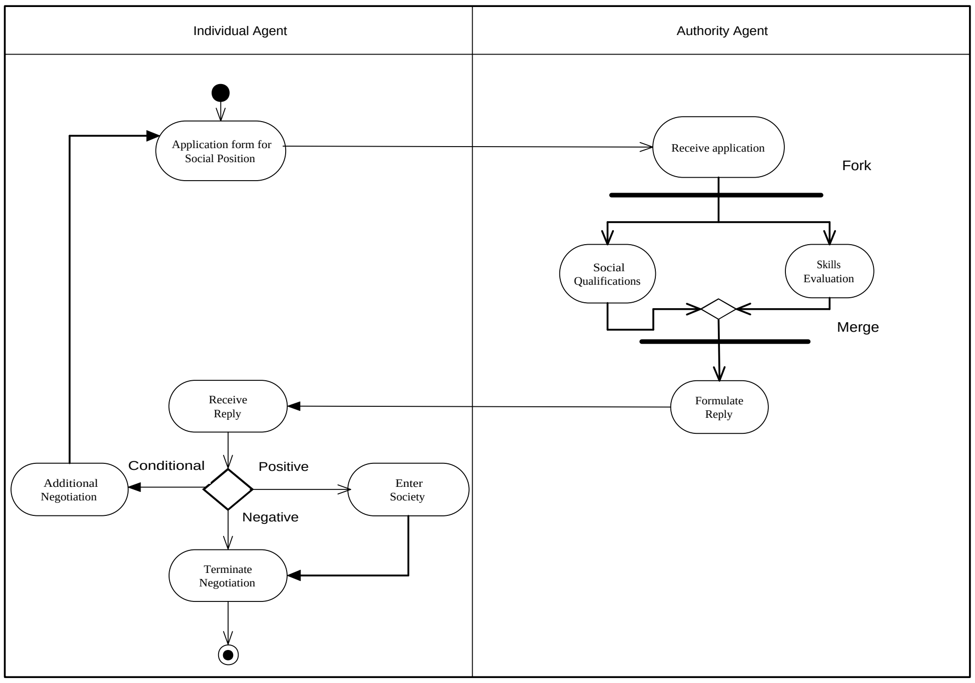


Figure 1.1: Application process for an agent to join an agent society

through a number of credit checks before its application can be processed further.

- a set of *social skills*; these relate to the communicative acts that the agent in question is capable of uttering. These acts will be checked against the protocols that the agent will participate in under the role(s) it is applying for, should it be accepted as a member to the society. An example of skills could be acts like *bid*, *register* and so on. In judging that, all other agents will be assumed to behaving according to the protocol rules.

In the case that the agent receives a *positive reply*, then it enters the society while if a *negative reply* is received it will have to seek another society that fulfills its goals or acquire the new social skills and/or social qualifications needed and then apply again for another - possibly the same - role. There is, also, the possibility of a *conditional reply* in which case the agent will have to enter another negotiation cycle - we are assuming that nothing will have changed in the meantime as a result of which the agent might have changed its opinion in relation to joining or not the society.

For the remainder of this thesis, we will be assuming that the agent has all the *social qualifications* needed and we will be focusing on verifying its set of *social skills* against the protocol requirements. We will not be looking at conditional replies either and focusing on providing the agent with a **positive** or **negative** response as to whether it is allowed to enter the society or not.

1.2.2 An agent's lifecycle in an artificial society

Once the agent's request for joining the society is accepted, the agent will need to be provided with descriptions of the protocols that it is going to be involved in as a member of the society. Each of the protocols will relate to a number of roles and sets of communicative acts that the agent can use in these protocols, depending on what role it assumes in them. During the time that the agent is part of the society, the set of protocols needed to interact with other agents may not need to remain constant. In an agent-oriented application of a business application domain, for example, it is often convenient to have every user represented by an agent so that the user accesses business applications and business processes via the agent (similar to the way people access the web via a browser). In this setting, if the user of the agent is promoted (thus, been involved in more and, possibly, different protocols or in the same protocols but with different roles), the user's agent will need to accommodate the new assignment as well. The protocol assignment part, therefore, is not static but might require the agent, while in the society, to have to learn to cope with more communicative acts. This is depicted in Figure 1.2. Moreover, we need to ensure that the agent roles in the new set of protocols that the agent receives does not contradict the ones it has at the moment, i.e. there are no *role conflicts*.

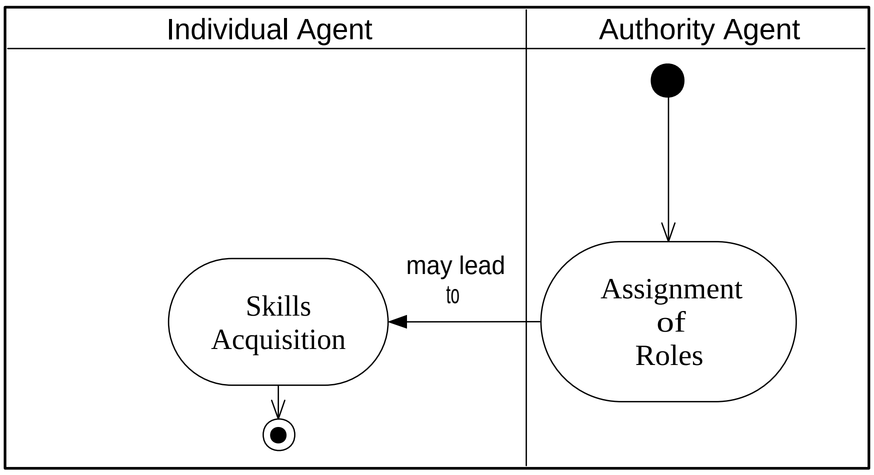


Figure 1.2: Assignment of roles to an agent

In this work, we will not be considering the dynamic part of skill acquisition and role allocation, but we are only concerned with the allocation of role(s) to the agent when it enters the society. Moreover, as we only look at the static nature of role allocation, we do not include in our analysis any *role conflict* issues.

1.2.3 Leaving a society

The status of the agent as a society member will need to be revised over time. It might be the case that the agent has achieved its goal, in which case it can leave the society (e.g. in the auction scenario because the auction ended or the agent acquired the item it was bidding on from another auction). On the other hand, the society might need to revise its membership, as a result of the agent failing to comply with some of the protocols or other constraints on membership (e.g. time constraints due to limitations on resources availability).

The process in the form of an activity diagram is shown in Figure 1.3.

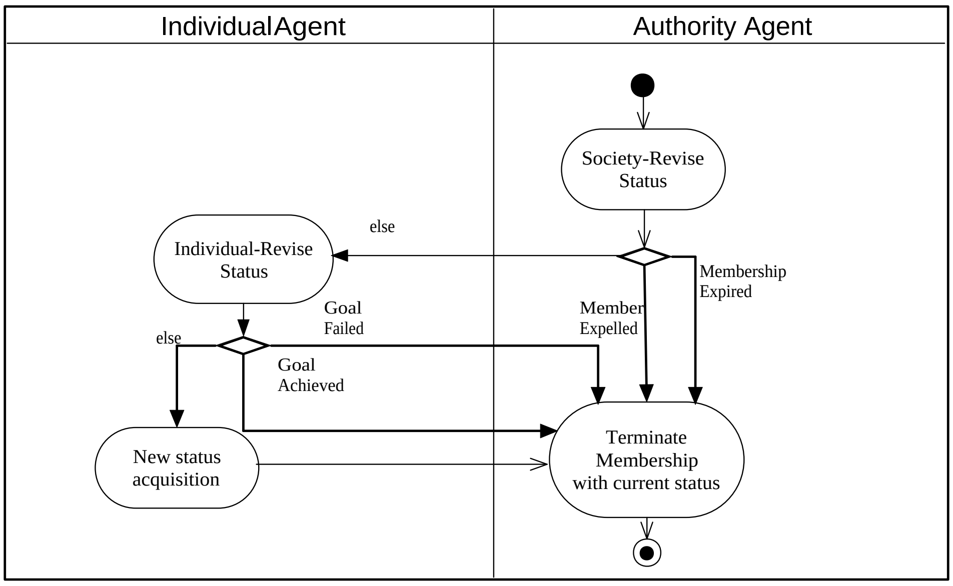


Figure 1.3: An agent leaving a society

As we will not be looking at this phase, we will just assume that the agent will leave the society once its goal has been achieved or the society will expel the agent if there are any constraints that the agent violates. Furthermore, we are not interested in the possibility of the agent breaking the rules of the protocol, as we only look at whether the agent can converse with other agents participating in the protocol. We are not interested in its actual behaviour once it joins the society, but on its *potential* to generate conversations with the other member agents on the basis of its communication skills.

1.3 Issues

Section 1.2 presents the whole lifetime of an agent within an artificial agent society. This thesis will focus on the first part, namely the admittance of an agent into an artificial society. In using this representation (*artificial agent societies* where member *agents* provide services through *protocols* in which participant agents assume *roles*) a number of issues arise.

(Issue 1) *how do we formulate a game-based version representation of a protocol in terms of roles and how is access to the society regulated*?

(Issue 2) Assuming the agent is granted access to the society, the next issue is that of the knowledge it gets upon entrance. Sure enough, we can provide all the details of the protocol but this will create performance and, maybe, security problems. The issue to deal with, then, becomes: *can the agent be given a minimal protocol description that it can follow without worrying about the full protocol complexity*?

(Issue 3) Furthermore, one of the issues with protocols is that of protocol decomposition [77], i.e., the breaking of complex ones into their constituents. If the protocol is a well-designed one, then it should be possible to decompose it into the constituent protocols (one for each role participating in the protocol) and the resulting protocols should be implementable, i.e., they should not contain any information not known to the agent when it has to make decisions as to how to proceed in the course of the protocol. In this case, the question of whether the protocol is well-designed, i.e., is implementable by the participant agents or not is raised. The question that we need to look at is: *how can we verify that a protocol is well-designed? If it is not, can it be repaired*?

1.4 Aims & Objectives

1.4.1 Aims

The aims of this thesis are:

(Aim 1) (relating to (Issue 1)) Provide a framework based on the games metaphor that:

- represents protocols of an artificial society;
- links the representation of protocols to social notions (roles, skills);

(Aim 2) (relating to (Issue 1)) provide a computational framework that can assess the application of an agent wishing to join the society;

(Aim 3) (relating to (Issue 2) & (Issue 3)) provide a framework that can decompose a protocol into smaller ones containing the minimal information that the participant will need to know and assess if it is well-designed;

(Aim 4) (relating to (Issue 2) & (Issue 3)) provide a framework for repairing the protocol if it is not well-designed.

1.4.2 Objectives

The objectives of the project are:

(Objective 1) (relating to (Aim 1)) produce a framework that uses the games metaphor as the main representation mechanism [104]. We chose the games metaphor as users are familiar with the concept. Moreover, it is an intuitive metaphor as games and protocols are both rule-governed.

(Objective 2) (relating to (Aim 1) & (Aim 2)) We aim at producing executable specifications, which are event-based and structured; the protocols as games representation meets these requirements as it is based around the concept of events (moves made by the players). We represent moves by considering the Event and Situation Calculi frameworks to represent the evolution of the formulated protocols. We are interested in the evolution of the protocol in cases where focus is on the global state or on concurrent actions or property changes over time and those two frameworks have proven to be well suited for this purpose.

(Objective 3) (relating to (Aim 2)) evaluate the approach through the use of protocols with different structures. One structure that might cause problems is that of loops, as it might mean that we run the danger of looping continuously and not reaching any result. We need to filter out these moves and allow them to occur a certain number of times. Another category that is of interest is in protocols where we have two instances of the same role - in this case, both instances should receive the same behavioural description.

(Objective 4) (relating to (Aim 3) & (Aim 4)) use a technique that will allow us to get a minimal version of the original protocol containing only the rules pertinent to a specific role.

(Objective 5) (relating to (Aim 3) & (Aim 4)) use the aforementioned technique in order to determine if the protocol is well-formed as well as repair it so it has this property in the end.

1.5 Research Hypotheses

In the course of this research, we made the following assumptions:

(Hypothesis 1) We take a bird's eye view of the society and look at it as a collection of norms encapsulated in protocols and roles - nothing else (e.g. reputation, trust) is of interest.

(Hypothesis 2) The strategies of the agents are, also, not taken into consideration as they cannot be observed.

(Hypothesis 3) We look at societies whose admittance requirements will only depend on the protocols. That is, admittance to the society is only judged on the basis of the agent's ability to participate in the relevant societal protocols. We do not take into account the current or future composition of the society (in terms of numbers of agents playing specific roles) and how additional preferences regarding its composition or any other aspect can affect agent's admittance. For example if an auction society has reached a pre-defined maximum number of bidders, another one applying for a bidder's place who can perform well in the bidder's protocol(s) will not be rejected on the basis of a large number of bidders already present in the society.

(Hypothesis 4) We only look at the message exchanges between societal agents in the context of pre-defined protocols and roles. We also assume that this exchange is free of any further constraints, e.g. due to environmental models. If an exchange is allowed by a protocol and the agent selects to utter that message, then the exchange will happen.

(Hypothesis 5) We are not interested in the architecture or communication language of the individual agents. We assume that there is a shared *lingua* and that agents themselves are responsible for ensuring that they can understand the language spoken by other agents before they make the application for entering the society.

(Hypothesis 6) We are not interested in the low-level primitives of agent communication. We assume that messages will always arrive to their recipients and any problems (e.g., network delays, concurrent communication and so on) are outside the scope of this work.

(Hypothesis 7) When an agent is checked against the protocols that it will play as a society member (if admitted to the society), the check is done assuming that the other participants are ideal and collaborative. This means that they can perform all moves prescribed by the protocol and they always collaborate. As mentioned in (Hypothesis 3), the current composition of the society is not taken into consideration at all. This means that the new agent admitted might posses skills that the agents currently in the society cannot match. However, this approach has the following advantages:

(a) it can serve as a first pass. Assuming agents that can realise all moves, we can discard applicants who do not meet our expectations. This is much easier and effective than taking into consideration the current composition of the society and what moves each of its members can perform;

(b) it provides opportunities for expansion. As new agents are added, the skills that cannot be put in use at the moment might prove extremely useful later on. For example, an agent that can take Visa payments, when no other agent in the society can make a Visa payment now.

1.6 Description of the approach and architecture of the techniques used

This work aims at providing a framework for deciding if an agent is competent to join a society offering services it is interested in and, if accepted, provide it with the minimal enactable protocol with actions only pertinent to its role. The approach followed to achieve this is:

1 the agent applies for entry to the society providing his *communication abilities* (i.e., the messages it can utter/understand);

2 the Authority Agent of the society uses the game-based formulation of the protocol (see Chapter 3 for the details) to decide if the agent is *competent* to join the society or not;

3 provided it is considered to be competent, it will need to be provided with information on the role it is taking up. This will involve creating a role specification from the protocol one. The technique we use is based on an automata-theoretic technique called *branching bisimulation equivalence*(*bbe*) Chapter 4 for the details).

4 the role specification is then checked for enactability.

- if it is found to be non-enactable, then the original protocol is modified appropriately (check Section 4.3 for the details).

The process is shown in Figure 1.4.

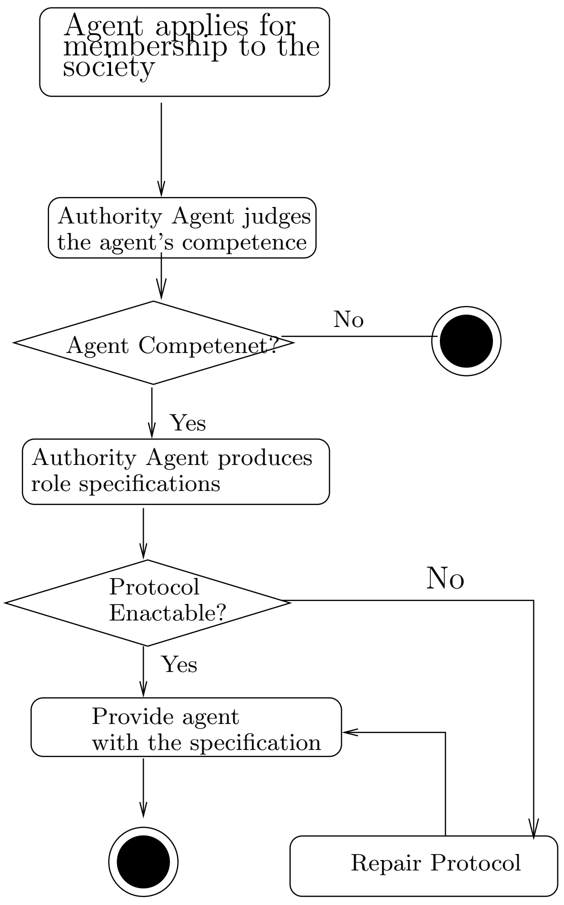

Agent applies for membership to the society
Authority Agent judges the agent's competence
Agent Competenet?
No
Yes
Authority Agent produces role specifications
Protocol Enactable?
No
Yes
Provide agent with the specification
Repair Protocol

Figure 1.4: Our approach

1.7 Structure of the Thesis

This is the introductory chapter, where we provide the motivation for the research and outline its aims and objectives, as well as state any assumptions made and listing the relevant publications. The rest of the thesis is structured as follows:

Chapter 2 presents the background to our work. The concepts that form the cornerstones of our work are presented briefly; namely the concepts of *agent societies*, the original *game-based protocol representation framework* and *roles*.

We, then, move on in Chapter 3 to describe the theoretical aspect of our model. We describe how to model roles in games protocols and how to compose them to compose the societal protocols. This is followed by a definition of how a protocol can be represented in both *Event* and *Situation* calculi formulations. Once the agent is accepted into the society and should be given the protocols it has to enact, the question that arises is whether it is competent to participate in the protocol or not. We define the agent's strategy and, on the basis of that, differentiate between different degrees of competence. Finally, the relationship between the game-based approach that we use to describe the protocols and the LTS one we are using for representing them is discussed.

Chapter 4 looks at the algorithms for meeting our objectives, namely *de-composing the protocol into roles*, *repairing it if needed* and *providing the agent only with the minimal set of information required to enact its role.* We discuss three different approaches for repairing a protocol, ranging from repairing every single problematic transition to a selected subset of them.

Once the theoretical framework and the algorithms have been discussed, in Chapter 5 we evaluate cyclic and acyclic protocols. We look at examples in both formulations - *Event* and *Situation* Calculi. We also present an example of breaking down a protocol that contains two instances of the same role, an auction with two bidders. If the protocol is well-designed, then it should provide the same set of options for both bidders as they enact the same role.

Chapter 6 reviews related work concerning competence (or *conformance* or *interoperability*), extracting rules pertinent to a role from the description of the whole protocol and breaking down a protocol into component roles. A brief account of each approach is given as well as a comparison with the game-based approach that we adopt in this work.

Finally, Chapter 7 concludes the thesis by presenting its main findings. It also draws on directions for future research.

1.8 Previous Publications

The thesis contains materials from the following publications:

(Publication 1) *Agents acquiring Resources through Social Positions* [66]. This publication describes the top-level framework for all stages of an agent's lifetime within a society. These include the application for entrance, how it might evolve whilst in the society and the point at which, voluntary or not, it leaves the society.

(Publication 2) *An agent development framework based on social positions* [65]. In this publication, we discuss and link the main components of our approach, namely roles and skills. Moreover, an attempt is made to look at the problem of dynamically assigning new roles to the agent. It also contains an initial attempt to define the concept of *social position* as a placeholder for roles, which we have not explored any further as it was not required to meet our aims.

(Publication 3) *Competence Checking for the Global E-Service Society Using Games* [105]. In this publication, we present the game-based framework for representing protocols. Furthermore, we describe the evolution of a protocol in both time-independent (Situation Calculus) and timed (Event Calculus) games.

(Publication 4) *From Agent Game Protocols to Implementable Roles* [59]. In this publication, we present an algorithm for producing an individual role specification from a protocol. We, also, address the problem of protocols with structural problems (non-enactability) and propose an algorithm for repairing them.

Chapter 2

Background

2.1 Introduction

In this chapter, we look at the main concepts around which our work is built and present the work already been carried out in these areas. The presentation is done in a sequential order so that the next term will depend on the previous one and will be motivated by that. More specifically, we discuss the concepts of *agent societies*, *game-based representation of protocols*, *roles (and skills)* and *competence.*

2.2 Agents

In recent years, there have been major advances in the way we perceive and interact with computers. These advances were mainly due to the progress made in the areas of distributed systems and networks as well as the reductions in size and the increased penetration that computer devices enjoy in our everyday life.

As a result, in contrast with the past, where the size of the devices as well as the absence of open area networks made it impossible for the user to interact with computing devices while on the move, it is now perfectly possible to use small computing devices embedded in items we use everyday (such as mobile phones or MP3 players) for performing tasks that would otherwise be performed from our desktop. As an example, many of us would use our mobile handset to connect to the Internet and participate in auctions, book tickets, read and/or send emails, edit office documents and so on.

In conducting these applications, one option is to have the end user fully involved in initiating and conducting them, i.e., if the user wants to purchase a specific item in an auction, one option would be to look at all the auction sites himself, evaluate and follow all the auctions that this item features in. However, a more realistic and desirable way forward, would be to allow the software running on the device make some decisions and give it a degree of autonomy so the interaction required from the user would be kept to a minimum.

This way repetitive tasks can be simplified and a less error-prone system can be built. In the aforementioned auction example, it would be much easier if the software running on the mobile phone or the PDA could detect that a new auction came up with an item that interests the user and joined it automatically.

In order for the software to work that way, it should exhibit a high degree of autonomy and be able to reason about certain properties of the user. Furthermore, it should be goal-oriented and focus on generating different plans for achieving the user goal(s). Finally, it should be able to operate in an open context (environment) and exchange information with other software entities of the same kind.

This description fits naturally the description of a *software agent.* There is no commonly agreed definition of an agent, however there is some agreement over the characteristics that a software entity should exhibit in order to be classified as an agent - these are as follows [19, 78]:

- encapsulates its own state - that is, every agent has its own view of the world and that might be different to the view of other agents in the system;
- the environment where the agent is situated is dynamic and open - there are changes occurring that the agent must register (those that are of interest to it);
- it should have the ability to reason on the basis of observations registered from the environment;
- it should be able to communicate with other agents within the same environment and execute actions based on data acquired from the observation of the environment, the local state and the internal reasoning that took place.

The definition above focuses on a number of characteristics that collectively define the *weak notion* of agency, i.e., the absolute minimum characteristics that an application must possess in order to be classified as an agent. These would be *autonomy*, *social ability*, *reactivity* and *pro-activity*. Furthermore, agents can exhibit human-like characteristics, such as benevolence, fairness and veracity, and these would define the *strong notion* of agency.

2.3 MAS & Artificial Agent Societies

As an extension to the term agent, a *collection* of agents can be viewed as a *Multi-Agent System*. A Multi-Agent system is a system made up of software agents, where ([19]):

- not all problems can be solved with information that an agent of the system possesses;
- there is no global system control;
- data is decentralised; and
- computation is asynchronous.

In the context of the environment described in Section 2.2, physical everyday devices are transformed into computing devices. On the other hand, due to the technological advances and the availability of bandwidth, they would able to migrate from one network/platform to the other with ease. As a result, the question of *how is access to these resources going to be regulated* arises.

Our view is that these network resources can be organised around the concept of *artificial agent societies* [82, 90]. These are viewed in different ways in the literature. Davidsson defines it as *a collection of agents interacting with each other under the presence of some norms and social laws* [26]. Moreover, a distinction can be made between *agent societies* and *agent communities* in that the latter should represent agents in closer proximity to those belonging in an agent society. In the same context, an *agent society* is defined in [93] as a collection of agents with similar interests working towards the achievement of a goal (the focus been on the collaboration between the agents rather than their proximity).

Formally, Artikis defines an open agent society as follows [5]:

- a set of agents;
- a set of constraints on the society (a description of the societal norms and rules that the agents have to obey);
- a communication language;
- a set of roles that the agents can play;
- a set of states of affairs that hold at each time at the society; and
- a set of agent owners.

Furthermore, Davidsson expands this definition in [28], to include the following items:

- a set of agent designers;
- the environmental infrastructure in terms of communication and computation;
- the environment owner (the organisation or institution that defines who enters the society and what is the configuration of the environment - roles to be assumed, communication language and so on); and
- the environment designer.

The additional items in this definition are due to the fact that not all system stakeholders are considered in [5] and any constraints due to them will need to be acknowledged and explicitly stated. The societal constraints element of the definition by Artikis would cover all the constraints but it is useful to look at constraints on a per stakeholder basis.

In [33], Doran defines the artificial society as a set of heterogeneous and located inter-related agents. There is no assumption about the type of agents, just the assumption on the society side that it is *open*, i.e., any agent can join and leave freely. Society is formally defined as a *network graph* where every node is a site and there are *bidirectional links*, called channels, to join nodes (essentially sites will correspond to agents and links will correspond to ways of communication). The agent members of the society will be situated on a site and will have a unique identifier, an owner, an internal unobservable selection strategy, might be mobile, exchange information via the channels specified and can act on society items.

In [32], Dignum focuses more on the normative aspect of the specification: an organizational-oriented view is taken upon the definition of an artificial society. More specifically, the society is described by an organisational model and an individual (agent) model. The organisational model is defined in deontic logic as a tuple **OM** = (R, CF, I, N), where:

- R is the set of role descriptions;
- CF is the communicative framework for the society (as agent will be heterogeneous and speak different languages, this framework will be used to provide common grounds for understanding between the agent members);
- I is a set of scripts to be realised in interaction scenes. Interaction scenes can represent states in a protocol execution, while scripts will essentially represent an execution path for reaching these states; and
- N is the set of social laws (or norms) - this is specified in terms of contracts that, if broken, will result in the offending agent been sanctioned.

In [48], the authors view society as a medium for agents to meet and interact. They are characterised by their openness and the social norms and laws that the society will need to have in place to enforce the order in it.

The information from the approaches described above can be summarised in Table 2.1:

Property	Artikis	Davidsson	Doran	Dignum
Set of Agents	✓	✓	✓	✓
Set of Constraints	✓	✓	✗	✓
Communication Language	✓	✓	✗	✓
Set of Roles	✓	✓	✗	✓
Set of States	✓	✓	✗	✓
Set of Agent Owners	✓	✓	✓	✗
Set of Agent Designers	✗	✓	✗	✗
Environmental Infrastructure	✗	✓	✗	✓
Environment Owner	✗	✓	✗	✗
Environment Designer	✗	✓	✗	✗

Table 2.1: Comparison of artificial agent society definitions

In our approach, artificial agent societies are viewed as a collection of agents having specific *skills*, i.e., communicative acts that they can perform. These societies are *normative* societies and are organised around the concepts of *protocols* and *roles.*

More specifically, a *protocol* will describe a pattern of interaction between two or more agents and will describe what actions they are allowed to take and at what states of the protocol evolution. The description of the protocol will be in terms of behavioral components, i.e., in terms of what the agents can do when they engage with the protocol. The behavioral recipes will be expressed in terms of *roles* - every role will specify *the expected behaviour of an individual in a given context* [14].

As an example, we can consider the case of a user seeking to purchase an item (e.g. a mobile phone) from an online auction. The user will pass the information of the item he wants to purchase to his representative agent, who will then take over and look for auction services from which it can purchase that item.

This will involve evaluating a number of auction societies with regards to whether the item it is interested in is available as well as whether any other auction-related preferences are met (e.g. shipment has to be made within two days of the auction end). It will then have to join these societies and participate in the auction(s) for the item it is interested in.

In this context, if an agent needs to get access to the resources offered by an agent society, i.e., the services that the agents offer, such as the auction, payment and shipping agent societies, it will need to request and gain access to that society. In other words, the question now becomes: *what would be the criteria that we can use to grant access to an agent into an artificial agent society.*

2.4 Entry to Society

In representing collection of agents as artificial agent societies governed by normative rules expressed in the form of protocols and roles, we need to make a decision concerning how the society will accept (or not) new members. There are four possibilities regarding the openess of the society to new members [27]:

- *open* societies, where agents can join and leave at any time without any restrictions;
- *closed* societies, where membership is pre-defined and static; any

agent is either a member and it is allowed access to the resources offered by the society or not;

- *semi-open* societies where membership is granted on successful judgement of a submitted application form; and

- *semi-closed* societies, where entrance to new members is not allowed, but agents external to the society can nominate agents already in the society to act as their representatives.

In light of the type of systems we want to represent in our approach (highly dynamic, adaptable, critical and fault-tolerant systems) the option of closed societies seems too restrictive for our purposes. On the other hand, the option of completely open societies might lead to problems such as poor communication and/or communication deadlocks between member and non-member agents – the same applies to semi-closed ones, as the agent will have to entirely rely on agents who are already members of the society.

A more realistic approach seems to be the restriction of agent entry to the society on the basis of satisfying certain conditions. As long as these conditions are met, the agent can become a member of the society until it accesses the resources to achieve its goals or it decides to leave the society.

A choice has, thus, to be made regarding a number of factors that will control the agent's admittance (or not) to the society, namely:

- what will the admittance criteria be;

- who will conduct the check of whether the agent meets the criteria or not; and

- how will the societal behavioral rules, i.e., protocols be represented.

We look at each one of them in the following Sections.

2.4.1 Admittance Criteria

The first item we need to decide is on the basis of what criteria would the new agents be admitted into the society. As stated in Section 1.5, we are interested in all societies who use protocols to judge the agent's *competence* as part of their admittance criteria. We concentrate on that part only – societies can have other criteria as well. As an example, in a society with one auctioneer and a million bidders if a new bidder agent applies for membership, the current number of agents who are bidders in that society already makes no difference (as well as other properties of the applicant agent, e.g. the good or bad reputation it carries from other societies that it was a member of in the same role or in similar roles).

The question, thus, of whether an agent should be admitted to a society or not is transformed to a question of whether it should be able to understand and use the societal rules, as the latter are expressed in the relevant protocols.

2.4.2 Authority Agents (AA)

Every society that is *semi-open* and accepts members only by application, will need to have a way of regulating access [27]. The role of this component will be to assess the membership applications received and reach a decision as to whether the application will be accepted or not.

In our societies, this task is performed by one (or more) *Authority Agents (AA)* - their task is to know all societal rules and how they are related to the society's protocols, i.e., what roles participate in each protocol and what are the conditions for making a move within each protocol. On the basis of this knowledge, they need to decide if the applicant agent will be granted or denied entry.

2.5 Game-based representation of protocols

As the decision of the *Authority Agent* will be based on the actions that the agent can perform in relation to the role it is applying for and the protocols in which the role participates in, we need a suitable representation of protocols.

In the literature, a number of choices exist with respect to modelling protocols. These include different types of automata [16, 57, 64] such as Finite State Automata (both deterministic and non-deterministic), B*ü*chi automata and so on. The notion of automata is an attractive formalism as it is easy to understand and implement - the notation of deterministic finite automata, specifically, has been widely used [31, 89].

Another approach is to model the protocols as games [106], emphasising the similarities between the two approaches. The protocol participants can be seen as players, the rules of the protocol as the rules of the game and the participants decision on their next move they make could be seen as part of their game strategy. Moreover, the starting point of the protocol would be the starting point of the game and the final point(s) would be when the game ends - depending on the final point reached, as there might be many, one or the other participant would be declared the winner. Finally, it can represent any of the protocols represented as an FSA and can offer a higher level of granularity in the form of subgames.

In [68], a game description language is specified for playing online games consisting of a *Game Manager* that will be connecting a number of *General Game Playing* sessions; the players do not know the rules of the game in advance, but they can understand the language in which they are written. The development of the language was carried out as a response to the large explosion of state space for a number of games (e.g. chess) and the inability to differentiate - in a standard FSA - between changes due to player moves or game dynamics. As an example when two players are playing a game of Tic-Tac-Toe, we need to differentiate between the case that the game terminated because one of the players won or the network went down.

As a result, the *Game Development Language (GDL)* was introduced, based on *datalog*⌉ [47]. This language describes the relations between the players and the state of the game. The players of the game are defined via the *role* relation, while the state of the game is described via a series of relations - *init* to describe what is true at the beginning of the game, *control* to describe the player who is in control, *next* to describe the next player who is to make a move, *legal* that describes what are the legal moves in a state and *does* to describe the moves that a player makes (or has made). Finally, the *goal* rules will describe the goals of a player and what he will be trying to achieve in the game.

On the other hand, the *Game Manager* is responsible for synchronising the state between the players and receiving their moves. Every player will be assigned a single role - every time that the player has to make a move, the player will send the move to the *Game Manager* and it will be the responsibility of the *Game Manager* to assess its validity and either apply the move if it is valid or randomly select a move and apply it if the player submitted an invalid move or did not submit a move at all. It will, also, be its responsibility to keep the clock for the game, i.e., if there are any time constraints regarding the selection of moves by the players, then *Game Manager* will enforce that. It can, also, provide some time to the players for reading the rules before the start of the game. All the communication in *GDL* happens through HTTP connections.

Our approach is based on the existing games framework described in [103, 104, 106]. In this framework, for each game we need to specify the following components:

- the state of the game;
- the moves that the players can make; and
- the rules of the game.

Moves will describe what actions are available, while the rules will describe what actions are legal. The moves are represented by Prolog *select/2* predicates in the form `select(Player, Act)`. The state of the game is represented by *rule/3* predicates as follows: `rule(State, Property, Conditions)` meaning that property *Property* holds in state *State* if conditions *Conditions* hold. Any property that we want to check against is considered to be a goal of the player and check for multiple properties can be done recursively (by using a Prolog list construct).

When a player makes a move, this will change the state of the game by producing some effects; these are represented by *effects/2* predicates as follows: `effects(State, Actions)` defines the effects of actions *Actions* on the current state *State* of the game. This is further extended to the *effectsRule/4* predicate, which is defined as `effectsRule(State, Action, Conditions, Actions)`. If the game is in state *State*, if conditions *Conditions* of action *Action* are met, then actions *Actions* are carried out. *Action* is the move that the player(s) decided to make and we need to look at *Conditions* one at a time, checking that they are met; for example, in a request-query protocol we need to check that a query has been made before the agent produces a response. In other words, if *Conditions* of *Action* are verified in the current state of the game, *Action* is a valid move and its effects will be described by *Actions*.

The game itself is defined as a sequence of steps as shown below:

- check that the state we are currently in is not a terminating state;
- find the next *valid* move,
- apply its effects.

A distinction is made between moves that the game makes available to the players (*available*), the moves that are allowed at a certain stage of the game (*legal*) and the intersection of those (*valid*).

In order to play a game between two or more players, we have the option of filtering the moves through an *umpire*. The umpire will check whether the game is in a terminating state or not and - if not - check that a valid move has been selected, display its effects and then call itself recursively to check if the new state is a terminating state or not. If the game has reached a terminating state, this will be offered as a solution – the game can be terminated there, or the user might request more terminating paths. The program is shown in Listing 2.1:

```
umpire(Umpire, State, Result, game(State,Result)):-
        stop(Umpire,State,Result,terminating(State,Result)).
umpire(Umpire,State,Result,game(State,Result)):-
        not terminating(State, _),
        check(Umpire,Move,State,valid(State,Move)),
        display(Umpire,State,Move,effects(State,Move)),
        umpire(Umpire,State,Result,game(State,Result)).
```

Listing 2.1: umpire/4 predicate

The *check/4* predicate in Listing 2.1 can be run in both a *backward* mode - the player selects a move and the umpire checks if it is valid - or in a *forward* mode - all valid moves are provided to the player and he selects one.

As an example, we can look at the *request-reply* protocol in Figure 2.1.

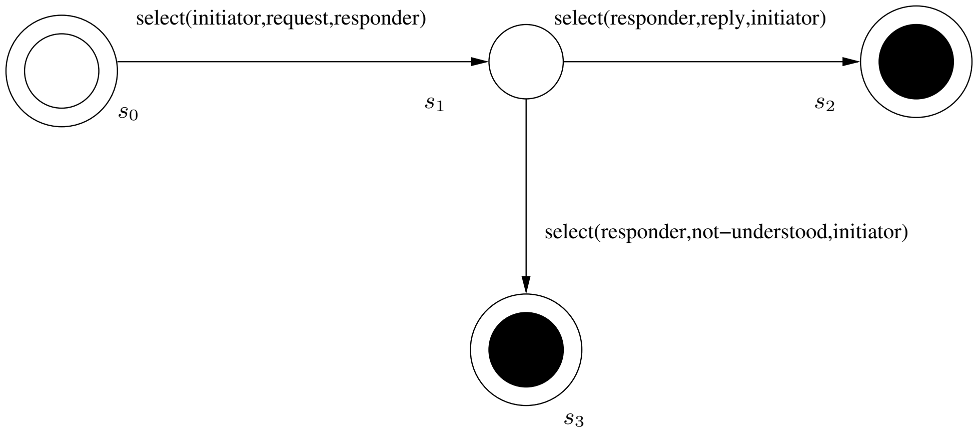

select(initiator,request,responder)
select(responder,reply,initiator)
select(responder,not−understood,initiator)
s_0
s_1
s_2
s_3

Figure 2.1: A simple request-reply protocol

It consists of two roles, *initiator* and *responder*, with the agent assuming the role of initiator making a request for information to the agent assuming the role of responder. The responder agent can, then, choose to *reply* with the requested information or with a *not-understood* move. Both of these moves take the protocol to a final state where it terminates.

The code in Listing 2.1 is saving the moves made so far by the game players in *State* and if the last move is a terminating move, *State* becomes equal to *Result* which is the end result of the game. If the last move selected is not a terminating move, then the state of the game is checked for another valid move. Once selected, its effects are displayed and the *umpire* predicate is called again to check whether it is a terminating one or not; if not, the cycle continues.

The *stop/4* predicate simply calls the *terminating/2* predicate, which contains information about the moves that bring the game to completion along with any pre-conditions that will have to be met for a move to be a terminating one. The same holds for the *check/4* predicate; it merely calls the *valid/2* that looks at the current state of the game and decides on the next valid move to be selected. Finally, the *display/4* predicate is calling the *effects/2* one so that the effects of the last move selected in the previous step will be reflected on the current state of the game.

The request-reply protocol that we are looking at has two terminating moves, *reply* and *not-understood.* These can be selected provided that a request move has been made by the initiator of the protocol beforehand. Assuming a rule of the form

```
rule(requestreply(ID),terminating(reply(P1,P2,Query)),
     [next(P1),previous(select(P2,request(P1,Query)))]).
```

Listing 2.2: Example of a protocol rule

if the second player selects *reply*, it is his turn to make a move and the previous selected move has been a request by the first player then the game terminates.

In the case of non-terminating moves, we need to specify their effect on the state of the game. The effects of, for example, a *request* move in the protocol of Figure 2.1 are shown in Listing 2.3. The player who makes the move is not the next player any more, but becomes the last; furthermore, the move is added as the last move selected.

```
effectsRule(qprotocol,select(P,Act),[Act=request],
              [delete(previous(select(L,_))),add(previous(select(P,Act))),
              delete(next(P)),add(next(L)),delete(last(L)),add(last(P))]).
```

Listing 2.3: effectsOne/4 predicate

This approach also allows for games with *subgames*, i.e., games that contain other games as independent components. There is no restriction on the relationship between the game and its subgame components - i.e., we can represent subgames where once the agents enters the subgame, it has to terminate it before going back to the main game as well as subgames where the player can interleave and go back to the main game without having terminated the subgame first.

Once engaged in the protocol (game), the players will need to get information about what are the moves they are entitled to make and in which context. This is the task carried out by roles; setting a behavioural context regarding the agents' behaviour.

2.6 Summary

In this chapter, we presented the basic concepts on which we will base our approach as well as work already done on them. We will be using the *agent* metaphor to link the low-level implementation details of web services to the high-level description of service providers and service issuers. Each agent will have a number of skills, that is a number of messages they can utter (understand) in the course of a protocol. A test should be run to verify that the skills they have can lead to successful co-operation when the protocol runs. This calls for a formal framework to represent protocols & roles, as well as providing a local view for the roles which is the subject of the next chapter.

Chapter 3

Formal Framework

3.1 Introduction

In this chapter we present the formal framework of our approach, detailing how the protocols as games concept is formulated as an LTS; we show how a protocol and a role within a protocol are defined. If the definition of roles is given, then we can work out the protocol as the *parallel composition* of the role definitions. Furthermore, we present the representation - as well as the evolution - of protocols by integrating in our representation of games (a) the *Situation Calculus* where the focus is on changes of a global state and (b) the *Event Calculus*, if concurrency is a domain requirement and the focus is on local changes of state properties. The choice of the frameworks is done on the basis of seeking *executable specifications*; thus, the representation of the state as Normal Logic Programs that can produce executable specifications. Finally, we explore the different degrees of *competence* for an agent on the basis of its strategy S and use the *Netbill* protocol to illustrate them.

3.2 The Netbill Protocol

In this section, we will be illustrating the main concepts of our framework by the use of a variation of the e-commerce protocol NetBill [25, 46]. This can be used by a society that aims at allowing *merchants* to sell goods to *customers* and make use of *payment gateways* in order to collect payment. An agent wishing to enter a society where *Netbill* is available will have to apply for the role of customer, merchant or gateway depending on the goal it wishes to achieve when entering the society. In the original protocol, there are three roles - *customer* (c), *merchant* (m) and *gateway* (g)- and eight overall steps for a customer to purchase goods from a merchant and the merchant to process payment for the order through NetBill's gateway. These are depicted in Figure 3.1 and are as follows:

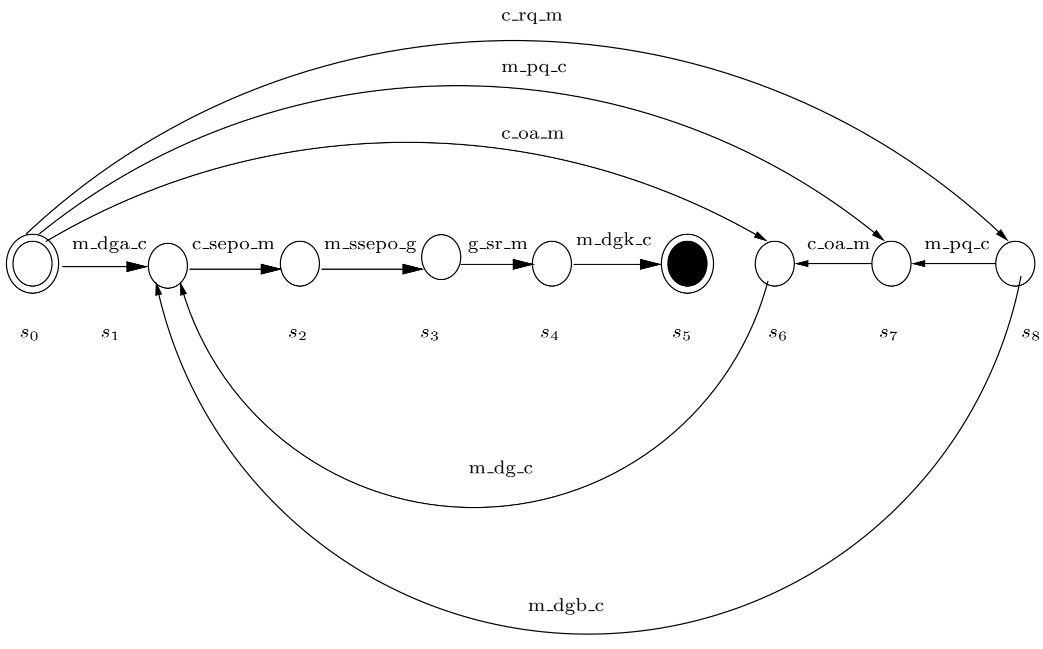


Figure 3.1: A variant of the NetBill protocol

- The customer requests a quote for some digital goods from a merchant - see the transition (s_0, $(c, rq, \{m\})$, s_8), i.e., from state 0 to state 8, labelled as $(c, rq, \{m\})$.
- The merchant provides a quote to the customer - (s_8, $(m, pq, \{c\})$, s_7).
- The customer accepts the quote made by the merchant - (s_7, $(c, oa, \{m\})$, s_6).
- The merchant proceeds to deliver the ordered goods encrypted with a key K - (s_6,$(m, dg, \{c\})$, s_1).
- The customer signs an *Electronic Purchase Order* (EPO) with the merchant - (s_1, $(c, sepo, \{m\})$, s_2).
- The merchant signs in its turn the EPO and sends it to the NetBill gateway - (s_2, $(m, ssepo, \{g\})$, s_3).
- The NetBill gateway internally checks the information on the EPO, transfers the money and ends by sending the merchant a receipt - (s_3,$(g, sr, \{m\})$,s_4).

- Finally, the merchant sends the customer the key needed to decrypt the goods it purchased - (s_4, $(m, dgk, \{c\})$, s_5).

We made the following additions to the original NetBill protocol to create one with branching structure so that we can illustrate problems when the agent has to make a decision but does not have all the information required:

- The merchant can now make a price quote directly - (s_0, $(m, pq, \{c\})$, s_7); e.g., in the case of a promotional offer.
- The merchant could select to deliver the goods as its first move - (s_0, $(m, dga, \{c\})$, s_1); e.g., when the customer has good credit and solid reputation with that merchant. In this case, the encryption method used in the delivery can be more relaxed than the normal one as the process involves a trusted customer.
- The customer might accept the merchant's quote directly - (s_0,$(c, oa, \{m\})$, s_6); e.g., when the merchant is trusted or this is a recurring order.
- On reception of a quote request, the merchant can make the quote and ship the goods directly without waiting for a formal acceptance of the quote - (s_8, $(m, dgb, \{c\})$, s_1); e.g. when dealing with a trusted customer or a recurring order. The delivery and encryption method will have to be different again, as if it is a recurring order it will mean that the customer is low on stock for this particular item.

3.3 Definition of a protocol in a society

We present a representation of protocols based on Labelled Transition Systems, following closely the game metaphor and the notation of [105], where the protocol is described as a set of player roles that interact via dialogue moves representing message passing.

A protocol interpreted as a game will need to provide structures for specifying what happens when the game starts, as well as describing the evolution of game. This is happening through the transition between states, once certain actions are triggered. In this section, we look at the LTS-based definition of the protocol and Sections 3.6 and 3.7 describe the frameworks used for specifying what applies initially to a game protocol and, also, how the game properties (next player to make a move, properties that hold at a specific state) are been modelled.

A **game protocol** ($\mathbb{P}^G$) is defined as a tuple

$$<N,\ R,\ S,\ I,\ F,\ A,\ M,\ V,\ E>,$$

where the components are as follows:

1. N is the game protocol's name;
2. R is the set of *player roles*, participating in the game protocol;
3. S is the set of game *protocol states*;
4. I is the *initial* state of the game protocol, $I \in S$;
5. F is the set of *final* states of the game protocol, with $F \subseteq S$.
6. A are the labels of the *actions* that are known to the game protocol, that is, all possible messages which can be exchanged by any two (or more) players at any time during the execution of the protocol.
7. *game moves* are described by the relation $M = R \times A \times (2^R \setminus \varnothing)$, i.e., the available actions are associated with the role that can perform them, as well as, the roles which will be the recipients of that action, assuming that an action can be performed on multiple recipients.
8. *valid moves* are defined for a role according to the state that the protocol is in and narrow down the choices of the next message to be sent by the role. They are defined by the relation $V = S \times M$.
9. *effects* of the valid moves are calculated by the relation $E = V \times S$, i.e., the transition relation of the LTS. This relation is used to determine the next state of the LTS on the basis of the current one and the move that has been selected by the agent.

For the NetBill protocol shown in Figure 3.1 we have the following formal representation:

$$<N,\ R,\ S,\ I,\ F,A,\ M,\ V,\ E>,$$

where:

$$\begin{aligned}
N &= \text{NetBill}\\
R &= \{c, m, g\}\\
S &= \{s_0, s_1, s_2, s_3, \ldots, s_8\}\\
I &= \{s_0\}\\
F &= \{s_5\}\\
A &= \{rq, pq, oa, dg, sepo, ssepo, sr, dgk\}\\
M &= \{(c, rq, \{m\}), \ldots, (m, dgk, \{c\})\}\\
V &= \{(s_0, (c, rq, \{m\})), \ldots, (s_7, (m, dgk, \{c\}))\}\\
E &= \{(s_0, (c, rq, \{m\}), s_8), \ldots, (s_4, (m, dgk, \{c\}), s_5)\}
\end{aligned}$$

3.4 Modelling Roles in Game Protocols

This section presents the formal framework for describing a role in the context of a protocol and links it to the previous notion of a game protocol. In a *role-oriented* model, a game protocol is assumed to be a set of roles, i.e., $\mathbb{P}^R = \{\mathbb{R}^R\}$; the protocol will be produced as a composition of the constituent roles.

The *role descriptors* are effectively a projection of the *relevant* components of the game protocol onto a specific role; a role, thus, is specified as a tuple $<N^R$, R^R, S^R, I^R, F^R, A^R, M^R, V^R, $E^R >$. The component R^R represents the *set of roles* that the role N^R will be interacting with during the course of the protocol, either as initiators or as recipients of messages sent (received) by the role in consideration. Any message by any other role would be considered irrelevant and illegal. The *actions* known to the role are A^R, i.e., moves that the role can perform or receive. Given that $\text{has\_role}(R, 2^{R^R})$ are all the subsets of R^R containing N^R, the known *moves* set $M^R = \{\{R\} \times A^R \times (2^{R^R} \setminus \varnothing)\} \cup \{(R^R \setminus \{R\}) \times A^R \times \text{has\_role}(R, 2^{R^R})\}$ specifies whether an action can be performed by the role itself (and by which other roles it can be received) or whether it can be performed by some other role and received by the role in question (possibly among others).

We should note here that each role is represented by an *LTS* . In order to work out the possible interactions between the roles, we need to *compose* these LTS using *parallel composition*, i.e., any actions that are common between the roles will have to be performed at the same time.

3.5 From Roles back to Game Protocols

If the role descriptors for the roles participating in a protocol are available, then we can construct the protocol by taking the *synchronous composition* of the role LTS. The steps for the composition and, thus, the final protocol will be as follows:

- The roles of the protocol should be the union of all the rolesets from the role definitions ($R = \bigcup_{r \in \{\mathbb{R}^R\}} R_r^R$).
- The state of the game can be defined at any moment to be the composition of the state of all its roles ($S = \prod_{r \in \{\mathbb{R}^R\}} S_r^R$).
- The initial state would be the state where all roles are in their initial states ($I = \prod_{r \in \{\mathbb{R}^R\}} s_{0_r}$).
- The final state would be the state where all roles are in their final states - therefore, any combination of role terminating states will be a terminating state for the game protocol as well ($F = \{s : \forall r \in \{\mathbb{R}^R\}.s_r \in F_r^R\}$).
- The available moves for the game protocol would be the union of available moves of all role moves and the same will hold for the game actions ($A = \bigcup_{r \in \{\mathbb{R}^R\}} A_r^R,\ M = \bigcup_{r \in \{\mathbb{R}^R\}} M_r^R$).
- The valid moves of the game will comprise of those moves where all roles are in a state to make a valid move - transitions with the same labels in their LTS's will be synchronised (as this is a requirement for parallel composition). This means that both roles should be in a state where they can select to make the respective moves ($V = \{(s,m) : s \in S, m \in M \wedge \forall r \in \{\mathbb{R}^R\}.m \in M_r^R \Rightarrow (s_r, m) \in V_r^R\}$).
- The effects of a move will be derived by looking at the effects of each move on an individual role, when all other roles remain in the same state. The new state of the protocol will be the result of the composition of the new state of the role making the move with the states of the other roles. The formal representation is :

$$\begin{aligned} E \;=\; & \{(s_1, m, s_2) : (s_1, m) \in V \wedge \forall r \in \{\mathbb{R}^R\}. \\ & ((s_{1_r}, m) \in V_r^R \Rightarrow (s_{1_r}, m, s_{2_r}) \in E_r^R) \vee ((s_{1_r}, m) \notin V_r^R \Rightarrow \\ & s_{2_r} = s_{1_r})\}. \end{aligned}$$

In the case of breaking a protocol into its constituent roles, the parallel composition of the resulting role LTS will not always give us back the original protocol (or one that is *branching bisimilar* to it); see Section 5.4.1 on Page 94 for a counter-example.

3.6 Game evolution in Situation Calculus

We are using the games metaphor [103, 104, 106] in order to describe the interactions among participants in protocols of artificial agent societies as interactions and moves made in the context of a game. The idea here is that protocol enactment happens through a series of communicative acts, which are viewed as moves made between players in the context of a game. The game does not, necessarily, need to have a winner or a loser in the traditional sense (in an order protocol, the supplier wins by making a sale and the merchant wins by acquiring the item(s) it wanted). Termination of the game signals that the protocol has reached a state where no further moves can be made by any of the participants of the protocol. The focus is on the interaction between the different participants with the ultimate aim of being to apply the formalisation to practical applications, as for example the *Netbill* protocol [25, 46].

The next Sections will present a brief overview of *Situation Calculus* as well as describe the representation for the different components of the game definition.

3.6.1 Situation Calculus

Originally the *Situation Calculus* has been presented as a first-order logic formalism to represent change in dynamically changing worlds [67, 72, 85]. Its formalisation allows for actions that have an effect on the current state of the world, as it is perceived by an agent.

The basic assumption of the *Situation Calculus* is that the world may be modeled as a system of interacting *finite automata* or *transition systems* [72]. Using the calculus we formalise a domain problem by providing a model describing:

- how the state of this world will be described, including the initial state;
- any actions that can be performed in the world;
- the preconditions and the effects of an action;

- the state following after a certain action has been decided;
- how properties of the world that are not affected by actions remain the same.

3.6.1.1 Formalisation of Situation Calculus

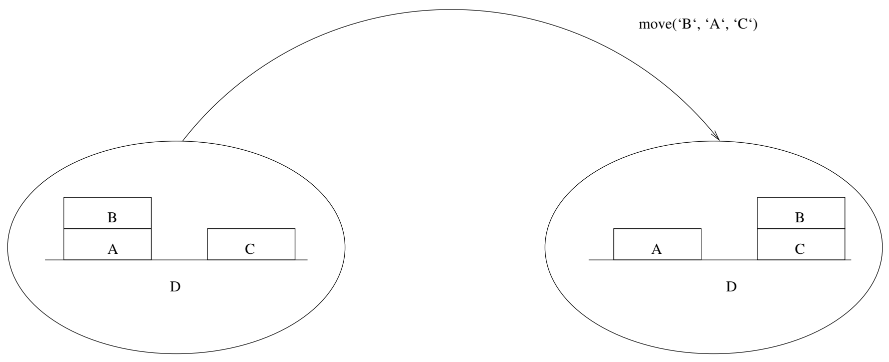


Figure 3.2: Blocks world example.

In Figure 3.2, on the left-hand side we see the initial situation where blocks A and C are on the table D and block B is on block A. On the right-hand side the picture shows the situation where we have moved block B from A on the block C.

Using the *Situation Calculus*, we model the world as a sequence of *situations*. The *initial state* of the world is modeled by an *initial situation* which is denoted by the constant symbol s_0. In each situation we describe the facts that hold true in it. For example, to describe a blocks' world as shown on the left in Figure 3.2 (initially block B is on top of block A and after the move block B is on top of block C), we need to write:

```
holds(on('B', 'A'), s0).
holds(on('A', 'C'), s1).
```

Situations change as a result of *actions*. When an action A takes place in a situation S, it brings about a new situation that is referred to by the function *do(A, S)*. Thus, to derive the new situation, we need to apply an action to it through the *do* function. Note that a situation is not equivalent to a state; a situation is a history of actions recorded by the *do* function, while a state is what holds true at a specific situation. For example, in the blocks' world, the situation where the action `move('B', 'A', 'C')` is performed, is defined as: `do(move('A', 'B', 'C'), s0)`.

In this situation the relations between blocks in the state of the world have changed, as we discuss next.

3.6.1.1.1 Fluents

A fluent is a predicate whose value might change from a situation to another one, as the result of the effects of an action. In our blocks' world example, the *on('B', 'A')* is a fluent that is true in the initial situation and false in the situation that resulted from moving block *'B'* on top of block *'C'*. Typically, a set of fluents can be true in many different situations, for example, there are many different configurations of the blocks' world where block B is on top of block C in different situations.

3.6.1.2 Preconditions of Actions and Effects

It is often the case that actions have preconditions. In *Situation Calculus* these are modeled by a special predicate called *possible(S, A)*: this denotes that in situation S it is possible to perform action A.

```
possible(S, move(From, Obj, To)) ←
   block(Obj),
   holds(clear(Obj), S),
   holds(on(Obj, From), S),
   holds(clear(To), S).
```

Listing 3.1: possible/2 example in Situation Calculus

This way of defining the preconditions of the actions allows us to define generally an action's effects. For example, in order to define what holds after a move block action has taken place, we need to write what is known in the *Situation Calculus* as an effects axiom, as follows:

```
holds(on(Obj, To), do(move(From, Obj, To), S) ←
   possible(S, move(From, Obj, To)).
```

This defines the positive effects of the actions. We can also state what properties an action terminates after it is performed. In this version of situation calculus, we define this by writing what becomes *abnormal* in the new situation. For example, when we move a block from one position to another, we need to also write: `abnormal(on(Obj, From), move(From, Obj, To), S).`

The above assertion states that it is *abnormal* to move the block in a situation and the block to remain where it was in the new situation. The abnormality predicate above must be understood in conjunction with the frame axiom, which we describe next.

3.6.1.3 The Frame axiom

One of the main issues that we need to take into consideration when we describe how fluents change as a result of actions being carried out in situations, is expressing what remains unchanged. For example, we need to state generally that facts such as *on(A, C)* remain there until an action changes them. We do this by describing a general frame axiom as in Listing 3.2.

```
holds(do(A, S), F) ←
    holds(S,F),
    \+ abnormal(F, A, S).
```

Listing 3.2: Definition of The Frame Axiom

This states that a fluent F that holds in a situation S continues to hold after an action A has been executed, as long as it is not abnormal that it holds in the new situation. This frame action, using a *holds* predicate, is much more general than the way successor-state axioms are presented in full-first order logic formulation of the situation calculus [72].

3.6.1.4 Static Facts

To complete the formulation of a domain description using the *Situation Calculus*, we assume the existence of facts that hold across situations, or are not situation dependent. For example the assertions in Listing 3.3 allow us to describe the rigid facts for the blocks world of interest:

```
block('A').
block('B').
block('C').
table('D').
```

Listing 3.3: Static Facts

3.6.2 Evolution of a game

We represent the evolution of a game as the logic program shown in Listing 3.4 [102, 105]:

The game is defined by a recursive predicate through a game *Situation* structure, which we first need to check if it is a terminating state or not. If it is, the game terminates with result equal to *Result*. If it is not terminating and there is a *valid* move that can be made, that *valid* move is selected, its *effects* are applied to produce a *NewSituation*, and,

```
game(Situation, Result):-
    terminating(Situation, Result).
game(Situation, Result):-
    \+ terminating(Situation, _),
    valid(Situation, Move),
    effects(Situation, Move, NewSituation),
    game(NewSituation, Result).
```

Listing 3.4: Game Evolution Representation

then, we continue exploring this *NewSituation* in order to reach the final *Result*.

We, now, need to define how a game situation will be represented, how initial and terminating states will be defined (what will hold at the beginning of the game and what would indicate its termination), how the selection of valid moves by the player will be made and how the effects of these moves are assumed to produce a new situation. Once all these concepts are fully defined, the Authority Agent will be able to explore the interactions between the various players and decide on whether to accept or not an agent's application for joining the society.

3.6.3 Game Situations using the Situation Calculus

A game situation will be represented by terms of the form `sit(Name, Id, Narrative)`. A situation, thus, will consist of three elements:

- *Name* to represent the name (or type) of the protocol (e.g. *auction*);
- *Id* to represent the instance of the protocol that this situation describes. This makes it possible to have many instances of the same protocol running at the same time (e.g. *auc1*).
- *Narrative* to represent the moves made in the game so far - that list of moves will define the state of the game. This is represented as a Prolog list and an empty *Narrative* means that no move has been made in the game.

As an example, a term of the form `sit(order, s0, [])` represents an *order* protocol with an instance name of *s0* and in which protocol no move has been made so far; the narrative component of the description is empty ([]). A situation with a non-empty narrative will be shown shortly.

3.6.4 Game Moves

A move *Move* made by player *Player* is represented by a term of the form `select(PlayerId, Action, Recipients)`; in cases where there is only one role instance, *PlayerId* can be substituted by the role name as there is no ambiguity regarding who made the move.

In the protocols we look at in this thesis, the move is specified only by its name (e.g. *bid*). However, the representation could be enhanced with moves that are parameterised, i.e., the specification of the move will depend not only on the type of the move (expressed by its name), but on a number of other parameters as well. As an example, rather than simply stating that an agent places a bid for an auctioned item we could add the value of the bid and the id of the item, e.g. *select(bidder1,bid(watch1,12),auctioneer)*.

3.6.5 Game Situation

We now need to describe a game situation as a result of the moves made by its players. In this context, we need to be able to reason about properties that describe the game and see how they change when players choose their moves (in an auction protocol, these properties would involve the current highest bid and the bidder who made it). In the Situation Calculus version of the formal framework, we combine the game formulation with our own interpretation of the situation calculus [72] expressed as a normal logic program and describe the state of the game with a series of *holds* predicates, as shown in Listing 3.5

```
holds(sit(Name, Id, []), F):-
    initially(sit(Name, Id, []), F).
holds(sit(Name, Id, [M | Ms]), F):-
    effect(F, M, sit(Name, Id, Ms)).
holds(sit(Name, Id, [M | Ms]), F):-
    holds(sit(Name, Id, Ms), F),
    \+ abnormal(F, M, sit(Name, Id, Ms)).
```

Listing 3.5: holds/2 predicate in Situation Calculus

The first rule in Listing 3.5 states that if we have not started the game yet (the list representing the moves is an empty list), a property F will hold if it holds in the initial situation, represented by *initially/2* assertions. The second rule states that a property F holds, if it is the result (effect) of the last move made in the game. Note that we use a list to keep a history of the moves and the leftmost element refers to the latest move being made. Finally, the third clause states that a property F will hold if it held before the move was made and the move applied did not affect its persistence.

We will be using the mail order protocol in Figure 3.3 to illustrate our formalism. In this protocol we have two roles, *merchant* and *supplier*. Initially, the merchant agent makes an order which can either be confirmed or refused by the supplier (the supplier can also reorder once an order is placed). If the supplier agent refuses the order, the protocol terminates. If it confirms it, the merchant has the option of either withdrawing (protocol termination) or accepting it and then the supplier will have to notify the merchant at which point the protocol terminates.

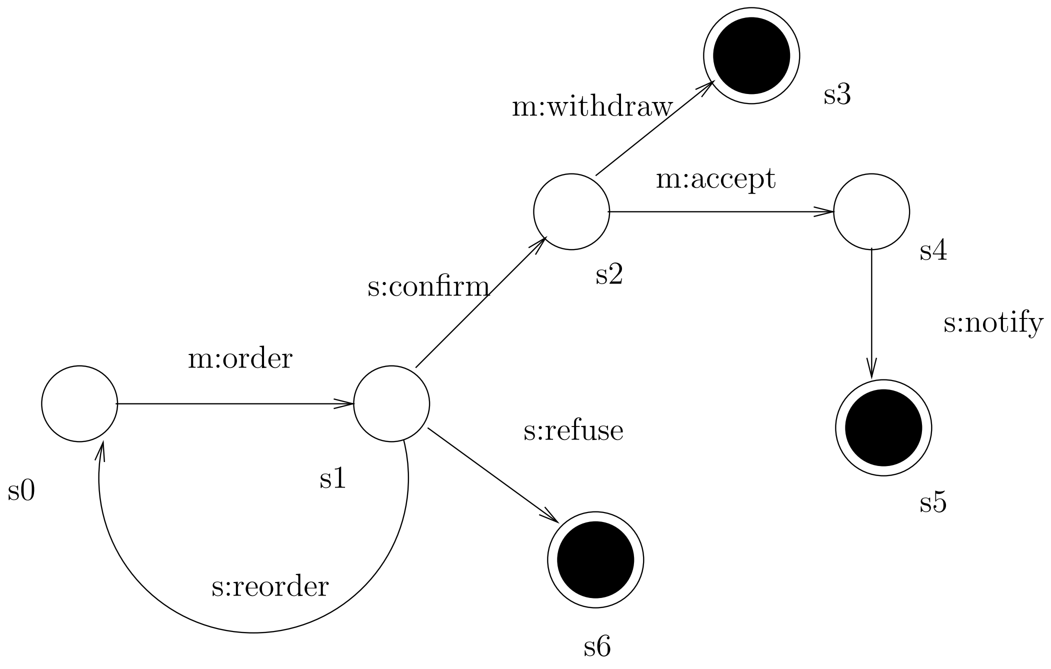


Figure 3.3: A Mail Order protocol

3.6.6 Initial and Terminating States

As mentioned in Section 3.6.2, we need to specify the initial and terminating states of the protocol, i.e., we need to specify what properties hold when the protocol starts and when it ends. The properties holding at the beginning of the game will be recorded by the *initially* predicate, while those describing a terminating state will be specified with *terminating* clauses.

A typical example of properties holding initially are the properties related to the roles of the players. In the mail order protocol involving a manufacturer and a supplier, Listing 3.6 indicates that player *p1* is the manufacturer and player *p2* is the supplier; the *role_of* predicate is used to define that the player specified as the first parameter has the role specified in the second one.

```
initially(sit(mail_order,order,[]),role_of(p1, manufacturer)).
initially(sit(mail_order,order,[]),role_of(p2, supplier)).
```

Listing 3.6: Initial Situation of a mail order protocol

The protocol will terminate when the supplier selects to perform a *notify*, *withdraw* or *refuse* move. In this case, the current state of the game as expressed by the *Situation* argument of the *terminating* predicate becomes a terminating path and is returned by the *game* predicate in the variable *Result* (the running path is a terminating path so it is returned). In other words, the current situation becomes the final result of the game, as it is one of the (possibly) many ways that a sequence of valid moves can be terminated.

In our games framework, we can write it as:

```
terminating(Situation, Situation):-
    Situation = sit(mail_order, s0, N),
    holds(Situation, last_move(select(Player, Act,_))),
    holds(Situation, role_of(Player, manufacturer)),
    member(Act, [refuse, withdraw, notify]).
```

3.6.7 Valid Moves

When describing an interaction protocol as a game, it is important to be able to differentiate between valid and invalid moves - an auctioneer, for example, will need to know which bids are valid so that the item under auction can be adjudicated to the highest bidder. In our framework a move will be *valid* if it meets two conditions - it should be:

- *available* - the protocol must be providing this move as a move that is meaningful to be made within its context (e.g. *accept* in the case of the manufacturer-supplier example);
- matching certain conditions depending on the state the game is in - the fact that the protocol makes a move available does not mean that the user can always select it. For example, the manufacturer agent can select *accept* only if the supplier agent has selected *confirm* as the previous move in the protocol. In this case, *accept* would be a valid move only if there has been a *confirm* move made by the supplier before the merchant selects *accept*.

Listing 3.7 shows how to make the *order* move available to any player of the mail order protocol.

```
available(sit(mail_order, _, _),select(_, order, _)).
```

Listing 3.7: Available predicate in Situation Calculus

If we want to represent that it is valid for a supplier to select the *notify* move, if the previous move was made by a merchant agent and it was an *accept* move, we can write it as in Listing 3.8.

```
valid(sit(mail_order, s0, N), select(P1, notify, P2)):-
    holds(sit(mail_order, s0, N),role_of(P1, supplier)),
    holds(sit(mail_order, s0, N),last_move(select(P2, Act, P1))),
    holds(sit(mail_order, s0, N),role_of(P2, manufacturer)),
    Act = accept.
```

Listing 3.8: Valid Moves of a mail order protocol

3.6.8 Effects of Moves

Every time a valid move is performed in a communication protocol, the effects of the move need to be applied to the current game situation so that they can bring about the next one. In terms of updating the game situation, all we need to do is extend the *Narrative* component of the *situation* predicate to include the last move that the player chose. The *Narrative* component is the component holding the list of moves made by the players so far in the course of the protocol. This is represented by an *effects* predicate, an example of which is shown in Listing 3.9. We are representing the game moves as a list, so we need to prepend the last move made to the head of that list.

```
effects(sit(mail_order,s0,Ms), M, sit(mail_order,s0,[M| Ms])).
```

Listing 3.9: Effects predicate in Situation Calculus

The update of the specific game properties is done implicitly via *effect* and *abnormal* definitions. The *effect* predicate describes what is the effect of a move made in the game on certain properties of interest. For example, the effect of any move is that it changes the last_move property in the next situation. Listing 3.10 demonstrates that the effect of selecting a move *M* is that the value of the *last_move* predicate will have to change to reflect that.

```
effect(last_move(M), M, sit(mail_order, s0, _)).
```

Listing 3.10: Effect predicate for the mail order protocol

We, also, need to provide *abnormality conditions* - i.e. conditions when a property should not hold. The code in Listing 3.11 states that once a new move has been selected, it is abnormal for the *last_move* predicate to have the value of the old move.

```
abnormal(last_move(Mold), Mnew, sit(mail_order,s0,[ Mnew | Mold])).
```

Listing 3.11: Abnormal predicate in the mail order protocol

3.7 Timed games in Event Calculus

A number of protocols have a time element in their description. In the auction protocol presented in Figure 3.4, we are using a compact version with logic propositions binding player actions rather than listing all different combinations to present a high-level view of the protocol. In this context, b:$bid_1 \cup bid_2 \ldots bid_n$ is used to accommodate both the cases where the first bidder bids or the second one or the n^{th} one does. In this protocol, we can have multiple bidders who will be sending messages to the auctioneer in no particular order, and time constraints might be in place for the bidders to place a bid; after this period of time passes, the bid is not accepted.

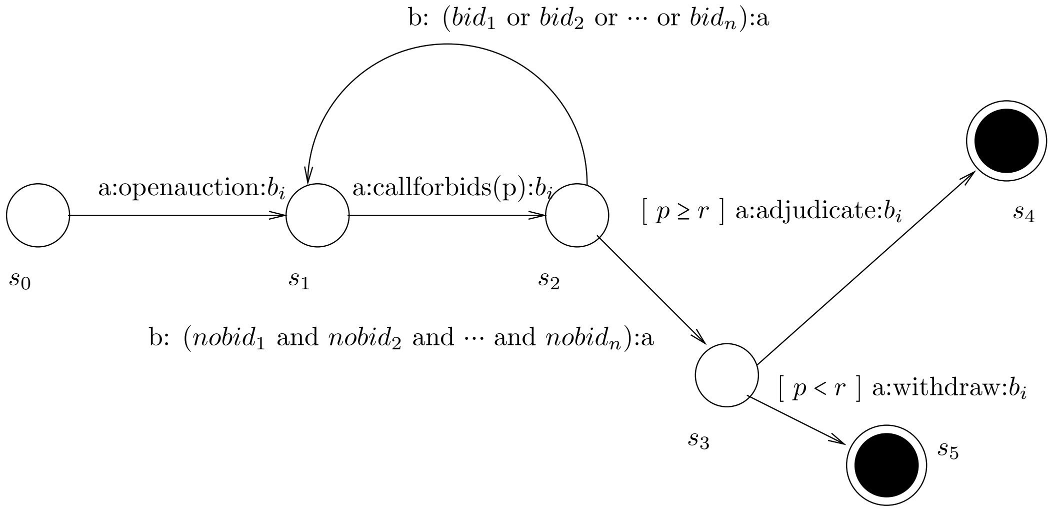

b: (bid_1 or bid_2 or ⋯ or bid_n):a
a:openauction:b_i
a:callforbids(p):b_i
[$p \geq r$] a:adjudicate:b_i
s_4
s_0
s_1
s_2
b: ($nobid_1$ and $nobid_2$ and ⋯ and $nobid_n$):a
[$p < r$] a:withdraw:b_i
s_3
s_5

Figure 3.4: An English Auction

In this protocol (an English auction protocol), the auctioneer opens the auction with an *openauction* call, followed by a call for bids at a start price *p*. If any of the bidders place a bid (by sending a bid_i message to the auctioneer), the auctioneer will issue another call for bids, this time at the price of the new bid. This process will be repeated until none of the bidders expresses an interest in bidding for the item been auctioned. In that case, the auctioneer will have to compare the current price of the item against the reserve price (a price that, if not met, the auctioneer is not obliged to sell the item). If the current price is higher than the reserve price, then the auctioneer adjudicates the item to the highest bidder; otherwise, it is withdrawn.

The following Sections provides a brief overview of the *simple* version of Event Calculus and describe how the components of the game will need to be changed in order to incorporate time constraints.

3.7.1 Event Calculus

In complex computer systems, such as those representing agent communities, protocols and agent actions need to be represented in a strict and theoretical basis. This requirement demonstrates the need for a formalism that will be used to represent the effects of the agent actions over time, as well as any dependencies amongst them (an action might be the starting point for a property or it might be terminating it). We will be using the *simple version* of *Event Calculus* [60, 95–98] to define them.

3.7.1.1 The simple version of ***Event Calculus***

Event Calculus allows quantifications over *fluents*, *actions* or *events* and *time points*. A fluent can be anything that its value can change over time - for example *next(Player)* in the case of a turn-taking game. Actions or events (both terms can be used interchangeably) that happen at different time points influence the truth value of fluents and allow us to reason about them, provided we know what happened up to that point in time. This, of course, makes the assumption that nothing else that influences the truth value of these predicates happened in the time period under consideration, otherwise our reasoning would be based on the wrong value for the fluent. An example of an event is the selection of a move by a player, denoted as *select(P,M,R)*; this might affect the truth value of some of the fluents we are interested in. In our case when a move is selected, the value of the fluents *next(Player)* and *last_move(Move)* are affected.

The predicates of *Event Calculus* allow us to reason about what is true initially, what happens at which time points, what are the effects of an event on the value of the predicates and what predicates hold at what times. The list of the predicates specifying what events initiate/terminate the truth value of what fluents, as well as the fluents that hold true initially, when an event happens, whether it holds true at time point T and when a fluent is clipped - its value changes to false between time points T_1 and T_2, is summarised in Table 3.1.

initiates(E, F, T)	fluent F starts to hold after event E has happened at time T
terminates(E, F, T)	fluent F stops holding after event E at time T
initially(F)	fluent F starts from time 0
happens(E, T)	event E happens at time T
holds_at(F, T)	fluent F holds at time T
$clipped(T_1, F, T_2)$	fluent F is terminated between time points T_1 and T_2

Table 3.1: The predicates of the simple version of *Event Calculus*

3.7.1.2 The axioms of the *Event Calculus*

The three axioms in Listing 3.12 connect the different predicates of *Event Calculus.*

The first one says that a fluent holds at time T, if it holds initially and it has not been stopped until that time point, while the second that a fluent holds at time T_2 if an event A happened at time T_1 - prior to T_2 - that initiated it and it has not been stopped between the time points T_1 and T_2. Finally, the third one states that a fluent is stopped between the time points T_1 and T_2 if there is an action A that happens at time T, somewhere in the time interval that we look at, that terminates the fluent.

```
holdsat(F,T) ← initially(F) ∧¬ clipped(0, F, T)
holdsat(F,T2) ← happens(A,T1) ∧ initiates(A, F, T1) ∧T1 < T2 ∧ \+
    clipped(T1, F, T2)
clipped(T1,F,T2) ←∃ A, T [happens(A, T) ∧T1 < T < T2 ∧ terminates(A,F,T)]
```

Listing 3.12: Event Calculus Axioms

3.7.2 Representing games in the Event Calculus

Time constraints cannot be represented in the simple version of the *Situation Calculus* framework that we have used so far. Extended versions [61, 62, 86, 87, 91, 110] to support reasoning with time in *Situation Calculus* do exist but we prefer to use event calculus for this, as its ontology provides a more natural way to represent time.

To represent timed games, we need to change the representation of situation to include a reference to the time component of the game. The new representation is `sit(Name, Id, Time, Narrative)`.

As before, *Name* is the name (some times we can use the type if there is only one protocol of each type running) of the protocol, *Id* is the id of the game - to facilitate the running of multiple games in parallel, and *Time* is the time point at which we are looking at the game's state. On the other hand, the narrative of what happened so far in the game is now expressed in terms of *episodes*, a set of moves that have happened at the same time point in the game (for example, multiple bidders bidding at the same time point). In their general form, *episodes* are written as `at([select(Player1, Act1,Recipients1), ..., select(PlayerN,ActN,RecipientsN)], T)`. This means that at time point T a number of moves have been selected (Act1 from Player1, Act2 from Player2, ..., ActN from PlayerN); player identifiers can substitute role identifiers if there is one role instance for each role.

In this context, the episode `at([], T)` would mean that nothing happened at time point T (no move specified).

As we will have to reason about a number of properties in the game, we need to have a way of representing what properties hold and when. This is discussed in Section 3.7.3.

3.7.3 Reasoning about properties in timed games

We are using the *simple version* [97] of *Event Calculus* [60], rather than *Situation Calculus* - as the concept of time is inherent in the formulation of Event Calculus, to reason about games with a time component. Listing 3.13 shows the process for deciding whether a property holds or not at a certain time point as well as when it stops holding (i.e., it is *clipped*) between time points T_i and T_n when a event happens in this time period that terminates it.

```
holds(sit(Name, Id, T_n, Narrative_n), F):-
  0 ≤ T_n,
  initially(sit(Name, Id, T_i, Narrative_i), F),
  \+ clipped(F, sit(Name,Id,T_i,Narrative_i),sit(Name, Id, T_n, Narrative_n)).

holds(sit(Name, Id, T_n, Narrative_n), F):-
  happens(E, T_i, Narrative_i, Narrative_n),
  T_i < T_n,
  initiates(E, F, sit(Name, Id, T_i, Narrative_i)),
  \+ clipped(F,sit(Name,Id,T_i,Narrative_i),sit(Name, Id, T_n, Narrative_n)).

clipped(F, sit(Name, Id,T_i,Narrative_i),sit(Name,Id, T_n, Narrative_n)):-
  happens(Estar,T_j,Narrative_j,Narrative_n),
  T_i < T_j,
  T_j < T_n,
  terminates(Estar,F,sit(Name, Id, T_j, Narrative_j)).
```

Listing 3.13: holds/2 predicate in Event Calculus

One important difference between the normal *Event Calculus* formulations and ours is that we do not assert knowledge facts into the knowledge base, but hold them as a list in the game situation predicate. We hold them as a list of episodes in the game situation and they describe a sequence of situations in terms of selections made by the game players previously in the game.

The first clause defines that a property F holds at a time point T_n if the property holds initially and nothing has happened from the time of the initial situation until T_n to change the persistence of the property. The second clause states that a property F holds if an event happened at a time point T_i that initiated F and F has not been clipped in any episode between time points T_i and T_n. A property F is *clipped* if we have some event happening at time point T_j, which lies between T_i and T_n, that terminates F.

We, now, need to define the *happens/4* predicate in line with our representation of events as lists of episodes. This predicate takes four arguments, the first been the event (or the group of events) that we want to check for, the second one is the time we are checking for, while the third and fourth ones are the situations of the game at different (or the same) time points. The general idea is that an event happened if it is part of the list showing the episode moves. As we might be looking for single or multiple events, we have definitions that cover both cases. Listing 3.14 shows the definition for single events, while Listing 3.15 for multiple events.

```
happens(E, T_n, [at(Es_n, T_n) | Situation_n],[at(Es_n, T_n) | Situation_n]):-
    member(E, Es_n).

happens(E, T_i,  [at(Es_i, T_i) | Situation_i],  [at(Es_n, T_n)| Situation_n]):-
    happens(E, T_i, [at(Es_i, T_i) | Situation_i],Situation_n).
```

Listing 3.14: happens/4 definitions in Event Calculus for single events

In the case of single events, if we want to check that an event happened at a specific time point, then the third and fourth elements of the *happens/4* predicate will need to be the same - then, the event happened if it is a member of the list of events that happened at this time point. Alternatively, if we are checking for whether an event happened between two time points, i and n, and we check if the event happened at time i, then we are not interested in the list of episodes at time point n, so we can discard it and call the *happens/4* predicate with the rest of the list as the fourth argument.

```
happens(at(Es_n, T_n), T_n,[at(Es_n,T_n)|Situation_n],[at(Es_n,T_n)|Situation_n]).

happens(at(Es_i, T_i), T_i,[at(Es_i,T_i)|Situation_i],[at(Es_n,T_n)|Situation_n]):-
    happens(at(Es_i,T_i),T_i,[at(Es_i,T_i)|Situation_i],Situation_n).
```

Listing 3.15: happens/4 definitions in Event Calculus for multiple events

In the case of multiple events, the logic is similar. An episode of multiple events for the time point T_n happened by definition at time point T_n (third and fourth arguments of *happens* will be the same). Alternatively, if we are interested in checking whether a sequence of events happened at a time point T_i and we are looking at the time period between T_i and T_n , i.e., the third argument of *happens* would be the list of episodes for time point i and the fourth the list of episodes for time point n we can discard from the fourth argument the list of episodes for time point n).

3.7.4 Effects of actions in timed games

As a result of the changes in the representation of a situation, we need to make changes to the rule describing the effects of an episode. The update consists of adding the episode to the list containing the moves made so far and then increment the clock of the game by one as shown in Listing 3.16. Here, the assumption is made that any new episodes will last for one game time unit, but obviously other time increments can be used.

```
effects(sit(N, Id, T, E_s), at(M_s, T),
        sit(N, Id, NewT, [at(M_s, T) | E_s])):-
    T > 0,
    NewT is T + 1.
```

Listing 3.16: Effects of a move in Event Calculus

3.8 Relationship between games and LTSs

As mentioned in Section 1.1, we want a representation of protocols that is intuitive and easy to understand and used by both the designer and the user of the protocol. The representation of protocols as games meets both requirements as it provides a rich framework with a number of possible extensions. In Section 3.3, we have described a game protocol in terms of an LTS, while Section 2.5 provides a game-based decription of protocols (based on [104]).

The question that arises is how the two descriptions are linked together. As they describe the same set of interactions, there needs to exist a mapping and a transformation process that given the LTS description will output the game-based one and vice-versa. Some of them are straightforward as they are the same in both approaches (e.g. initial and final states). Others need some more thought as the LTS framework is more theoretic whereas the game-based one places the emphasis on the production of executable specifications.

The relationship between our games framework and LTS is shown in Table 3.2.

LTS Representation	**Game Representation**
Initial State	Initial Situation Description
Final State(s)	Those states (situations) described by the `terminating/2` predicate
States	Game Situations (essentially sequences of moves)
Transition Function	`effects/3` predicate
Input Alphabet	Player moves (as described by `select/3` statements).

Table 3.2: Relationship between game and LTS components

The states in the LTS can be represented by the moves that have been made so far (i.e., the *situations* in the LTS representation). The transition function of an LTS describes the effect that a move has when made on a given state; in the LTS representation this is dealt with by the *effects* predicate. Finally, the input alphabet for the LTS consists of the moves the players make at certain states. The LTS representation describes them in terms of *select* statements.

3.9 Formal definition of Competence

In Section 3.3, we defined how social interaction in a society is formulated and understood; we can use the formulation to define what we mean that an agent is *competent* to enter the society.

More specifically a game protocol, $\mathbb{P}^G$, is defined as a tuple $< N, R, S, I, F, V, M, V, E >$. Based on this definition, we will define the competence levels of an agent, g, whose skills are Skills(g), with respect to a role, r, of a protocol, p.

As in section 3.4, the valid moves for the role that the agent is assuming are defined by equation 3.1 and the actions of its role are defined by equation 3.2.

$$V^r = \{v = (s_i, (r_n, \alpha_j, \{r_{k_m}\})) : v \in V \wedge (r = r_n \vee r \in \{r_{k_m}\})\} \tag{3.1}$$

$$A^r = \{\alpha_j : \exists (s_i, (r_n, \alpha_j, \{r_{k_m}\})) \in V^r\} \tag{3.2}$$

(Comp.1:FC). *Fully Competent (FC).* The agent is fully competent when equation 3.3 holds.

$$A^r = \text{Skills}(g) \tag{3.3}$$

(Comp.2:CuA). *Competent under adversity (CuA).* The agent needs to have a strategy, S_g^r, that it can use to avoid reaching a state where it must utter an action that does not belong in its skills or receive an action that it does not understand. This strategy is effectively a subset of V^r – we allow the agent to know the protocol context when deciding which action to perform. The agent must remove from V^r all the valid moves that it is allowed to do, which it can either not perform or if it performs them it may reach a state where it will be unable to execute the protocol. First we need to define the set of valid moves that the role can enact itself, as in equation 3.4.

This is effectively the set that the agent can restrict, since it contains the actions that it can decide to perform or not.

$$V_e^r = \{v = (s_i, (r_n, \alpha_j, \{r_{k_m}\})) : v \in V \wedge r = r_n\} \tag{3.4}$$

To define the required strategy we need first to define the set of trace runs, T of the protocol, as in equation 3.5.

$$T = \{t = (e_0, e_1, \ldots) : e_i = (s_i, r_n, \alpha_j, \{r_{k_m}\}, s_i') \in E \wedge s_{i+1} = s_i'\} \tag{3.5}$$

The projection, T_{Str} of traces to a set of valid moves, Str, is the set of trace runs where traces whose effects' valid moves don't belong to Str have been removed, as in equation 3.6.

$$\begin{aligned} T_{Str} = \{t = (e_0, e_1, \ldots) : e_i = (s_i, (r_n, \alpha_j, \{r_{k_m}\}), s'_i) \in E \wedge s_{i+1} = s'_i \\ \wedge (s_i, (r_n, \alpha_j, \{r_{k_m}\})) \in Str\} \end{aligned} \tag{3.6}$$

The projection of traces effectively introduces a projection of the protocol states into the reachable states, S_{Str}, under that strategy. The strategy of an agent that is competent under adversity is given by equation 3.7.

$$Str^r_g = \begin{array}{l} \{\ v = (s_i, (r_n, \alpha_j, \{r_{k_m}\})) \\ :\ v \in V^r_e \\ \wedge\ \forall\ \ s \in S_{Str^r_g \cup (V \setminus V^r_e)} \\ \quad .((\ r \neq r_n \wedge r \notin \{r_{k_m}\}) \\ \\ \qquad \vee \\ \qquad ((\forall\ (s, (r_n, \alpha_k, \{r_{k_m}\})) \in V \\ \qquad\qquad .\ ((r \neq r_n \wedge r \in \{r_{k_m}\}) \rightarrow \alpha_k \in \text{Skills}(g)) \\ \qquad\qquad \wedge(\ \ (\exists(s, (r, \alpha_k, \{r_{k_m}\})) \in Str^r_g \\ \qquad\qquad\qquad \wedge \alpha_k \in \text{Skills}(g)) \\ \\ \qquad\qquad\quad \vee(\exists\ (s, (r_n, \alpha_k, \{r_{k_m}\})) \in V \\ \qquad\qquad\qquad .r \neq r_n \wedge (r \notin \{r_{k_m}\} \\ \qquad\qquad\qquad\qquad\qquad \vee \alpha_k \in \text{Skills}(g)))) \\ ) \\ ) \\ )\} \end{array} \tag{3.7}$$

In equation 3.7, the enacted valid moves of the role are constrained in such a manner that the set of reachable states has only states with actions that: (i) do not involve the role $(r \neq r_n \wedge r \notin \{r_{k_m}\}))$, or (ii) valid moves where if the agent is the recipient it must be able to understand all of them, and either there is an action that the agent can perform and has not been constrained $(((r \neq r_n \wedge r \in \{r_{k_m}\}) \rightarrow \alpha_k \in \text{Skills}(g)))$, (iii) or there is an action that it can understand or that it is not involved in $((\exists(s, (r, \alpha_k, \{r_{k_m}\})) \in Str^r_g \wedge \alpha_k \in \text{Skills}(g))\exists(s, (r_n, \alpha_k, \{r_{k_m}\}) \in V.r \neq r_n \wedge (r \notin \{r_{k_m}\} \vee \alpha_k \in \text{Skills}(g))))$. So an agent must be able to understand all valid moves that involve it as a recipient in the set of reachable states but may choose to perform no valid move if there is another possible valid move (to avoid deadlocks). The existence of a move that is not constrained by the strategy or of a move that cannot be constrained by the strategy (in the bottom part), is there to ensure that reachable states will have at least one action. While an agent's strategy may be the empty set, $Str_g^r = \varnothing$, which means that the agent never performs any actions itself, only receives actions, we do demand that the set of reachable states to contain some final states, $F \cap S_{Str_g^r \cup (V \setminus V_e^r)} \neq \varnothing$, so as to avoid the case where the protocol cannot be executed at all.

The *maximal* strategy of an agent is the maximal set Str_g^r. It should be noted that in the case of fully competent agents, we have that $\max(S_{Str}) = V_e^r$, since they do not need to constrain the set of reachable states at all.

(Comp.3:CuC). *Competent under cooperation (CuC).* The agent in this case does not have a strategy to constrain the set of reachable states to those where it can perform an action or understand all messages received. Instead, it needs the cooperation of the other protocol participants to form a *partial* strategy, as shown in equation 3.8, that allows the set of reachable states to be non-empty, i.e. $F \cap S_{pStr_g^r \cup (V \setminus V_e^r)} \neq \varnothing$.

$$pStr_g^r = \begin{array}{l} \{v = (s_i, (r_n, \alpha_j, \{r_{k_m}\})) \\ : v \in V_e^r \\ \wedge \forall \;\; s \in S_{pStr_g^r \cup (V \setminus V_e^r)} \\ \quad .((\; r \neq r_n \wedge r \notin \{r_{k_m}\}) \\ \\ \qquad \vee \\ \qquad (\,(\exists \;(s, (r_n, \alpha_k, \{r_{k_m}\})) \in V \\ \qquad\quad .\; ((r \neq r_n \wedge r \in \{r_{k_m}\}) \rightarrow \alpha_k \in \text{Skills}(g)) \\ \qquad\quad \wedge(\;\; (\exists (s, (r, \alpha_k, \{r_{k_m}\})) \in pStr_g^r \\ \qquad\qquad \wedge \alpha_k \in \text{Skills}(g)) \\ \\ \qquad\quad \vee(\exists \;(s, (r_n, \alpha_k, \{r_{k_m}\})) \in V \\ \qquad\qquad .r \neq r_n \wedge (r \notin \{r_{k_m}\} \\ \qquad\qquad\qquad \vee \alpha_k \in \text{Skills}(g)))))))\} \end{array} \tag{3.8}$$

The only difference between equation 3.7 and equation 3.8 is that now the agent need only be able to respond to *some* messages of the other agents, not to *all* of them, that is, the ∀ symbol in the sixth line has been replaced by an ∃ symbol. As long as the other agents cooperate and do not send it messages it cannot understand, the agent can participate in the protocol.

Effectively, an agent in this category may never perform an action itself, just like in the case of agents who are competent under adversity. In addition, in this case an agent may never receive a message from the other agents either, essentially participating in the protocol only in name.

(Comp.4:I). *Incompetent (I).* In this case the agent has no strategy through which it can participate in the protocol and still allow the set of reachable states to contain some final states, that is, we have that $F \cap S_{pStr^r_g \cup (V \setminus V^r_e)} = \varnothing$.

Essentially, an agent is *incompetent* (Comp.4:I) if it can find itself at a state of the protocol where it is *required* to do an action it cannot. The competency levels we have just defined, essentially consider different cases for what a requirement may be.

Agent A	**Agent B**	**Protocol Deadlocks**
(Comp.1:FC)	(Comp.1:FC)	No
(Comp.1:FC)	(Comp.2:CuA)	No
(Comp.1:FC)	(Comp.3:CuC)	Possibly
(Comp.2:CuA)	(Comp.2:CuA)	No
(Comp.2:CuA)	(Comp.3:CuC)	Possibly
(Comp.3:CuC)	(Comp.3:CuC)	Possibly
(Comp.4:I)	$\star$	Possibly

Table 3.3: Games where agents have different competency levels

It is interesting to consider now how an agent of competency type A will behave with an agent of competency type B when participating in the same protocol - will the protocol ever deadlock? Table 3.3 presents the different cases we have. We can see that a protocol may deadlock only if an agent is incompetent or competent under cooperation. The latter is the case because the other agents (even those who are fully competent (Comp.1:FC)) may not wish to cooperate with this agent.

Therefore, the authority agent needs to have a simple test to identify agents that are either incompetent (Comp.4:I) or competent only under cooperation (Comp.3:CuC). Checking whether an agent has (at least) a partial strategy is easily performed by finding a terminating trace run of the protocol where all the actions involving the agent in question belong into its skills [1]. Checking whether an agent is fully com-

[1] A terminating trace run that does not involve the agent at all is an obvious such example.

petent (Comp.1:FC) is even easier, since we only need to verify that equation 3.3 is true.

3.10 Competence and Protocol Completion

The existence of a *maximal* strategy for an agent does not imply that it will be used, even in cases where the agent is fully competent and its *maximal* strategy is the full protocol. This is the case as what the agent *will actually do* when the protocol runs depends on a set of *personal preferences* that we do not have access to and, symmetrically, the fact that the other agents have certain skills does not mean they will use them as that would depend on a set of personal preferences that we did not take into account. As a result, fully competent agents may choose to use conflicting strategies. For example, we can look at the simple *greeting* protocol in Figure 3.5. It involves two agents that do nothing else than greeting each other. If agent A greets first, agent B follows and vice versa.

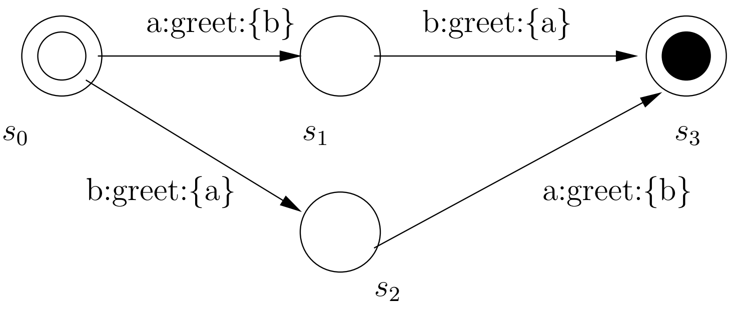


Figure 3.5: A Simple Greeting Protocol

This protocol has two possible traces, namely $s_0 \xrightarrow{a:greet:\{b\}} s_1 \xrightarrow{b:greet:\{a\}} s_3$ and $s_0 \xrightarrow{b:greet:\{a\}} s_2 \xrightarrow{a:greet:\{b\}} s_3$. Assuming that both agents have the skill *greet*, they are both *fully* competent. The maximal strategy of agent A following Equation 3.8 is $V_e^r = \{(s_0, (a : greet : \{b\}), s_1), (s_2, (a : greet : \{b\}), s_3)\}$ and for agent B $V_e^r = \{(s_0, (b : greet : \{a\}), s_2), (s_1, (b : greet : \{a\}), s_3)\}$. For both agents the only move in the set V_e^r are the *greet* actions that they can perform and the only action that they are not the sender but they can receive (as it is part of their skill set) is the *greet* action that the other agent performs that follows their initial greeting. As a result, their maximal strategy are the couple of *greet* moves made by them. One would expect that the two agents will be able to fully utilise the protocol, as they have the skills to do so. However, we have not taken into account their own *personal preferences.* We use the case where each agent has a preference that says that it does not greet the other agent, unless it receives a greeting from it first, i.e., $greet(A) \leftarrow greet(B)$. If that is the case, then none of the agents will do anything, as they will be waiting for one another to act first.

More specifically, if V_e^r is the maximal strategy of the agent and $P_{\mathcal{A}}$ is the set of preferences for the agent (expressed in the form of if-then-else rules or any other notation) that influences the moves it will be making in the protocol, then the part of the protocol that the agent will enact will be $V_e^r \cap P_{\mathcal{A}}$, as the valid moves that the agent can enact will be further constrained by its personal preferences.

For the greeting protocol in Figure 3.5 $P_{\mathcal{A}} = \{s_0 \xrightarrow{b:greet:\{a\}} s_2 \xrightarrow{a:greet:\{b\}} s_3\}$ and $P_{\mathcal{B}} = \{s_0 \xrightarrow{a:greet:\{b\}} s_1 \xrightarrow{b:greet:\{a\}} s_3\}$. Given their maximal strategy V_e^r and their preferences, we can see that the agents will never engage in conversation, as they will both wait for each other to greet first.

3.11 Competence Checking as Planning

Our game-based specification so far allows us to check the evolution of a protocol, by describing all valid game situations that agents can be in, if they follow the social rules imposed by the protocol. As a result, an authority agent can use it to check valid moves when they are made on-line - acting as a referee - or offline - acting as an auditor for a certain sequence of moves representing a particular interaction. However, the specification as it is currently provided does not allow an authority agent to check for competence. For competence checking the authority agent must have the competence profile (essentially the set of actions that it can send/receive) of a specific candidate agent and a representation of the competence profiles of existing agents within the society it is an authority agent of. It can then use the rules of the game, the competence profile of the agent at hand and the other agents, to construct hypothetical situations that reach a terminating situation. The construction of these hypothetical situations amounts essentially to the authority agent planning for these situations using moves that belong to agent profiles and are based on the rules of the protocol. Therefore, we can augment this approach by considering *competence checking* as a special case of planning under the constraint that rules of the game are observed.

Listing 3.17 shows how the agents formulate their plans on the basis of the rules of the game.

```
plan(game(Situation, Result), Situation, Result):-
    achieved(terminating(Situation, Result), Situation, Result).
plan(game(Situation, Result), Situation, Result):-
    \+ terminating(Situation, _),
    assume(valid(Situation, Move), Situation, Move),
    apply(effects(Situation, Move, NewSituation), Situation, Move,
        NewSituation),
    plan(game(NewSituation, Result), NewSituation, Result).
```

Listing 3.17: Competence Checking by Planning

In order to plan for a game, we can stop when a terminating state has been achieved. If that is not the case, a valid move needs to be selected, its effects should be applied on the situation of the game to give us a new game situation and that new situation will be used for running the planning procedure again.

The definition for *achieved/3* and *apply/3* respectively is straightforward as it involves simply calling the *terminating/3* and *effects/3* definitions, as they have been defined for a specific protocol (e.g. see Sections 3.6.6 and 3.6.8 for the mail order protocol). The use of different names is chosen simply to reflect the domain of planning that this variation is used in. Listing 3.18 shows their implementation.

```
achieved(Terminating, Initial, Result):- call(Terminating).

apply(Effects, Situation, Move, NewSituation):- call(Effects).
```

Listing 3.18: Move effects and checking for terminating state

3.11.1 Competency Profiles

When planning for valid moves, however, we need to consider what the players would do given their competency profiles. Such a profile contains the *skills* and *service abilities* of the agents (i.e., the services that the agents offer, the messages that it can understand).

For a communication protocol a competency profile amounts to the communication acts that the agent can utter and understand in the context of that protocol. To describe a competency profile we use rules of the form shown in Listing 3.19.

```
competent(Agent, do(Situation, Act)):- Conditions.
```

Listing 3.19: Generic form of Competency Rules

The Authority Agent of the society will need to keep competence profiles of every agent in the society. This way it should be able to assess whether these agents would be able to fully exploit the protocol or just part of it. As an example, for the two agents in the *mail_order* protocol in Figure 3.3, the competence rules in Listings 3.20 and 3.21 will hold. Player *p1* is able to select moves *order* and *accept* in a *mail_order* situation while player *p2* can select the acts called *reorder*, *confirm* and *notify* - for all rules, there are no conditions limiting the agents from performing them.

```
competent(p1, do(sit(mail_order, _, _), order)).
competent(p1, do(sit(mail_order, _, _), accept)).
```

Listing 3.20: Competence Profile for p1

```
competent(p2, do(sit(mail_order, _, _), reorder)).
competent(p2, do(sit(mail_order, _, _), confirm)).
competent(p2, do(sit(mail_order, _, _), notify)).
```

Listing 3.21: Competence Profile for p2

We assume that competence profiles will always generate ground instances of actions.

3.11.2 Hypothesising Valid Moves

We can now specify the *assume/3* predicate shown in Listing 3.22.

```
assume(Valid, Situation, select(Player1, Act, Player2)):-
    competent(Player1, do(Situation, Act)),
    call(Valid),
    acceptable(Situation, select(Player1, Act, Player2)).
```

Listing 3.22: Definition of the *assume/3* predicate

During the planning process a move is generated by the competence profile of a player and then it is checked for validity. There is, though, one more element that we need to check and possibly place a constraint on: whether making this move will cause unwanted loops (in the mail order protocol of Figure 3.3 such a loop will be caused if the *manufacturer* selects *order* in state s_0 and the *supplier* selects *reorder* in state s_1); this is described in Listing 3.23. This is the case as the program checking for the competency of the agent might be trapped inside the loop and not look at the other moves that the agent could possibly make. Furthermore, these loops might cause undesired effects from a business perspective, i.e., in the previous example the supplier is constantly accumulating goods since he is reordering all the time (as there is one role instance for each role, we can also use the player identifiers as senders and receivers instead of the role identifiers).

```
[select(p1, order,p2), select(p2, reorder,p1)]
```

Listing 3.23: Loop Example in Situation Calculus

In order to deal with such loops, we first identify them in the protocol and then we demand that the chosen move is *acceptable*. The *acceptable* predicate will check the current situation of the game, i.e., the history of moves made so far against the move selected by the *valid* predicate and decide if it forms part of an unwanted loop. The definition of *acceptable* is shown in Listing 3.24.

```
acceptable(sit(_,_,Narrative), Move):-
  \+ cyclic(sit(_,_,[Move|Narrative])).
```

Listing 3.24: Acceptable Predicate

The *cyclic* predicate is, then, defined in Listing 3.25. It starts by incorporating the move selected to the current game situation and then checks whether that situation creates any undesirable loops.

```
cyclic(sit(_,_,Moves)):-
   cyclic_pattern(CyclicPattern),
   check_list( CyclicPattern , Moves).
```

Listing 3.25: Cyclic Predicate

We should note here that *cyclic_pattern/1* is a domain specific assertion that the developer of the protocol has to provide. In the mail order protocol described in Section 3.3, an example of a cyclic pattern would be *[select(p1, order,p2), select(p2, reorder,p1)]* (again, player identifiers correspond to role identifiers). That is, the merchant agent is ordering goods and the supplier agent is reordering them. The predicate *check_list/2* checks whether the list of moves follows the pattern. The definition of *check_list/2* is given in Appendix G.

3.12 Summary

In this chapter we presented the formal framework of our approach. We, also, presented a representation based on games that can be used to represent protocols in both a centralised way and as a distributed collection of player roles. Furthermore, we discussed how this representation can be specified describing both games that have no concurrency requirement using the framework of Situation Calculus as well as games with such requirements with the use of Event Calculus. The two frameworks are used for modelling the state of the game. Finally, we gave a formal definition of the concept of *competence* and commented on how competence influences protocol completion.

The next step would be to produce the algorithms that given a protocol representation of the kind described in this chapter will generate its role components as well as detect any problems (states where the agent players would have to make a choice as to what to do next, but will not have the necessary information). In these cases, we will need algorithms that describe how to go about remedying these situations. Furthermore, we need to ensure that in protocols where multiple instances of one role are present - for example in an auction with multiple bidders, all role instances will have access to all the moves prescribed by the protocol for the bidder role. This is the subject of the next chapter.

Chapter 4

Decomposing the protocol into roles

4.1 Introduction

In this chapter, we present two algorithms for manipulating protocols. The first one decomposes a protocol into separate role-specific descriptions, ensuring that agents are provided with all the information they require for performing their role and only that. The second algorithm attempts to repair protocols that cannot be decomposed into roles. Finally we show how these algorithms can be extended for protocols where a role may have multiple instances participating.

4.2 The Need for Decomposing Protocols

This section looks at the inverse problem of that in Section 3.5, i.e., how to produce the individual role descriptions for a protocol that is given in its full LTS form. In order to achieve this, we need to compute role descriptions like those in Section 3.4, starting from the protocol descriptions of Section 3.3.

This is the case as once the agent is accepted into the society, there is a need for providing it with the protocols it will be using (as determined by the role(s) it applied for). One solution would be to give it the *full* protocol, so that it has complete knowledge of what is happening. This is, however, inappropriate for a number of reasons, namely:

- **security** - there might be sensitive information in the protocol that should not be revealed to all agents. As an example, we can think of a protocol that involves online payment; the merchant should not see the number of the customer's credit card, but another role (gateway) needs to see it and process that information.

- **information overload** - the agent might not need all the information in the protocol as it is of no relevance to its role. If the agent is overloaded with information, or if it is required to process much more information than it needs to, then the process of running the protocol becomes more error-prone and comes at the cost of a high overhead. In addition, if the system in consideration is distributed, then the cost of everyone receiving all messages is even higher.

As a result, the optimal solution would be for every agent to receive only the messages related to the role(s) it plays in the protocol. As our basic representation of a protocol is that of an LTS, essentially we want a smaller LTS that will contain only the *relevant* information for the agent in hand. By relevant, we mean that we need to keep enough information for the agent to be able to enact its role and nothing more, nothing that it will provide it with more information than what is strictly needed. In other words, we want a smaller and equivalent LTS to the original protocol that will contain only the moves that are relevant for a specific role; a minimal protocol with regards to the role's knowledge. States and actions that the role knows about should be kept intact and we should look at whether any actions that do not have the role as a sender or a recipient affect its knowledge on deciding on the next action to perform.

This new protocol will have to be equivalent to the original one in terms of the traces that it can produce if we only take into account the role's knowledge and everything else is considered a *silent* (τ) action. As we are using an LTS description of the protocols, the notion of *bisimulation* can be used to that effect - Section 4.2.1 provides a brief overview of the notion.

We discussed the relationship between *Event Calculus* and *Situation Calculus* with the LTS representation of the protocol in Section 3.8. Both frameworks can be used to model the state of the game. As a result, we can represent the evolution of a game as an LTS by labelling each state with the sequence of moves the player has made in order to reach that state. This, however, would be difficult as - e.g. in the case of loops, there could be an infinite number of possible evolutions that lead to the same state. On the other hand *bisimulation* runs on LTSs, is well-researched, optimised algorithms have been developed, the process is fully automated and well-developed and tested toolsets, e.g. *CADP* and μCRL2, exist.

4.2.1 Bisimulation

Bisimulation [73, 113] is a way of minimising labelled transition systems on abstract (silent) actions (state minimisation technique) while at the same time preserving the properties of the original model. It, also, has the property of been computed automatically without any manual involvement [40].

We can define bisimulation as follows [92]: A binary relation $\mathcal{R}$ on the states of a *Labelled Transition System* is *bisimulation* if whenever $s_1 \, \mathcal{R} \, s_2$:

$$\begin{aligned} &\text{for all } s_1' \text{with } s_1 \overset{\mu}{\rightarrow} s_1' \text{, there is } s_2' \text{ such that } s_2 \overset{\mu}{\rightarrow} s_2' \text{ and } s_1'\mathcal{R}s_2' \\ &\qquad (\forall s_1'. s_1 \overset{\mu}{\rightarrow} s_1' \Rightarrow \exists s_2' \colon s_2 \overset{\mu}{\rightarrow} s_2', s_1'\mathcal{R}s_2'); \\ &\text{the converse, on the transitions emanating from } s_2 \\ &\qquad (\forall s_2'. s_2 \overset{\mu}{\rightarrow} s_2' \Rightarrow \exists s_1' \colon s_1 \overset{\mu}{\rightarrow} s_1', s_2'\mathcal{R}s_1'). \end{aligned} \tag{4.1}$$

Bisimulation can also be defined along the same lines on Mealy automata which are similar to LTSs but have no initial and/or final states [92]. Another definition of bisimulation can be obtained using relations, fixpoints and game theory [74, 84]. In this case the interest is on the concept of *fixed point operators* where bisimulation is defined as the greatest fixed point of a relation involving the actions that can be observed depending on the type of bisimulation that we are looking at.

For the purposes of bisimulation, we consider that the state of a system at any point in time is given by the actions that can be executed at that time point, as well as the actions that can follow that choice. If we, therefore, want to call two systems *bisimilar* it should be true that for every evolution of one of them through a sequence of actions, the same evolution should be possible for the second one using the same actions [75] and vice-versa.

We are especially interested in transition systems where some transitions are not observable by an external observer of the system. In a system describing a business transaction, the merchant has to talk to the bank in order to get the transaction authorised but that is not observable by the customer. As a result, we want to check whether the two systems are equivalent or not on the basis of the actions that we can observe - this is done by defining a family of actions that we can observe on the LTSs and checking if they match the criterion in Equation 4.1.

In producing a new transition system that will have the same behaviour as the original one, it is important to decide on how we are going to deal with silent actions. Depending on the chosen way, we can distinguish between a number of different *bisimulation types*, which we analyse next.

4.2.2 $\tau^\star\alpha$ Bisimulation

In this notion of bisimulation [111], any sequence of silent actions (τ) preceeding an observable action (α) can be replaced by the observable action, i.e., $\tau^\star\alpha \simeq \alpha$.

In the context of LTS, we can distinguish between two types of actions: *visible (observable) actions* and *invisible (silent) actions* that are internal and, as such, not observable. Those actions are substituted with τ and, according to [73], one or multiple instances of them cannot be distinguished by observing the system. As a result of this, any visible action preceded by any number of silent actions will be equivalent to one instance of the visible action. More formally, the $\tau^\star\alpha$ equivalence is denoted as $\approx_{\tau\alpha}$ and is the equivalent of the bisimulation relation (see Section 4.2.1) for the $\{\tau^\star\alpha \mid \alpha \in \mathcal{A}\}$ set of actions.

This is a convenient way of defining equivalence ($\tau^\star\alpha = \alpha$, all silent actions omitted), however it does not preserve always the structure of the transition system we are looking at.

In order to overcome that, *branching bisimulation* was introduced; this is discussed next.

4.2.3 Branching Bisimulation

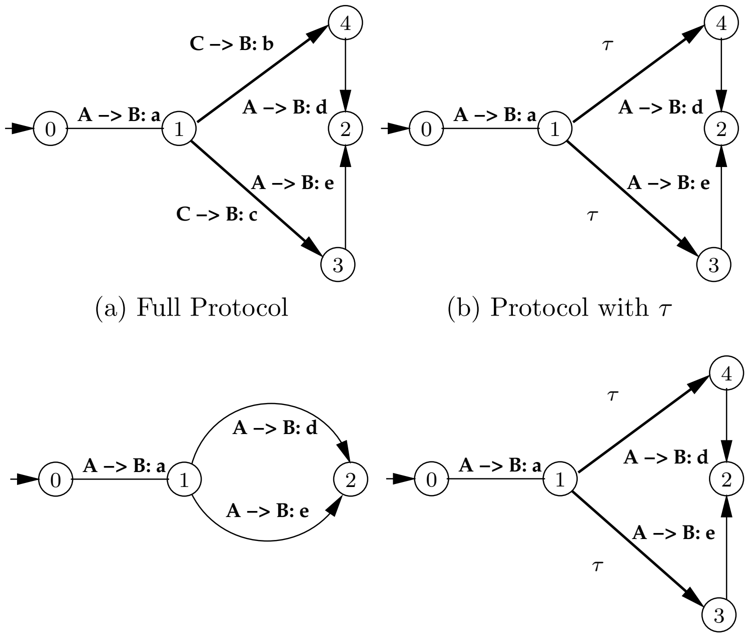
In branching bisimulation we not only worry about the silent actions, but also in preserving the branching structure of the LTS. In other words, although there are τ actions, these cannot be removed always as the structure of the LTS will need to be preserved.

Two LTSs P and Q are *branching bisimilar* if there is a symmetric relation R between their states such that [111]:

- the initial states are related by R;
- If r and s are related by R and $r \xrightarrow{\alpha} r'$, then either $\alpha = \tau$ or there exists a path $s \Rightarrow s_1 \xrightarrow{\alpha} s_2 \Rightarrow s'$ such that r and s_1, r' and s_2 as well as r' and s' are related by R.

4.2.4 From Protocols To Individual Roles - The Algorithm

The next decision to be made is the type of bisimulation to be used. We assume that any action not involving the role as either a sender or one of the recipients is represented as a *silent* τ (tau) action. The simplest form is the $\tau^{\star}\alpha$ one [111], in which every silent action preceding a non-silent action is ignored.



(a) Full Protocol (b) Protocol with τ

(c) $\tau^{\star}\alpha$ Bisimulation (d) Branching Bisimulation

Figure 4.1: Non-implementable protocol due to incomplete knowledge

In Figure 4.1a, we have a protocol with three roles, namely A, B and C. Role A can perform actions a, d and e, all messages going to role B. Role C can perform actions b and c, with both messages again going to role B. If the only knowledge that roles are allowed to have is the actions they have performed and received and we replace non-observable actions for role A with τ, then the resulting role LTS is shown in Figure 4.1b. As we can see there, role A will need to decide what message to send to role B (d or e), but this decision will need to be based on what message C sent to B (b or c) and that is knowledge that role A does not have.

This is true for two reasons. First, role A cannot know whether role C has already acted so that it can follow it with its last action. Second, role A cannot know what action role C has performed in order to follow it with the correct response. Instead, we need a bisimulation relation that will detect this problem and will not remove a silent action if it may cause lack of knowledge to perform further actions in the course of the protocol, e.g. in the case of branching in the previous protocol.

For this, we use *branching bisimulation* [111]. In Figure 4.1c, we see the result of $\tau^{\star}\alpha$ bisimulation been employed. The silent actions are removed and it looks like role A will have to make a choice between sending messages d and e to role B. This choice, however, is not entirely its own; it depends on a previous exchange of messages for which A has no knowledge about - therefore, the silent actions should remain in the protocol. Applying branching bisimulation gives the result shown Figure 4.1d.

At the top level, our approach is as follows (followed for every role in the protocol):

(A) the global protocol is transformed into a role specific one, where any action for which the agent in question is neither the sender nor one of the receivers of the message is replaced with the silent τ (tau) action;

(B) we compute the branching bisimilar LTS;

(C) if no τ transitions, we have a minimal role specific behaviour specification.

Section 4.3 presents different algorithms for repairing protocols. They differ in the number of silent transitions that are repaired as well as on the criteria that we use to decide on which transitions to repair.

Given a game protocol P, the process we need to follow to derive the component roles is described by the *derive_role* procedure shown in Listing 4.1. The protocol we obtain in the first pass might contain silent

(τ) actions, in which case *repair* is needed. We do not commit to any particular *repair* algorithm, as any of the ones in Section 4.3 can be used.

```
// r - role name, g - game protocol
derive_role(r, g) {
  ng = copy(g);
  // relabel irrelevant transitions to tau
  foreach (tr in ng.transitions) {
    if (r ∉ {tr.sender} ∪ tr.receivers)
      tr.label = tau;
  }
  // try to repair the protocol
  do {
    old_ng = copy(ng);
    ng_r = branching_bisimulation(ng);
    foreach (tr in ng_r.transitions) {
      if (tau == tr.label) ∧ is_problematic(tr))
            ng = repair(ng, tr.initialState, r);
            break;
    }
  } while ( ng != old_ng );
  return compute_role_attributes(r, ng_r, g);
}
```

Listing 4.1: Deriving a role from a protocol

The algorithm starts by looking at each transaction in the game protocol and checks to see if the role in question is either the sender or amongst the recipients of a move. If that is not the case, then the role transition label , i.e., move, becomes τ to indicate that the user is not participating in this message exchange. Branching bisimulation is then applied to the resulting protocol and that produces a new game protocol *ng_r*. The algorithm will, then, go through all transitions in the new game protocol and check if the label of the transition is τ or not. If it is, and the silent action is a problematic one (this depends on the algorithm from Section 4.3 that we choose to implement), it will call the *repair* algorithm (see Section 4.3 for the details) for that specific label (one repair might solve more than one problematic transitions). Once the transition is repaired, it moves on to check the next transition. We do not want to commit to any particular interpretation of repair in this case, so we are just stating that repair is needed. The loop will run until the protocol we start the loop with is the same as the protocol after the loop has run, i.e., either there are no silent transitions or if there are they do not cause lack of knowledge for the role. The attributes of the role are then returned.

The algorithm for computing the specification attributes of a role after resolving any lack of knowledge problems in the original protocol is described in Listing 4.2. The resulting LTS will have a higher number of transitions (as it is unlikely that no silent transitions needed repair) and possibly the set of the roles that the agent communicates with will get augmented by new roles as a result of the *repair* process. The number of states would, most likely, be lower as a number of the original LTS states would have been merged into the same equivalence class.

```
// r - role name, rg - role graph, g - game protocol
compute_role_attributes(r, rg, g) {
  roleset=states=actions=moves=valid=effects=∅;
  foreach (tr in rg.transitions) {
    m = tr.move;        a = m.action;
    sndr = m.sender;    rcvrs = m.receivers;
    s = tr.startState; f = tr.finalState;
    roleset = roleset ∪ {sndr} ∪ rcvrs;
    actions = actions ∪ {a};
    moves = moves ∪ {m};
    valid = valid ∪ {(s, m)};
    effects = effects ∪ {(s, m, f)};
  }
  foreach (s in rg.states)
    states = states ∪ {s};
    initial = equivalence_class(g.initialState, rg);
  foreach (s in g.FinalStates)
    finals += equivalence_class(s, rg);
    return ( <r, roleset, states, initial, finals,
            actions, moves, valid, effects> );
}
```

Listing 4.2: Computing the attributes for a role

It starts by setting all role attributes (roleset, states, actions, moves, valid moves and effects) to the empty set (∅). In order to compute the role attributes, there are two pieces of information that we need to look at - one is the role protocol transitions from which we can gather information about the roleset, actions, moves, valid and effects components and the second one is the set of states and final states from which we can gather information about the initial and final states of the role protocol.

For each transition in the role protocol, we can read the move of the transition (in the form *(Sender, Action, Recipients)*) as well as the start state and the final state of it. From the move we can extract the sender, action and recipients of the action. The sender and receivers will need to be added to the roleset of the role, as they represent roles that engage in communication with it. Similarly, the action part of the transition is added to the available actions component of the role, as it is an action that the role can choose to execute. The move as a whole will be added to the set of available moves for the role, while the initial state and the move components as a pair will be added to the valid moves set of the role. Finally, the whole transition will be added to the effects set.

The last two components that need to be specified are the initial state and the final state(s) of the role LTS. As the resulting LTS was produced using bisimulation, the states of the original role LTS are equivalence classes of the states of the protocol LTS so the initial state of the role LTS is the one that contains the initial state of the protocol LTS. Similarly, all states of the role LTS that contain a state designated as final in the protocol specification are designated as final states for the role as well.

4.3 Repairing non-enactable game protocols

If, after running *branching bisimulation* on the role's LTS, there are still silent (τ) moves, then that role's description may not be implementable due to lack of knowledge - the agent may arrive at a state where it will have to make a choice for which it will need to take into account information that it cannot observe.

In that case, if we want to produce a protocol that can be decomposed into roles, we will have to add the missing knowledge to the role in question. The following Sections describe possible approaches for repairing the silent moves.

4.3.1 Updating all silent moves

One approach is to find the equivalent states in the original protocol of the problematic states in the bisimulated one and add the role as a recipient to any messages originating from these states in the protocol. The algorithm is described in the Listing 4.3.

```
// Legend: Game Protocol variables start with GP,
//         Role Protocol variables start with RP
repair(GP, RP_badState, GP_role) {
  GP_class= equivalence_class(RP_badstate, GP);
  // Add role to the receipients of the moves of these states
  foreach (GP_state in GP_class)
    foreach (GP_tran from GP_state.transitions)
      GP_tran.move.receivers = GP_tran_move.receivers ∪ GP_role;
  }
```

Listing 4.3: Updating all silent transitions

This algorithm repairs the protocol by adding the extra information that was missing and was causing the occurrence of the τ move, i.e., adds the role in question to the recipients of the communication act. At the beginning, we calculate all states from the original protocol that are in the equivalence class of the originating state of the transition with the silent move in the bisimulated protocol. Once these are found, for every transition that starts from these states in the original protocol, the set of receivers is updated with the inclusion of the role whose LTS we are calculating. This, of course, is a rudimentary approach that results in large protocols as a number of transitions that need no updating do get updated.

4.3.2 Updating frontier silent moves

Another approach would be to repair a few transitions of the original protocol, those that start from any state in the original protocol that belongs to the same equivalence class as the original state of the silent action in the bisimulated protocol and finish in any of the states belonging to the same equivalence class as the end state of the same transition.

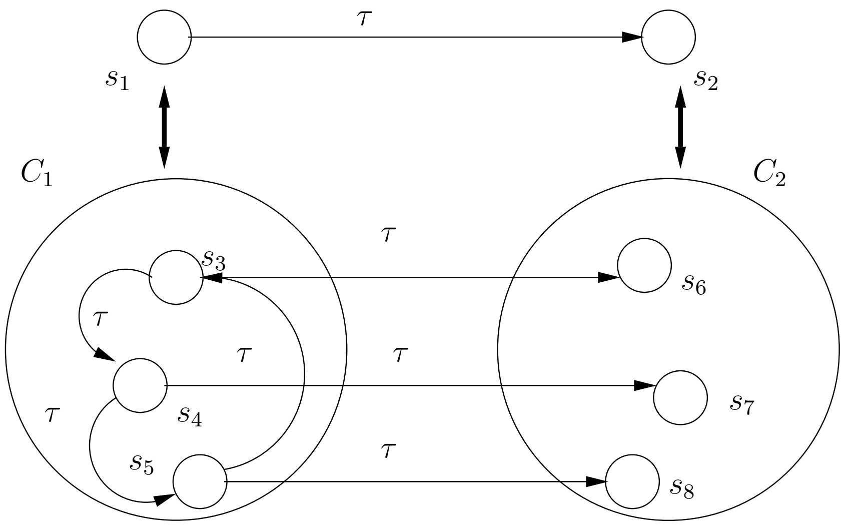


Figure 4.2: Branching Bisimulation Equivalence Classes

In Figure 4.2 after running branching bisimulation we have states s_1 and s_2 linked with a τ transition. However, as branching bisimulation is an equivalence relation placing states into equivalence classes, each of these two states would belong to an equivalence class of states from the original LTS. In this case, we have two equivalence classes $C_1 = \{s_3, s_4, s_5\}$ (represented by s_1) and $C_2 = \{s_6, s_7, s_8\}$ (represented by s_2). By looking at the transitions, we can see that the transitions from states belonging to class C_1 to states belonging to class C_2 are all τ transitions that need to be repaired. The benefit, however, in comparison with the approach described in Section 4.3.1 is that we do not repair any silent transitions *internal* to the class , i.e., the transitions from s_3 to s_4, s_4 to s_5 and s_5 to s_3.

The algorithm that performs the repair is described in Listing 4.4:

```
//Legend: Game Protocol variables start with GP,
//        Role Protocol variables start with RP
repair(GP, RP_Transition, GP_role) {
  RP_initial_state = RP_Transition.initial_state;
  RP_end_state = RP_Transition.final_state;
  GP_equiv_initial_states = equivalence_class(initial_state,
GP);
 GP_equiv_end_states = equivalence_class(RP_end_state,GP);
 //Add role in the recipients of the moves
 //of those transitions that start in
 //GP_equiv_initial_states, end in GP_equiv_end_states
 //and is a silent transition in the original protocol
 foreach (GP_tran from GP_state.transitions) {
  GP_initial_state = GP_tran.initial_state;
  GP_final_state = GP_tran.final_state;
  GP_m = GP_tran.move;
  GP_recipients = GP_tran.recipients;
  if (GP_initial_state ∈ GP_equiv_initial_states ∧
   GP_final_state ∈ GP_equiv_end_states ∧
    GP_role ∉ GP_recipients)
   GP_tran.move.recipients = GP_tran.move.recipients ∪
                                    GP_role;
  }
}
```

Listing 4.4: Updating silent actions by looking at equivalence groups

4.3.3 Updating selected silent moves

Our approaches to protocol repair so far, have considered silent actions as something that needs to be removed from the role's final LTS- their presence would imply lack of knowledge and failure in implementation.

However, this is not always true. A silent action in a role's protocol needs to be repaired only if it is causing problems in the role's action selection process. Assuming a branch where the first move in both leaves is τ, the following combinations exist for the follow-ups:

- the two actions following the silent ones are both receive actions for the role - in that case, we do not need to repair the transition as the role has no decision to make and just waits to receive a message;
- the two actions following the silent ones are both send actions for the role and they are different in terms of either the move or the recipients of the move (or both); in this case repair is needed so that the role will have the required information to decide on which move to pursue;

- one of the following moves is a send, while the second one is a receive; we need to repair the protocol in this case too, as the role in question will need the extra information to decide whether it will wait to receive the prescribed message or go ahead and send a message.

If such moves are found in a role's LTS, then they need to be repaired. This presents the overhead of having to examine a much larger section of the protocol every time we come across a silent move, but gives smaller final protocol sizes.

The algorithm for repairing a protocol in this way is shown in Listing 4.5 (this time we have to include the role LTS as well, as it is been used to check for the moves that need to be repaired),

```
//Legend: Game Protocol variables start with GP,
//        Role Protocol variables start with RP
repair(GP, RP_Transition, GP_role, RP) {
  RP_initial_state = RP_Transition.initial_state;
  RP_end_state = RP_Transition.final_state;
  // check if the transition needs to be repaired
  RP_outgoing_transitions = find_outgoing(RP_initial_state);
  forall ( t ∈ RP_outgoing_transitions,k ∈ RP_outgoing_transitions, $k \neq t$)
      {
        if ( t.Move == "tau" ∧ k.Move == "tau"){
            final_state_t = t.FinalState;
            final_state_k = k.FinalState;
            outgoing_transitions_newt = find_outgoing(final_state_t);
            outgoing_transitions_newk = find_outgoing(final_state_k);
            forall (r ∈ outgoing_transitions_newt ∧ s ∈
                outgoing_transitions_newk){
               $Move_1$ = r.Move;      $Move_2$ = s.Move;
               $sender_1$ = r.Sender; $sender_2$ = s.Sender;
               $Recipients_1$ = r.Recipients; $Recipients_2$ = s.Recipients;
                if (($sender_1$ == $sender_2$ == GP_Role) ∧ (($Move_1 \neq Move_2$) ∨ (
                     $Recipient_1 \neq Recipient_2$))  ∨
                ($Sender_1$ == GP_Role ∧$Sender_2 \neq$ GP_Role∧GP_Role ∈ $Recipient_2$)){
                   // repair process
                  initial_equiv =equivalence_class(RP_initial_state,GP);
                  end_equiv = equivalence_class(RP_end_state,GP);
                  forall (v ∈ GP.Transitions) {
                    initial_state = v.InitialState;
                    final_state = v.FinalState;
                    if (initial_state ∈ initialequiv ∧
                        final_state ∈ endequiv)
                          v.Recipients = v.Recipients ∪ GP_Role;
                   }
                }
        }
    }
}
```

Listing 4.5: Updating selected silent transitions for role R

4.4 Protocols with multiple instances of the same role

In the protocols we have considered so far, there was only one instance of each role. However, in some cases more than one instance of some roles are required for the protocol to make sense or to be complete. In an auction protocol if there is only one participant assuming the role of bidder, some messages will not be uttered as they are only applicable when there are multiple bidders. In this case, we need to produce a version of the protocol with more than one participants playing the role of bidder to ensure the exchange of all possible messages.

Furthermore, as all LTSs describe the behaviour of the same role, we would expect them to be equivalent. The idea, thus, is that if we decompose the protocol instance and compare the LTS generated for each of the multiple instances of the same role, these LTSs will have to describe the "same" behaviour, i.e., be **bisimilar**.

4.4.1 Verification

Once the protocol is in the format specified in Section 3.3 on page 31, we can apply bisimulation to decide if the LTSs produced by bisimulation are the "same" (really, bisimilar as state number(s) might have been changed by the bisimulation tool) or not. Also, we need to use *branching bisimulation* as the auction protocol does have a number of branches and we need to maintain that structure. If we use $\tau^\star\alpha$ instead, branches in the protocol would be eliminated as any sequence of silent actions would be condensed with the next known action to the role. If the LTSs for the two bidder role instances prove to be the "same", then the auction protocol prescribes the same moves for both of them. If not, it might be the fault of the protocol or it could be that the auction house differentiates between different bidders (e.g. on the basis of their participation order) and this is reflected in the protocol.

The algorithm for checking whether LTSs from multiple role instances exhibit the same behaviour or not is described in Listing 4.6.

The first step would be to identify the number of role instances participating in the protocol; in the case of the auction protocol this number is equal to two. Before running branching bisimulation, we need to create the role specific LTSs for the individual role instances where any action

```
check_role_automata(GP,R) {
// GP is the Game Protocol
// R is the role in GP with multiple instances
n = number_of_role_instances(R);
for (i = 1 ... n)
    {
        $automaton_i$ = create_role_automaton(GP, $R_i$);
    }
run_branching_bisimulation($automaton_1$,...,$automaton_n$);
if (branching_bisimilar($automaton_1$,...,$automaton_n$))
    {
        return true;
    }
else
    {
        return false;
    }
}
```

Listing 4.6: Verifying multiple LTSs of the same role instance

not having the role instance as a sender or one of the recipients will be renamed to a silent (τ) action.

This will provide us with a role LTS for each role instance. Once we create all LTSs, we check to see if they are *branching bisimilar* to confirm that they are exhibiting the same behaviour. If they are *check_role_automata* returns true. Otherwise, it returns false and we need to check whether the business rules justify this difference in behaviour or whether the LTSs need to be repaired using one of the approaches in Section 4.3 on page 69.

4.5 Summary

In this chapter, we have discussed how to decompose a protocol into role LTSs and how to check, using the concept of *bisimulation*, if these role LTSs are implementable or the protocol needs to be repaired. We have presented three different repair techniques, each time decreasing the amount of extra information that we provide to a role in order for an agent to be able to implement it. Finally, we have presented an approach that deals with protocols where there are multiple instances of one (or more) role(s) during its execution. This is used to check if all LTSs for the same role instance describe the same behaviour, i.e., are *bisimilar*.

The next chapter provides examples of this approach in both frameworks (Event and Situation Calculi) described in Chapter 3. We, also, provide an example of an auction with two bidders and how the bidder LTSs for the two role instances can be checked for bisimilarity.

Chapter 5

Case Studies

5.1 Introduction

In this chapter we demonstrate our approach by the use of examples. We show how the competence of an agent can be assessed using both the Event and Situation Calculi approaches, both for acyclic and cyclic protocols. We, then, show how a protocol can be decomposed into its constituent roles and how the decomposition of a protocol with multiple role instances can be checked for correctness with the use of bisimulation.

5.2 Setting the Scene - an acyclic protocol

5.2.1 The protocol

We will be using the mail_order protocol that was partially described in Section 3.6.6 to demonstrate an acyclic protocol. In this protocol, a merchant is interacting with a supplier for the purchase of goods. The merchant makes an order, that the supplier can refuse, in which case the protocol terminates. On the other hand, if the supplier confirms the order and supplies an invoice, then the merchant can withdraw in which case the protocol terminates again. If the merchant accepts the offer, then the supplier agent notifies the merchant agent of the order details.

The protocol, in the form of an automaton, is shown in Figure 5.1.

In Sections 5.2.2 and 5.2.3, we describe how to formalise this game in terms of initial and terminating situations, valid moves, effects of moves and how to write the *game* predicate that will be checking the progress of

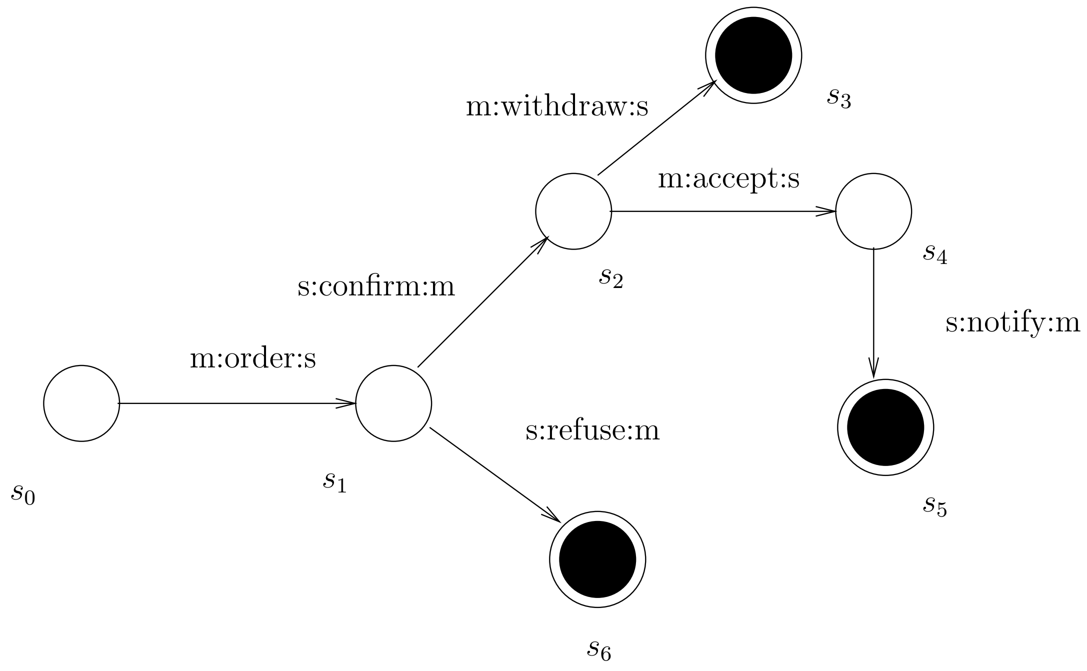


Figure 5.1: A mail_order protocol

the game and decide whether it reached a terminating point or not. The formalisation will be in both frameworks (Event and Situation Calculi).

5.2.2 Protocol Rules in Situation Calculus

5.2.2.1 Game Situations

As described in Section 3.6, a game situation is represented by the predicate *sit(Name, Id, Moves).* In the example, we are describing a mail_order protocol so the situation narratives will be of the form:

```
sit(mail_order, buy1, Moves).
```

Listing 5.1: Situation Narratives

5.2.2.2 Initial and terminating situations

The initial situation describes what holds initially in the game, i.e., information about the roles that participants play as well as any additional information regarding facts that are true when the game starts.

In the mail_order protocol example, the initial situation will describe the roles of the players as in Listing 5.2. We assume that the name of the merchant is *john* and the name of the supplier is *paul* and use the predicate *role_of(X, Y)* to assign roles to agents (agent X has role Y). In an initial situation, no moves have been made so the *Moves* component is an empty list.

```
initially(sit(mail_order,buy1,[]),role_of(john, merchant)).
initially(sit(mail_order,buy1,[]),role_of(paul, supplier)).
```

Listing 5.2: What holds initially in the protocol (Situation Calculus)

For the mail_order protocol to terminate, there are three options:

- the supplier refuses to provide the items requested;
- the supplier notifies the merchant that he accepts the order;
- the merchant withdraws his order.

Listing 5.3 specifies these terminating conditions using the predicate *last_move(X)* that indicates whether the last move selected by a participant in the protocol is *X*. The first rule covers the cases where the supplier terminates the protocol with a *refuse* or a *notify* move and corresponds to the first two conditions, while the second rule covers the case that the merchant terminates the protocol with a *withdraw* message.

```
terminating(Situation, Situation):-
  holds(Situation,last_move(select(P1,Act,P2))),
  member(Act, [refuse,notify]),
  holds(Situation,role_of(P1,supplier)),
  holds(Situation,role_of(P2,merchant)).

terminating(Situation, Situation):-
  holds(Situation,last_move(select(P1,Act,P2))),
  holds(Situation, role_of(P1, merchant)),
  holds(Situation,role_of(P2,supplier),
  Act = withdraw.
```

Listing 5.3: Terminating conditions for the protocol in SC

5.2.2.3 Valid Moves

Valid moves are the moves that can be selected by a player in a certain state of the game. The conditions on the state will be expressed as a sequence of *holds/2* predicates and valid moves in every state will be a subset of *available* moves, i.e., the moves that the game protocol makes available to its players.

In the mail_order protocol of Figure 5.1, the available moves are shown in Listing 5.4. Any player can select any move, as there are no constraints upon their selection. The players can select them at any time with the only constraint being that they have to be participating in a *mail_order* protocol.

```
available(sit(mail_order,buy1,Moves),select(_,order,_)).
available(sit(mail_order,buy1,Moves),select(_,refuse,_)).
available(sit(mail_order,buy1,Moves),select(_,confirm,_)).
available(sit(mail_order,buy1,Moves),select(_,withdraw,_)).
available(sit(mail_order,buy1,Moves),select(_,accept,_)).
available(sit(mail_order,buy1,Moves),select(_,notify,_)).
```

Listing 5.4: Available Moves in Situation Calculus

The second filter for the validity of a move is to decide whether it is legal or not, i.e., determine if it can be selected in a specific situation. The example we are looking at is a *shallow* protocol [35], in that the next move only depends on the type of the previous move and not on the contents of the message (e.g. the supplier can select the confirm move irrespective of what the customer ordered). As a result, the criteria we will be using to determine validity are the roles of the player (by using the *role_of/2* predicate) and the last move that has been selected by the other participant(s) (by using the *last_move/2* predicate).

The rules applicable to this protocol are as in Listing 5.5. For example, the first rule specifies that it is legal for a player to select the *order* move if its role is that of merchant and no last move has been selected previously in the game.

```
valid(sit(mail_order,buy1,N), select(P1, order,P2)):-
    holds(sit(mail_order,buy1,N),role_of(P1,merchant)),
    holds(sit(mail_order,buy1,N),role_of(P2,supplier)),
    \+ holds(sit(mail_order,buy1,N),last_move(_)).

valid(sit(mail_order,buy1,N), select(P1,confirm,P2)):-
    holds(sit(mail_order,buy1,N),role_of(P1,supplier)),
    holds(sit(mail_order,buy1,N),role_of(P2,merchant)),
    holds(sit(mail_order,buy1,N),last_move(select(P2,order,P1))).

valid(sit(mail_order,buy1,N), select(P1, refuse,P2)):-
    holds(sit(mail_order,buy1,N),role_of(P1,supplier)),
    holds(sit(mail_order,buy1,N),role_of(P2,merchant)),
    holds(sit(mail_order,buy1,N),last_move(select(P2,order,P1))).

valid(sit(mail_order,buy1,N), select(P1, withdraw,P2)):-
    holds(sit(mail_order,buy1,N),role_of(P1,merchant)),
    holds(sit(mail_order,buy1,N),role_of(P2,supplier)),
    holds(sit(mail_order,buy1,N),last_move(select(P2,confirm,P1))).

valid(sit(mail_order,buy1,N), select(P1, accept,P2)):-
    holds(sit(mail_order,buy1,N),role_of(P1,merchant)),
    holds(sit(mail_order,buy1,N),role_of(P2,supplier)),
    holds(sit(mail_order,buy1,N),last_move(select(P2,confirm,P1))).

valid(sit(mail_order,buy1,N), select(P1, notify,P2)):-
    holds(sit(mail_order,buy1,N),role_of(P1,supplier)),
    holds(sit(mail_order,buy1,N),role_of(P2,merchant)),
    holds(sit(mail_order,buy1,N),last_move(select(P2,accept,P1))).
```

Listing 5.5: Valid Moves in Situation Calculus

5.2.2.4 Effects of Moves

As the players select moves, the state of the game - as saved in the *situation/3* predicate changes. These changes are captured by *effects*, *effect* and *abnormal* rules. The *effect* rules can be summarised as follows:

- when a player makes a move, this is prepended to the last argument of the *situation* predicate; thus, the list of moves grows by one element, the last move (*effects*);
- when a player makes a move in the protocol, this automatically becomes the last move (*effect*).

The first item can be captured by the rule in Listing 5.6; if *john* makes an *order* move with *paul* as the recipient in the initial situation, then this move is added to the list of moves (in fact, it is the first move made in the protocol). In Listing 5.7 when *john* makes the same move in the initial situation, the effect of that move for the *last_move* predicate is that its value will have to change to reflect that this is the last move made in the game.

```
effects(sit(mail_order, buy1, []), select(john,order,paul),
        sit(mail_order, buy1, [select(john,order,paul)])).
```

Listing 5.6: Move effects in situations

```
effect(last_move(select(john,order,paul)),select(john,order,paul),
       sit(mail_order,buy1,[])).
```

Listing 5.7: Local Move Effects

Finally, we need to define *abnormality* conditions, i.e., conditions where a property holds when it should not hold. In our case, it would be abnormal for the *last_move* property to have the value of the old move, when a new one is selected. This is shown in Listing 5.8.

```
abnormal(last_move(select(john,order,paul)), select(paul,confirm,john),
    sit(mail_order, buy1, [select(john,order,paul)])):-
            \+ OldMove = NewMove.
```

Listing 5.8: Abnormal situations in Situation Calculus

5.2.2.5 An example run

By executing the rules specified in the previous sections, we obtain the possible outcomes shown in Listing 5.9.

```
R = sit(mail_order, buy1,
[select(john, withdraw, paul),select(paul,confirm, john),
 select(john, order, paul)]) ;
R = sit(mail_order, buy1,
[select(paul, notify, john),select(john, accept,paul),
 select(paul, confirm, john),select(john, order, paul)]) ;
R = sit(mail_order, buy1,
[select(paul, refuse, john),select(john, order,paul)]) ;
false.
```

Listing 5.9: Game Results in Situation Calculus

This list represents all the runs that an agent can realise on the basis of the competencies it supplied the Authority Agent with at the application process. These will have to be compared against the competence requirements of the society to decide the agent will be allowed entry or not. Essentially we assemble all the *runs* that the agent can realise, e.g. by using the `findall/3` predicate and then compare the result with the *runs* that the AA of the society is requiring the agent to realise.

5.2.3 Protocol Rules in Event Calculus

The alternative formulation in which we can specify the protocol rules is that of Event Calculus, as discussed in Section 3.7.1. As before, we will be describing the different components of the protocol viewed as a game. In this example, we could have selected either formulation but in the case that we had to take time into account when developing the game rules, an *Event Calculus* formulation would be more appropriate as it is a more natural way of representing time.

5.2.3.1 Game Situations

The description of a situation in Event Calculus is very similar to the one in Situation Calculus, but with an added time component. The time component is needed as in some protocols (e.g. an auction protocol), moves have to be selected within a pre-defined period of time or time is an important component of the game (e.g. the auction runs for a pre-defined amount of time). The general form of a situation in Event Calculus is `sit(Name, Id, Time, Narrative)`. In the mail_order protocol example, at time point 2, the predicate describing the situation could be:

```
sit(mail_order, buy1, 2, [at([select(paul,confirm,john)],1),at([select(
    john, order,paul)],0)]).
```

This situation predicate describes a mail_order protocol with an Id of *buy_protocol* that is at time point 2. The moves that have been made so far are an *order* move from the merchant agent (*john*) and a *confirm* move by the supplier (*paul*).

5.2.3.2 Initial and Terminating Conditions

We need to define what holds true at the beginning of the protocol as well as what would terminate it. For the mail_order protocol, what holds initially is information regarding the roles that the protocol participants play. These would be expressed by the statements in Listing 5.10.

```
initially(sit(mail_order,buy1,0,[]),role_of(john,merchant)).
initially(sit(mail_order,buy1,0,[]),role_of(paul,supplier)).
```

Listing 5.10: Initial episodes in Event Calculus

The protocol can terminate by two actions of the supplier (*notify* and *refuse*) and one of the merchant (*withdraw*). In *Event Calculus* this is described as in Listing 5.11.

```
terminating(sit(mail_order,buy1,T,N),sit(mail_order,buy1,T,N)):-
        holds(sit(mail_order,buy1,T,N),last_moves([select(P1,X,P2)])),
        holds(sit(mail_order,buy1,T,N), role_of(P1,supplier)),
        holds(sit(mail_order,buy1,T,N), role_of(P2,merchant)),
        member(X, [notify,refuse]).

terminating(sit(mail_order,buy1,T,N),sit(mail_order,buy1,T,N)):-
        holds(sit(mail_order,buy1,T,N),last_moves([select(P1,X,P2)])),
        holds(sit(mail_order,buy1,T,N), role_of(P1,merchant)),
        holds(sit(mail_order,buy1,T,N), role_of(P2,supplier)),
        member(X, [withdraw]).
```

Listing 5.11: Terminating conditions in Event Calculus

5.2.3.3 Valid Moves

As in Section 5.2.2.3, we describe what the agent can do in the current state of the game in the form of *valid moves*. They are expressed as mostly a sequence of *holds* constraints and, at every state, they form a subset of the moves that the protocol makes available to the players.

Listing 5.12 describes the moves that the mail_order protocol makes available to its players. These moves can be selected by any player at any time over the course of the protocol.

For a move to be selected, it needs not only to be available but the player should be able to select it on the basis of the current state of the game

```
available(sit(mail_order,buy1,_,_),select(_,order,_)).
available(sit(mail_order,buy1,_,_),select(_,confirm,_)).
available(sit(mail_order,buy1,_,_),select(_,refuse,_)).
available(sit(mail_order,buy1,_,_),select(_,withdraw,_)).
available(sit(mail_order,buy1,_,_),select(_,accept,_)).
available(sit(mail_order,buy1,_,_),select(_,notify,_)).
```

Listing 5.12: Available moves in Event Calculus

and its role, e.g. the supplier agent can select *refuse* only if the merchant agent has selected *order* immediately before. In contrast with *Situation Calculus* , we are now using lists to store the moves players make as a number of them might choose to move at the same time point. For example, an auctioneer makes a call for bids and all bidders will have to reply by the next time point; their collective moves will be stored in a list. For the mail_order protocol we are considering, the rules are specified in Listing 5.13.

```
valid(sit(mail_order,buy1,T,N), select(P1, order,P2)):-
    holds(sit(mail_order,buy1,T,N),role_of(P1,merchant)),
    holds(sit(mail_order,buy1,T,N), role_of(P2,supplier)),
    \+ holds(sit(mail_order,buy1,T,N),last_moves(_)).

valid(sit(mail_order,buy1,T,N), select(P1,confirm,P2)):-
    holds(sit(mail_order,buy1,T,N),role_of(P1,supplier)),
    holds(sit(mail_order,buy1,T,N), role_of(P2,merchant)),
    holds(sit(mail_order,buy1,T,N),last_moves([select(P2,order,P1)])).

valid(sit(mail_order,buy1,T,N), select(P1, refuse,P2)):-
    holds(sit(mail_order,buy1,T,N),role_of(P1,supplier)),
    holds(sit(mail_order,buy1,T,N),role_of(P2,merchant)),
    holds(sit(mail_order,buy1,T,N),last_moves([select(P2,order,P1)])).

valid(sit(mail_order,buy1,T,N), select(P1, withdraw,P2)):-
    holds(sit(mail_order,buy1,T,N),role_of(P1,merchant)),
    holds(sit(mail_order,buy1,T,N),role_of(P2,supplier)),
    holds(sit(mail_order,buy1,T,N),last_moves([select(P2,confirm,P1)])).

valid(sit(mail_order,buy1,T,N), select(P1, accept,P2)):-
    holds(sit(mail_order,buy1,T,N),role_of(P1,merchant)),
    holds(sit(mail_order,buy1,T,N),role_of(P2,supplier)),
    holds(sit(mail_order,buy1,T,N),last_moves([select(P2,confirm,P1)])).

valid(sit(mail_order,buy1,T,N), select(P1, notify,P2)):-
    holds(sit(mail_order,buy1,T,N),role_of(P1,supplier)),
    holds(sit(mail_order,buy1,T,N),role_of(P2,merchant)),
    holds(sit(mail_order,buy1,T,N),last_moves([select(P2,accept,P1)])).
```

Listing 5.13: Valid Moves in the Event Calculus Representation

5.2.3.4 Effects of the Moves

When a player selects to make a move, the state of the game will need to change and this will need to be recorded. We are using the *effects/4* predicate (Listing 5.14) to define how:

- the time of the game increases by one after each move;
- the list of moves that have already been selected in the game is updated;
- the value of the *last_moves* predicate (as well as any other predicates recording other parts of the game state - e.g. the running price of the auction, if required) is updated.

The code is as follows - the second line indicates that the supplier chose to perform move *confirm* at time point one after the merchant agent has selected to make an *order* move:

```
effects(sit(mail_order, buy1, 0, []),
        at([select(john,order,paul)], 1),
        sit(mail_order,buy1,1,[[select(john,order,paul)] |[]])).
```

Listing 5.14: Effects of a move (or moves) in a game episode

5.2.3.5 An example run

By executing the rules specified in the previous section, we obtain the outcomes in Listing 5.15.

```
R = sit(mail_order, buy1, 2,
[at([select(paul,refuse,john)],1),at([select(john,order,paul)],0)]);
R = sit(mail_order, buy1, 3,
[at([select(john,withdraw,paul)],2),at([select(paul,confirm,john)],1),
 at([select(john, order, paul)],0)]) ;
R = sit(mail_order, buy1, 4,
[at([select(paul,notify,john)],3),at([select(john,accept,paul)],2),
 at([select(paul,confirm,john)],1),at([select(john,order,paul)],0)]);
false.
```

Listing 5.15: Game Results in Event Calculus

This has, again, to be checked against the competence requirements set by the Authority Agent so that the approval (or rejection) of the agent's application can be decided.

5.3 Setting the scene - a cyclic protocol

The protocol in Section 5.2 is a simple acyclic one. However, this is not always the case and our formalism should allow to consider complex protocols that contain cycles. As an example, we will be looking at an electronic negotiation protocol in Figure 5.2 adapted from [35]. This protocol has a number of cases of moves that can form part of an infinite loop - for example (this is not a comprehensive list, as other cases of loops exist):

- the customer selects *challenge* and the merchant selects *justify*;
- the customer selects *certify* and the merchant selects *not-understood*;
- the customer selects *inform* and the merchant selects *not-understood*;
- the customer selects *certify*, the merchant selects *not-understood*, the customer selects *inform* and the merchant selects *not-understood*;
- the customer selects *inform*, the merchant selects *not-understood*, the customer selects *certify* and the merchant selects *not-understood*.

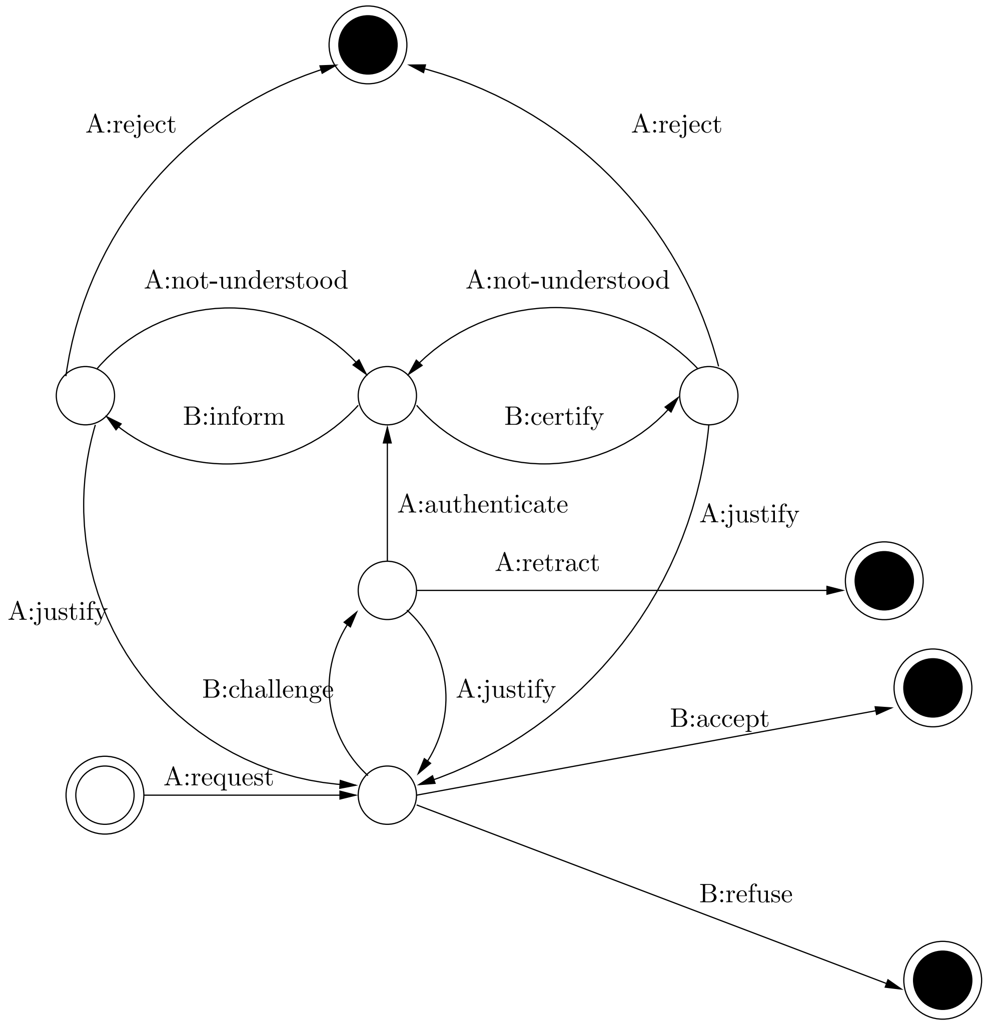


Figure 5.2: An electronic negotiation protocol

The occurrence of any of these cycles could possibly result in an infinite loop and the protocol will never terminate but will keep looping between these states. As we are assuming that the agent selection strategy is private, we have no way of knowing whether it will accommodate exiting from the loop itself or not, so we have to make allowance for it in the processing of protocol moves. In the next Sections, we describe how we deal with this kind of situation in both *Event Calculus* and *Situation Calculus* by discussing the *cyclic/1* predicate; the rest of the formalism is exactly the same.

5.3.1 Specifying cycles in Situation Calculus

In order to check for cycles, we need to check the representation of the state of the game for the occurrence of these moves. In other words, as the state of the game is represented by a list, it is sufficient to check if the specified moves exist in that list in the same sequence. This is done by the *cyclic/1* predicate, defined in Listing 5.16.

```
cyclic(sit(_,_,Moves)):-
    cyclic_pattern(CyclicPattern),
    check_list( CyclicPattern , Moves).
```

Listing 5.16: Identifying cyclic moves in Event Calculus

Intuitively, a move causes a *cyclic* problem, if a certain pattern is matched against the moves already made and the newly selected move. In the protocol of Figure 5.2, such a pattern could be two occurrences of the sequence *select(paul,challenge,john), select(john,justify,paul)*. This would be specified as a cyclic pattern and every new move selected by the agents assuming the roles of supplier and merchant in the protocol will have to be checked against that pattern. If adding the new move will result in a loop, the agent's choice will be rejected and it will have to make another one. We should note here that the validity of the move is not affected and an agent using the protocol is allowed to use that move as many times as it chooses to do so.

We are imposing this limitation only on the grounds of been able to check the agent's ability to terminate the protocol and as there is no control over the agent's selection strategy the only option is to introduce a new layer of filtering for the moves it selects. Furthermore, by using that version of *cyclic*, we can easily adapt it to cater for cases where we want to apply patterns like periodic occurrences of the cyclic moves (e.g. cases like *(LoopMoves,_,LoopMoves,...)*). The specification of problematic moves for the protocol in Figure 5.2 via the specification of cyclic patterns in *Situation Calculus* is shown in Listing 5.17. If

Move is selected, then all cyclic patterns are examined against the list of current moves enhanced with *Move*.

```
cyclic(sit(mail_order,_,Moves), Move):-
    Move = select(B,challenge,A),
    sublist([select(B,justify,A),select(A,challenge,B),
             select(B,justify,A),select(A,challenge,B)],Moves).
cyclic(sit(mail_order,_,Moves), Move):-
    Move = select(B,authenticate,A),
    sublist([select(A,challenge,B),select(B,justify,A),
             select(A,challenge,B),select(B,justify,A)], Moves).

cyclic(sit(mail_order,_,Moves), Move):-
    Move = select(B,challenge,A),
    sublist([select(B,justify,A),select(A,certify,B),
             select(A,certify,B)], Moves).

cyclic(sit(mail_order,_,Moves), Move):-
    Move = select(A,challenge,B),
    sublist([select(B,justify,A),select(A,inform,B),
             select(B,notunderstood,A),select(A,certify,B)],Moves).
cyclic(sit(mail_order,_,Moves), Move):-
    Move = select(A,challenge,B),
    sublist([select(B,justify,A),select(A,certify,B),
             select(B,notunderstood,A),select(A,inform,B)],Moves).

cyclic(sit(mail_order,_,Moves), Move):-
    Move = select(A,challenge,B),
    sublist([select(B, justify, A), select(A,certify,B),
             select(B, notunderstood, A), select(A, certify, B),
             select(B, notunderstood, A),elect(A,inform,B)],Moves).
cyclic(sit(mail_order,_,Moves), Move):-
    Move = select(A,certify,B),
    sublist([select(B,notunderstood,A),select(A,certify,B),
             select(B,notunderstood,A),select(A,certify,B)],Moves).
cyclic(sit(mail_order,_,Moves), Move):-
    Move = select(A,inform,B),
    sublist([select(B,notunderstood,A),select(A,inform,B),
             select(B,notunderstood,A),select(A,inform,B)],Moves).

cyclic(sit(mail_order,_,Moves), Move):-
    Move = select(A,challenge,B),
    sublist([select(B,justify,A),select(A,inform,B),
             select(B,authenticate,A)],Moves).
cyclic(sit(mail_order,_,Moves), Move):-
    Move = select(A,challenge,B),
    sublist([select(B,justify,A),select(A,certify,B),
             select(B,authenticate,A)],Moves).

cyclic(sit(mail_order,_,Moves), Move):-
    Move = select(A,certify,B),
    sublist([select(B,notunderstood,A),select(A,inform,B),
             select(B,notunderstood,A)],Moves).
cyclic(sit(mail_order,_,Moves), Move):-
    Move = select(A,inform,B),
    sublist([select(B,notunderstood,A),select(A,inform,B),
             select(B,notunderstood,A)],Moves).
```

Listing 5.17: cyclic_pattern examples

5.3.1.1 An example run

After running the code for the electronic negotiation protocol, we get the following results for the possible paths of the protocol that can be realised on the basis of the competence information supplied by the participant agents. The results shown in Listing 5.18 are not comprehensive, but the first few paths that we get from running the code. They will have to be matched against the societal competence requirements.

```
-> [ select(paul,accept,john), select(john,request,paul)]

-> [ select(paul,refuse,john), select(john,request,paul)]

-> [ select(paul,accept,john), select(john,justify,paul),
     select(paul,challenge,john), select(john,request,paul)]

-> [ select(paul,refuse,john), select(john,justify,paul),
     select(paul,challenge,john), select(john,request,paul)]

-> [ select(paul,accept,john), select(john,justify,paul),
     select(paul,challenge,john), select(john,justify,paul),
     select(paul,challenge,john), select(john,request,paul)]

-> [ select(paul,refuse,john), select(john,justify,paul),
     select(paul,challenge,john), select(john,justify,paul),
     select(paul,challenge,john), select(john,request,paul)]

-> [ select(john,retract,paul), select(paul,challenge,john),
     select(john,justify,paul), select(paul,challenge,john),
     select(john,request,paul)]

-> [ select(paul,accept,john), select(john,justify,paul),
     select(paul,certify,john), select(john,authenticate,paul),
     select(paul,challenge,john), select(john,justify,paul),
     select(paul,challenge,john), select(john,request,paul)]
```

Listing 5.18: Electronic negotiation protocol specified in SC

5.3.2 Cyclic Rules in *Event Calculus*

We use *cyclic/2* to check for cycles in the *Event Calculus* formulation as well. In the case of *Event Calculus* , as a number of moves can be made by multiple agents at the same time, the definition of *cyclic* (as part of the definition of the *acceptable_2* predicate) is moved inside the *assume/3* predicate as in Listing 5.19.

```
assume(Valid, sit(N,Id,T,Ns), E, at(Es,T)):-
       findall(E, Valid, All),
       member(Es, All),
       acceptable(sit(N,Id,T,Ns),at(Es,T)).
```

Listing 5.19: Assuming a move in Event Calculus

In Listing 5.19, we find all moves that are valid (for all players) and check to see if the move we are selecting is a sublist of them (so that we can capture all different combinations of the moves that can be made from one or more players). The next step is to check if the aggregate move selected is *acceptable* or not (where acceptable is equivalent to the move not introducing a cycle).

The definition of the *cyclic* predicate is shown in Listing 5.20:

```
cyclic(sit(_,_,_,Ns),at(Es,T)):-
    cyclic_pattern(CyclicPattern),
    sublist(CyclicPattern, Ns).
```

Listing 5.20: Definition of a cyclic move

Listing 5.21 shows examples of the specification of cyclic moves in *Event Calculus* .

```
cyclic_pattern([at(select(B,justify,A),_),at(select(A,challenge,B),_),
            at(select(B,justify,A),_),at(select(A,challenge,B),_)]).

cyclic_pattern([at(select(B,challenge,A),_),at(select(B,justify,A),_),
            at(select(A,challenge,B),_),at(select(B,justify,A),_)]).

cyclic_pattern([at(select(B,justify,A),_),at(select(A,certify,B),_),
            at(select(A,certify,B),_)]).

cyclic_pattern([at(select(B,justify,A),_),at(select(A,certify,B),_),
            at(select(B,notunderstood,A),_),at(select(A,certify,B),_),
            at(select(B,authenticate,A),_),at(select(A,challenge,B),_),
            at(select(B,justify,A),_)]).

cyclic_pattern([at(select(B, justify,A),_),at(select(A,inform,B),_),
            at(select(B,notunderstood,A),_),at(select(A,inform,B),_),
            at(select(B,authenticate,A),_),at(select(A,challenge,B),_),
            at(select(B,justify,A),_)].

cyclic_pattern([at(select(B,justify,A),_),at(select(A,inform,B),_),
            at(select(B,notunderstood,A),_),at(select(A,certify,B),_)]).

cyclic_pattern([at(select(B,justify,A),_),at(select(A,certify,B),_),
            at(select(B,notunderstood,A),_),at(select(A,inform,B),_)]).
...
```

Listing 5.21: Example of cyclic patterns

We are making use of the same rules as in the case of *Situation Calculus*, but with the difference that in the case of having two or more players making a move at the same time, our moves for each time point will have to be represented as lists (in this example, there is a single move at every time point, but this does not have to be the case in general).

5.3.2.1 An example run

Running the program in *Event Calculus*produces the following results - Listing 5.22 is a small sample to demonstrate the outcome to be checked against the competence requirements:

```
-> [at(select(paul,accept,john),1),
    at(select(john,request,paul),0)]

-> [at(select(paul,refuse, john),1),
    at(select(john,request,paul),0)]

-> [at(select(paul,accept,john),3),at(select(john,justify,paul),2),
    at(select(paul,challenge,john),1),at(select(john,request,paul),0)]

-> [at(select(paul,refuse,john),3),at(select(john,justify,paul),2),
    at(select(paul,challenge,john),1),at(select(john,request,paul),0)]

-> [at(select(john,retract,paul),4),at(select(paul,challenge,john),3),
    at(select(john,justify,paul),2),at(select(paul,challenge,john),1),
    at(select(john,request,paul),0)]

-> [at(select(paul,retract,john),4),at(select(paul,challenge,john),3),
    at(select(john,justify,paul),2),at(select(paul,challenge,john),1),
    at(select(john,request,paul),0)]
```

Listing 5.22: Results for the electronic negotiation protocol in EC

5.4 Entering the society - Role Assignment

Once an agent is judged to be competent to join a society by been able to realise at least the essential protocols (defined by the AA), it needs to know the protocol(s) it will be engaging in. In this Section, we provide an example of the algorithm for decomposing a protocol into constituent roles. In this case study, we consider a gateway agent who wants to enter a society of merchant and customer agents in order to handle payments on the merchant agent's behalf for purchases made by customer agents.

This society is making use of a variant of the NetBill protocol, described in Figure 3.1 on page 30, in order to process orders and payment. This is the standard NetBill protocol [25, 46], with a number of variations (for a full description of the protocol, see Section 3.2). Although this is a high-level description, NetBill is a protocol widely used in the area of e-commerce applications and transactions.

The formal representation of the protocol is:

$$<N,R,\ S,\ I,\ F,A,\ M,\ V,\ E>,$$

where:

$$\begin{aligned}
N &= \text{NetBill-v1}\\
R &= \{c, m, g\}\\
S &= \{0, 1, 2, 3, \ldots, 8\}\\
F &= \{5\}\\
A &= \{rq, pq, oa, dg, sepo, ssepo, sr, dgk, dga, dgb\}\\
M &= \{(c, rq, \{m\}), \ldots, (c, oa, \{m\})\}\\
V &= \{(0, (m, dga, \{c\})), \ldots, (0, (c, rq, \{m\}))\}\\
E &= \{(0, (m, dga, \{c\}), 1), \ldots, (0, (c, rq, \{m\}), 8)\}
\end{aligned}$$

The NetBill protocol is described by the *protocol* predicate in Listing 5.23. We will use the *gateway* role for illustrating the derivation process as it is the role that enjoys the most benefits from the process.

```
protocol(netbill1,
          [c,m,g],
          [0,1,2,3,4,5,6,7,8],
          0,
          5,
         [rq,pq,oa,dg,sepo,ssepo,sr,dgk,dga,dgb],
         [(c,rq,m), (m,pq,c), (c,oa,m), (m,dg,c), (c,sepo,m),
         (m,ssepo,g), (g,sr,m), (m,dgk,c), (m,pq,c), (m,dg,c),
         (c,oa,m)],
         [(0, (m,dga,c)), (1,(c,sepo,m)),(2,(m,ssepo,g)),
          (3, (g,sr,m)),(4,(m,dgk,c)), (6, (m,dg,c)),(8,(m,dgb,c)),
          (8, (m,pq,c)), (7,(c,oa,m)), (0, (c,oa,m)), (0,(m,pq,c)),
          (0, (c,rq,m))],
         [(0, (m,dga,c),1),(1,(c,sepo,m)),2),(2,(m,ssepo,g),3),
          (3,(g,sr,m),4),(4,(m,dgk,c),5),(6, (m,dg,c),1),
          (8,(m,dgb,c), 1),(8,(m,pq,c), 7),(7,(c,oa,m), 6),
          (0, (c,oa,m), 6),(0,(m,pq,c), 7), (0, (c,rq,m), 8)]).
```

Listing 5.23: Representation of the NetBill protocol

The next step is to calculate the role-specific LTS from the protocol. After processing the transitions, we obtain the result in Listing 5.24. Every transition where the gateway agent g is not a sender or a recipient of the message involved in the original protocol is replaced with a silent action (denoted by *tau*). The top line of the representation denotes the initial state of the LTS (*0*), the number of transitions in it (*12*) and the number of states (*9*). Each of the other lines describe a transition of the LTS in the form of *(InitialState, Move, FinalState)*, e.g. the third line describes a transition from state *2* to state *3* via the move *(m, ssepo, g)*. Most of the moves are *tau*, as the involvement of the gateway agent is minimal.

```
des(0,12,9)
(0,"tau",1)
(1,"tau",2)
(2,"(m, ssepo, g)",3)
(3,"(g, sr, m)",4)
(4,"tau",5)
(6,"tau",1)
(8,"tau",1)
(8,"tau",7)
(7,"tau",6)
(0,"tau",6)
(0,"tau",7)
(0,"tau",8)
```

Listing 5.24: Role-specific representation for the gateway in the NetBill protocol

We can, now, run the branching bisimulation reduction on this LTS. As this role is only involved towards the end stages of the protocol, most of its transitions are silent. Branching bisimulation will tell us if the silent transitions in the LTS can be removed without any loss of knowledge on the role's side. Indeed, in this case, the result of the bisimulation algorithm as shown in Listing 5.25 allows us to reduce the initial protocol (consisting of nine states and twelve transitions) to one of three states, two transitions and no silent transitions.

```
des(0,2,3)
(0,(m, ssepo, g),1)
(1,(g, sr, m),2)
```

Listing 5.25: The role-specific representation for gateway after the *bbe* reduction

As a by-product of the process we also obtain the mapping between the states in the original LTS and those in the new one as shown in Listing 5.26. Every line of this representation is a pair of the form *(OldStateNumber, NewStateNumber)*. For example states 0, 1, 2, 6, 7 and 8 are all merged into state 0 - these are the states with the silent transitions that were merged into a single state.

```
(0,0)
(1,0)
(2,0)
(3,1)
(4,2)
(5,2)
(6,0)
(7,0)
(8,0)
```

Listing 5.26: Equivalence between LTS states

The agent assuming the role of the payment gateway service is not involved in the message-passing when the protocol is in these states and does not need to know about them when assuming its role.

The results obtained for all roles in the original Netbill protocol are shown in Figure 5.3 (the bisimulation is performed by using the `ltsmin` tool of the μCRL2 toolset [15]). For the merchant agent, there is no

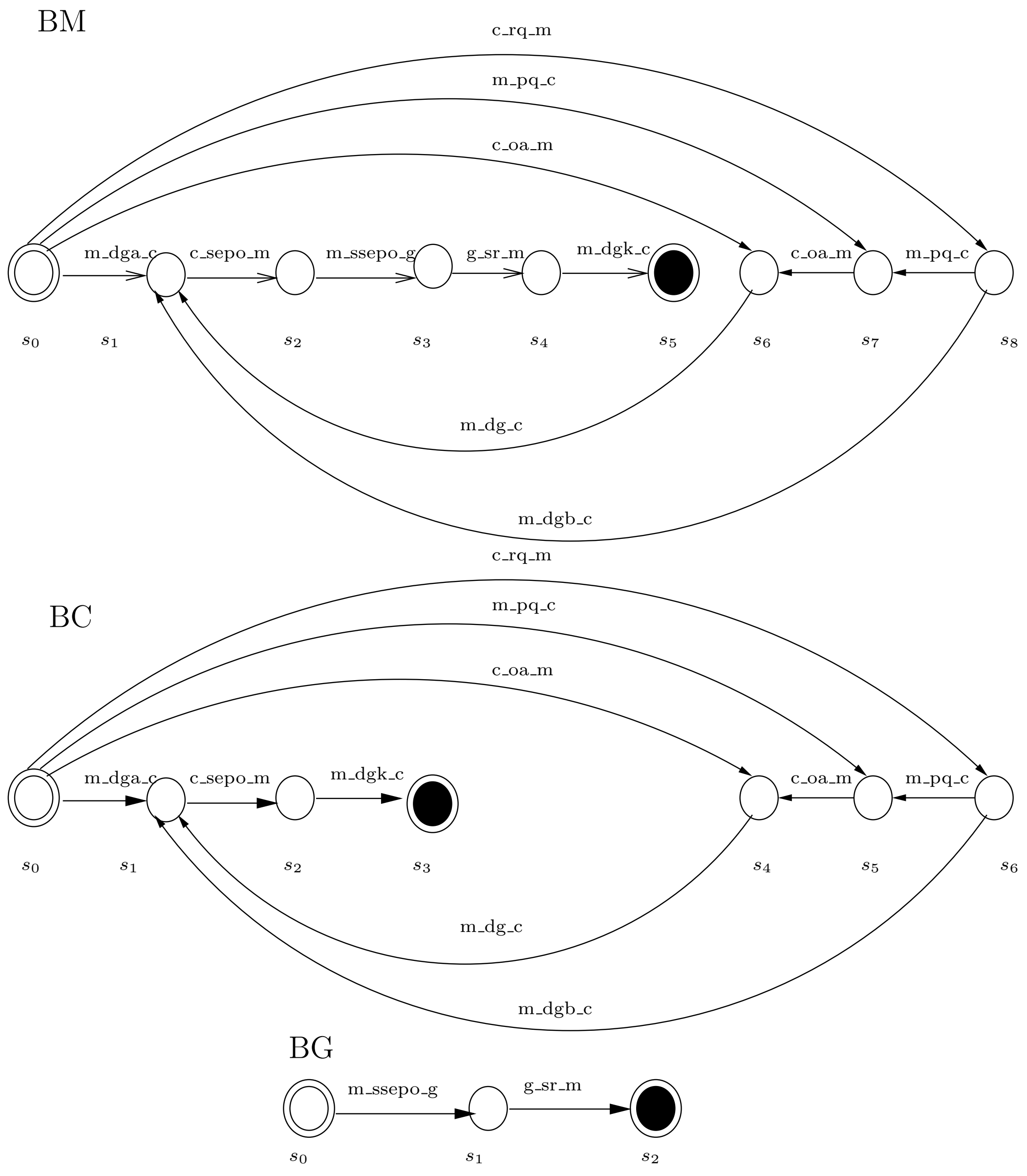


Figure 5.3: Branching bisimulation results for NetBill roles

change and it is receiving the full protocol. This is to be expected as the merchant is involved in all protocol communications and is communicating both with the gateway and the customer agent. As a result, since there is no communication that *does not* involve the merchant agent (either as the sender of the message or as one of its recipients), the protocol that it is receiving is the full one. The protocol for the customer agent is not that much smaller from the original protocol, but it does not include the conversations involving the gateway agent. This is information that the customer should not have access to. It is at the merchant agent's discretion how to handle payment and which gateway (or any other way) to choose in order to receive payment for the order. Also, if the customer agent knew about these interactions there might be the danger that it would try and alter the amount due for the purchase or any other order related information. The real benefit, however, lies with the gateway agent. An agent representing the gateway service needs not know the negotiations that have already happened between the merchant and the customer agent. Furthermore, it only needs to interact with the merchant agent and it does not need to talk to the customer agent at all. As a result, the protocol for the gateway agent after the branching bisimulation reduction is made up only of three states while the original one has nine. To complete the role specification of the gateway according to the definitions in Section 3.4, we need to compute the remaining components, i.e., roleset, initial and final states etc.This is shown in Table 5.1.

Role name	=	g
R^R	=	{m }
S^R	=	{0, 1, 2}
I^R	=	0
F^R	=	{2}
A^R	=	{ssepo, sr }
M^R	=	$\{(m, ssepo, \{g\}), (g, sr, \{m\})\ \}$
V^R	=	$\{(0, (m, ssepo, \{g\})), (1,(g, sr, \{m\}))\}$
E^R	=	$\{(0, (m, ssepo, \{g\}), 1), (1, (g, sr, \{m\}),2)\}$

Table 5.1: Gateway Role Specification

The gateway agent is only interacting with the merchant agent, so this is the only value added to the roleset of the gateway agent. The initial state of the new LTS is the state that is bisimilar to the initial state of the full protocol (that is state 0) and the final state is state 2, which is the bisimilar state of the final state of the original protocol for the gateway role (that is state 5 and, as we can see from Listing 5.26, its equivalent state is state 2). The moves for the role are the moves that have the gateway agent as either the sender or among the recipients of the message and these are $(m, ssepo, \{g\})$ and $(g, sr, \{m\})$. The actions in these moves are the available actions for the role from the original protocol (in this case *ssepo* and *sr*). The valid actions (as well as the effects relationship) for the role in question are obtained by looking at the valid moves in the original LTS, matching the states to the new states in the bisimilar LTS and rewrite the moves and effects rules. As an example, the initial state of the gateway LTS is the class of states in which the initial state (0) of the original LTS belongs - this is the class of states represented by 0.

5.4.1 From Roles back to Protocols

In this Section, we consider whether the *parallel composition* of the role LTSs we get after running *branching bisimulation* and performing any necessary *repairs* on the protocol will give us the original protocol. This is not true; we get neither the original protocol nor one that is branching bisimilar to it. As an example, consider the protocol in Figure 5.4; for brevity, we only show the message names. The protocol begins by role *R1* sending itself one of the messages *d* or *e*. After that, and depending on what was the first message that *R1* sent itself, it sends role *R2* message *a* or *b* and the protocol terminates.

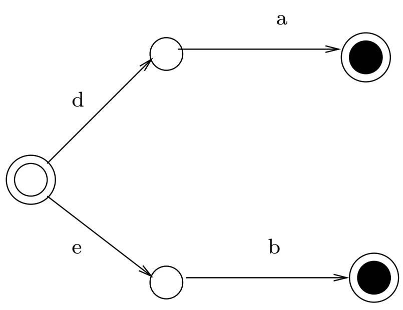


Figure 5.4: From role LTS to the whole protocol - counterexample

The LTS for the two roles where silent actions for the roles are replaced with τ are shown in Figure 5.6.

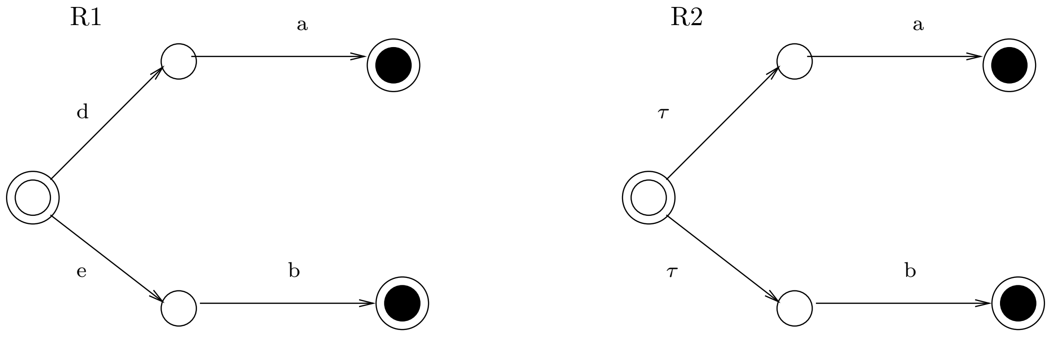


Figure 5.5: Role LTS for roles R1 and R2

Branching bisimulation will make no changes to the LTS for role *R1* as no silent actions exist. It will, also, make no changes to the LTS for role

R2 as the τ actions will be kept in the LTS to preserve the branching structure.

Because of the τ actions in the second LTS, we need to repair the protocol. If the repair algorithm in Section 4.3.3 is used, no repair is performed because both silent actions are followed by receive moves for the role. *R2* does not need to know what message *R1* sent itself, it just waits to receive the next message.

We, now, need to compose the LTSs for the two roles and check the result against the original protocol (or the LTS for role *R1* as they are the same). The parallel composition of the two LTSs will produce the result shown in Figure 5.6.

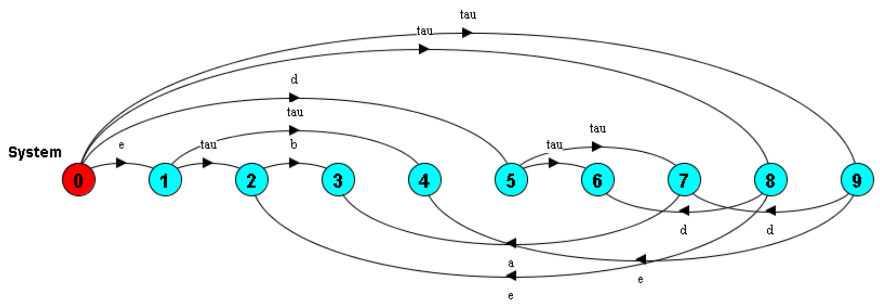


Figure 5.6: Parallel Composition of the LTSs for roles R1 and R2

The `ltscompare` tool of the μCRL2 suite reports that the two LTSs **are not** branching bisimilar.

5.5 Games with multiple instances of the same role

All the protocols we looked at so far were requiring a single instance of each participant role. There are, however, cases when multiple instances of the same role might be required. We will use the English auction protocol of Section 5.5.1 on page 96 with one seller, one auctioneer and two bidders to demonstrate this. The seller provides the auctioneer with the items to be auctioned and the auctioneer conducts the auction. There are two bidders competing for the items and the description of the protocol, in the form of an FSP specification, can be found in Section 5.5.1 on page 96 - we use this as it is an easy and compact way to give a high-level specification of the protocol without overwhelming the user with the full list.

The following Sections describe first the FSP specification of an English auction protocol with two role instances of bidders, how we can convert the protocol from the FSP specification into an LTS one on which we can run bisimulation, as well as the steps involved in the algorithm for checking that the two LTSs describing the behaviour of the two bidders are *branching bisimilar*.

5.5.1 The English auction protocol

The model is described by a specification written in FSP [6]. Without loss of generality the model considers that there are only two bidders, but it can easily be extended to cover for more bidders. The full FSP specification for the *Bidder* agent as well as the composition of the *Bidder* agents processes are given in Listing 5.27.

```
const N = 2

Bidder = (register -> BidderRegister
   | inform -> Bidder
   | end -> Bidder),
    BidderRegister= (accept_registration -> BidderBid
   |  reject_registration -> Bidder
   |  end -> Bidder),
   | BidderBid = (bid -> WaitBid
   |  cancel_bid -> Bidder
   |  inform -> BidderBid
     end -> Bidder) ,
    WaitBid = (accept_bid -> Wait
   |  reject_bid -> BidderBid
   |  inform -> BidderBid
   |  end -> Bidder),
    Wait = (inform -> BidderBid
   |  end -> Bidder).

||BidderI(I=1) =
 Bidder/{   bid[I]/bid,
           reject_bid[I]/reject_bid,
           accept_bid[I]/accept_bid,
           inform[I]/inform,
           cancel_bid[I]/cancel_bid,
           register[I]/register,
           accept_registration[I]/accept_registration,
           reject_registration[I]/reject_registration}.

||Bidders = (forall [i:1..N]
    BidderI(i))/{{win[b:1..N],no_win[1..N],no_win}/end}.
```

Listing 5.27: FSP code for the two bidder auction - Bidder

The constant N is used to identify the number of bidders participating at the auction. We set this to two for convenience, but can easily be changed to accommodate any number of buyers. At the initial state, the bidder can choose between *register*, *inform* and *end* messages. An *inform* or *end* message will take the agent back to the beginning of the process, while a *register* message will cause transition to the *BidderRegister* process.

At this state, the registration can be accepted via an *accept_registration* message that takes the agent to the *BidderBid* state. If the registration is rejected (via a *reject_registration* message) or the bidder quits (via an *end* message), then the bidder agent goes back to the initial state. In the *BidderBid* state, the bidder can place a bid (*bid*), cancel a bid (*cancel_bid*), be informed about a bid (*inform*) or quit the auction (*end*). If the bid is cancelled or the bidder quits the auction, then the agent goes back to the first state in the process. On the other hand, placing a bid will take the agent to the *WaitBid* state. In this state, the bidder agent can have the bid accepted (*accept_bid*), rejected (*reject_bid*), receive an inform message (*inform*) or quit the auction (*end*). A *reject_bid* or *inform* message will take the agent back to the *BidderBid* state, while an *end* message will cause transition to the initial state of the process. If the bid is accepted (*accept_bid*), the system transitions to the *Wait* state. In this state, the agent can either receive an *inform* message that takes it back to the *BidderBid* state from which it can place more bids, or an *end* message which takes the agent back to the initial state.

The specification we have looked at so far, describes the behaviour of a single bidder agent. In our model we are representing two agents; therefore we need to create the parallel composition of two bidder processes. This is done by *action interleaving*, where any *non-shared*, i.e. not with the same name, actions of the two processes can be interleaved arbitrarily but *shared*, i.e. with the same name, actions of the two processes must be executed at the same time. Furthermore, we will need to be able to determine where the move came from, i.e., which agent placed the bid. This is done by combining the bidder agent with an index *i* and is defined as renaming the bidder actions to include the index *i*. As an example, the *bid* message will be renamed to *bid[1]* if the first bidder placed the bid. Then, we compose all bidder processes. In doing so, we need to rename the messages that are common between the Auctioneer and Bidder agents as well as cater for all possible ways of ending an auction (e.g. the *bid* message will come up as *bid[1]* and *bid[2]* because of the composition; these are renamed to *bid* to achieve synchronisation with the *Auctioneer* process). As a result, we create the *Bidders* process by composing all individual *Bidder* processes and renaming all end messages to *win* and *no_win* ones with an index to indicate the agent that has won (or not) the auction. The *no_win* message with no indices covers the case where there is no winner at the auction.

The seller (Listing 5.28) and auctioneer (Listing 5.29) agents have a common action, *init*, that they perform. This takes the seller to the state *WaitInit* where it can receive *accept_init* messages. On receipt of this message, the seller goes to the *WaitEnd* state, in which it receives the *end* message when the auction comes to an end. Finally, if the *init* message is rejected, i.e., a *reject_init* message is received, then the process terminates.

```
Seller = (init -> WaitInit),
   WaitInit = (accept_init -> WaitEnd
|   reject_init -> STOP),
   WaitEnd = (end -> STOP).
```

Listing 5.28: FSP code for the two bidder auction - Seller

The auctioneer agent can accept an *init* message that starts off the process. It takes the agent to the *AnswerInit* state where it can choose between the *accept_init* and *reject_init* messages.

```
Auctioneer = (init -> AnswerInit),
   AnswerInit = (accept_init -> AuctioneerBid[0][0][0]
   | reject_init -> Auctioneer),
   AuctioneerBid[chb:0..N][i1:0..1][i2:0..1] = (
   | when i1 == 1 bid[1] -> AnswerBid[1][chb][i1][i2]
   | when i1 == 0 register[1] -> AnswerReg[1][chb][i1][i2]
   | when i2 == 1 bid[2] -> AnswerBid[2][chb][i1][i2]
   | when i2 == 0 register[2] -> AnswerReg[2][chb][i1][i2]
    stop -> AuctioneerAgreement[chb]),
   AnswerReg[b:1..N][chb:0..N][i1:0..1][i2:0..1] =
    (when b == 1 accept_registration[1] ->AuctioneerBid[chb][1][i2]
    | when b == 2 accept_registration[2] ->AuctioneerBid[chb][i1][1]
    | reject_registration[b] -> AuctioneerBid[chb][i1][i2]),
   AnswerBid[b:1..N][chb:0..N][i1:0..1][i2:0..1] =
    (accept_bid[b] -> InformBidders[b][i1][i2]
    | reject_bid[b] -> AuctioneerBid[chb][i1][i2]),
   InformBidders[b:1..N][i1:0..1][i2:0..1]=
     InformBidders[b][1][i1][i2],
     InformBidders[b:1..N][i:1..N][i1:0..1][i2:0..1]=
       if (i==1 && i!=b && i1==1) then (inform[1] ->
        InformBidders[b][2][i1][i2])
       else if (i==1 && (i==b || i1==0)) then
        InformBidders[b][2][i1][i2]
       else if (i==2 && i!=b && i2==1) then
        (inform[2] ->AuctioneerBid[b][i1][i2])
       else AuctioneerBid[b][i1][i2],
     AuctioneerAgreement[0] = (no_win -> Auctioneer),
    AuctioneerAgreement[chb:1..N] =
     (win[chb] -> Auctioneer
     | no_win[chb] -> Auctioneer).
```

Listing 5.29: FSP code for the two bidder auction - Auctioneer

In the case that the *reject_init* is chosen, the agent goes back to the beginning, as the initiation of the auction is denied (e.g. because the seller agent does not want to sell the goods any more).

On the other hand, if the *accept_init* message is chosen, the process moves to the *AuctioneerBid[i][j][k]* state. The state description uses three indices, i for holding the winning bidder and j and k to hold information about whether the buyer agents are registered or not (a value of one means they are registred). This is the state where the auctioneer agent can receive registration (*register[i]*), bidding (*bid[i]*) and *stop* messages. Both *register[i]* and *bid[i]* messages need to have an index, as we need to differentiate between the two bidders and be able to check who has registered/bid and who has not. A *register* message will cause transition to the *AnswerReg[i][j][k][l]* state with four indices. The first index stores the index of the bidder who has requested registration, the second one holds the index of the current winning bidder and the last two take the values zero (if the respective agent is not registered) or one (if it is). A *bid* message causes the auctioneer process to progress to the *AnswerBid[i][j][k][l]*, where the semantics of the indices are the same as for the *AnswerReg* state.

The auctioneer, once in the *AnswerReg* state, can respond with an *accept_registration* or *reject_registration* message (both of them need an index to identify the agent who has applied for registration). If the application for registration is successful from the first (second) bidder agent, the second (third) index of the *AnswerReg* state will change to one. If the auctioneer declines the application, there is no change in the indices. The same rules hold for the *AnswerBid* state (instead of *accept_registration* and *reject_registration* messages, there would be *accept_bid* and *reject_bid* messages). If the bid is rejected, then nothing needs to be done and the auctioneer will go back to the *AuctioneerBid* state, where it can receive *bid* messages.

If the bid is accepted, the bidders participating in the auction will need to be informed of this development. However, it is only the registered bidders different than the one who made the bid that will receive an *inform* message. The *InformBidders* initially has three indices - the first one about the bidder who is the winning bidder (the bidder who just placed the bid) and the second and third ones are one or zero depending on whether the agent is registered or not.

In order to inform all bidders, we need to keep track of the bidder who was informed last. This calls for an additional index to be added to the *InformBidders* process holding the id of the bidder to be informed about the new bid. As an example, *InformBidders[2][1][1][1]* means that the winning bidder is the second one, the first bidder is the next one to be informed of the bid and both bidders are registered bidders. In our example if the second index is one and the first bidder is not the one who make the bid (first index different than one) and the first bidder is registered (third index is one) then the *inform[1]* message is sent. The process, then, stays in the *InformBidders* state, but the second index changes to two. If the second agent is the one who made the bid or it is not registered, the process sends no *inform* message and moves on to the next bidder. When the last bidder is informed, the auctioneer moves again to the *AuctioneerBid* state where the agent can accept new bids.

In the event that the auctioneer agent receives a *stop* message, it moves to the *AuctioneerAgreement* state. This state takes only one index that describes the winning bidder. A value of zero would indicate that there is no winner and the process goes back to the initial state so that another auction can begin. If there is a winning bidder, then a *win* or *nowin* message is sent to the bidders and the auctioneer agent goes back to the initial state for the next auction. The FSP specification for the auctioneer is specified in Listing 5.29.

To obtain the full protocol we need to compose together the processes of all role instances. In doing so, the *end* message in the *Seller* agent specification will need to be renamed in accordance with the renaming in Listing 5.27. This is so that the seller will receive the information about who won (not won) the auction as well as the *no_win* message in the case that there was no winner. The FSP specification for the composition of the full system is given in Listing 5.30.

```
||System = (Auctioneer ||
    Seller/{{win[b:1..N],no_win[1..N],no_win}/end} || Buyers).
```

Listing 5.30: FSP code for the two bidder auction - System

So far, we have built the system in FSP for describing the auction with two bidders, an auctioneer and a seller. Our aim, however, is to use this model to decide if the protocol provides all instances of the same role (in this case bidder) with the same behaviour. We need to check this, as the specification might specify different rules for different role instances. As an example, if an instance represents a “trusted” bidder, its registration requested is automatically accepted, otherwise the auctioneer might need to run some tests before accepting it. In order to achieve that, we need a formalism similar to that of describing messages in Section 3.2, i.e., in the form *(Sender, Move, Recipients)*.

In our case the different components of a move are separated by _, e.g. the renaming of *bid[1]* to *b1_bid1_a* will imply that *b1* sent the message *bid1* to *a*. If the last component has the value of *all*, this indicates that all agents receive the message, i.e., the auctioneer (*a*), the seller (*s*) and all bidder agents (*b1,b2*).

The FSP code for the renaming is provided in Listing 5.31, e.g. the first line will rename transition labels of *bid[1]* to *b1_bid1_a*. The non-bidder actions have no index associated with them as there is a single agent participant for each of the roles of auctioneer and seller. Some of these actions (the ones concerning the outcome of the auction) are broadcasted to all agent participants. These are the ones that have *all* as the value of the *Recipients* field.

```
|| Auction = System /{
      b1_bid1_a/bid[1],
      b2_bid2_a/bid[2],
      a_rejectbid1_b1/reject_bid[1],
      a_rejectbid2_b2/reject_bid[2],
      a_acceptbid1_b1/accept_bid[1],
      a_acceptbid2_b2/accept_bid[2],
      a_inform1_b1/inform[1],
      a_inform2_b2/inform[2],
      b1_cancelbid1_a/cancel_bid[1],
      b2_cancelbid2_a/cancel_bid[2],
      b1_register1_a/register[1],
      b2_register2_a/register[2],
      a_acceptregistration1_b1/accept_registration[1],
      a_acceptregistration2_b2/accept_registration[2],
      a_rejectregistration1_b1/reject_registration[1],
      a_rejectregistration2_b2/reject_registration[2],
      a_win1_all/win[1],
      a_win2_all/win[2],
      a_nowin1_all/no_win[1],
      a_nowin2_all/no_win[2],
      a_nowin_all/no_win,
      a_acceptinit_all/accept_init,
      s_init_a/init,
      a_rejectinit_s/reject_init,
      a_stop_a/stop
     }.
```

Listing 5.31: FSP code for the two bidder auction - Auction

5.5.2 Verifying the correctness of the description

As already mentioned, we are trying to determine if the LTSs describing the two bidders are describing the same behaviour by checking if they are *bisimilar.* In this case the protocol is not given in the form of an LTS but in FSP so we need to convert it before applying the algorithm. The process followed for converting from FSP format to LTS cannot be fully automated, as there is information we need that will have to come from outside the protocol and sometimes might be uncertain. As an example, we do not know in advance how many instances of each role we need in order to capture all messages. This might be achieved by a *trial and error* approach (trying to increase the number of role instances and seeing if the new trace contains any additional messages).

5.5.3 Preparation

This is an example of the *verification* algorithm specified in Section 4.4.1. The protocol specification given is not in our framework but in FSP, so we need to convert it once we get the full protocol. As the protocol is given in terms of individual FSP processes, we need to compose them to get the full protocol. However, as multiple role instances from the same role are involved, we need to index their actions so we know which role instance is performing which move. As a result, the composition of bidder processes for the two role instances in Listing 5.27 includes an index (i) that carries this information. Furthermore, the moves specified in the FSP format will need to be changed to reflect the specification of moves in our framework (*(Sender, Act, Recipients)*). Finally, the transitions we will get from the LTSA tool will need to be converted into a format the tool that performs the bisimulation (μCRL2) uses, i.e., in the form of *(InitialStateNumber, Move, FinalStateNumber).*

The steps we need to take therefore for bringing the model to a format on which bisimulation can be applied are - this would correspond to step one from Listing 4.6:

(A) index the actions of each process with multiple role instances (e.g. the Bidder process in FSP is specified as the composition of two Bidder(i) processes);

(B) compose all role processes to obtain the full protocol;

(C) change the name of moves so that they comply with our framework (e.g. *bid[1]* becomes *b1_bid1_a*);

(D) convert the representation of the transitions from the FSP format to that of our framework (e.g. *Q1 = (a_rejectinit_s → Q2* should become *(1,(a,rejectinit,s),2)*;

(E) as we will be checking the LTSs for branching bisimulation, the transition names should be the same in both LTSs, otherwise the check will fail. Role or move elements of transitions can be renamed as long as they do not appear in the same role's LTS with multiple indexes (as is the case of *win1* and *win2* that appear in both bidder LTSs).

5.5.4 Converting the protocol specification

The first step is to produce the game LTS from the FSP specification (part (i), line 7 from Listing 4.6). From the *Transitions* menu of the LTSA tool, we can get the transitions of the LTS as specified in FSP. A small sample of them is shown in Listing 5.32.

```
Process:
    Auction
States:
    70
Transitions:
    Auction = Q0,
    Q0  = (s_init_a -> Q1),
    Q1  = (a_rejectinit_s -> Q2
          |a_acceptinit_s -> Q3),
    Q2  = STOP,
    Q3  = (a_stop_a -> Q4
          |b2_register2_a -> Q5
          |b1_register1_a -> Q69),
    Q4  = (a_nowin_all -> Q2),
    ...
```

Listing 5.32: Output of LTSA for the auction model with two bidders

The next step is to convert this information into a format that can be read by the branching bisimulation tool (i.e. (InitialStateNumber, Move, FinalStateNumber)). This is done by reading the transitions line by line and making the necessary conversions. As an example, the state *Q0* should become *0* and *s_init_a* should become *(s,init,a)*. Any move that has *all* as the designated recipient should have it expanded to *[s,b1,b2]*. This is the case as these messages have the auctioneer agent *a* as the sender, so they need to be sent out to the other agents. The first line is stating that this LTS has 162 transitions, 70 states and that the initial state is 0. The results, again a small sample, are shown in Listing 5.33.

```
des(0,162,70)
(0,(s,init,a),1)
(1,(a,rejectinit,s),2)
(1,(a,acceptinit,s),3)
(3,(a,stop,a),4)
(3,(b2,register2,a),5)
(3,(b1,register1,a),69)
...
```

Listing 5.33: Conversion of the output of LTSA to aut

In parallel, we need to create the *protocol* representation as specified in Section 5.4 on page 89. This is handled by the same code that converts the output of the LTSA *Transitions* menu into the role LTS. The result is shown in Listing 5.34.

```
protocol(auction2,
[s,a,b2,b1],
[0,1,2,3,4,5,40,6,7,16,29,30,8,9,10,11,15,12,13,14,17,18,19,20,21,
22,28,23,24,25,27,26,31,32,35,37,39,33,34,36,38,41,42,44,47,43,
45,46,...],
0,
2,
[init,rejectinit,acceptinit,stop,register2,register1,nowin,
...],
[(s,init,a),(a,rejectinit,s),(a,acceptinit,[s,b1,b2]),(a,stop,a),
...],
[(0,(s,init,a)),(1,(a,rejectinit,s)),(1,(a,acceptinit,[s,b1,b2])),
...],
[(0,(s,init,a),1),(1,(a,rejectinit,s),2),...]).
```

Listing 5.34: Protocol representation for the auction protocol

We, now, need to prepare the role-specific representations for the two bidder role instances (still part (i) of Listing 4.6). This is done by looking at the *effects* component of the protocol definition and reading and processing each element. If the role for which we are producing the LTS is not the sender or amongst the recipients of the message, then the move component is substituted by tau.

The representations for the two role instances,small samples, are shown in Listings 5.35 and 5.36.

```
des(0,162,70)
(0,tau,1)
(1,tau,2)
(1,tau,3)
(3,tau,4)
(3,tau,5)
(3,(b1,register1,a),69)
...
```

Listing 5.35: Role-specific LTS for the first bidder

```
des(0,162,70)
(0,tau,1)
(1,tau,2)
(1,tau,3)
(3,tau,4)
(3,(b2,register2,a),5)
(3,tau,69)
...
```

Listing 5.36: Role-specific LTS for the second bidder

Before we move on to the second step, checking the LTSs for bisimilarity, we need to make sure all transitions have the same names, as the check will fail if they are different (not only the structure has to be the same, but the action names have to match). As an example, a transition called *(b1, bid1, a)* in the first bidder's LTS and one called *(b2, bid2, a)* in the second bidder's LTS denote exactly the same thing; the bidder is placing a bid for the auctioned item, but the bisimulation check will fail because of the difference in the names. All actions with an index need to be renamed before we run the check for bisimulation, i.e., *b1* will become *b*, bid1 will become bid and so on (this applies to messages going to multiple recipients; *b1* and *b2* will be replaced by *b*). However, there are messages (*win1,nowin1,win2,nowin2*) that need to be treated in a different way than simply renaming them to *win, nowin.* This is because their meaning is special and depends on whose bidder LTS we are looking at. As an example, *win1* appearing in the first bidder's LTS will mean that it won the auction, whereas when appearing in the second bidder's LTS it will mean that the other bidder won the auction. For this reason, we rename them as follows: *win1, nowin1* becomes *winme, nowinme* in the first bidder's LTS and *winyou, nowinyou* in the second bidder's LTS, while *win2, nowin2* becomes *winyou, nowinyou* in the first bidder's LTS and *winme, nowinme* in the second bidder's LTS (so *winme* means that the bidder whose LTS we are looking at won the auction).

After applying the name changes in Listings 5.35 and 5.36 we get the LTSs for the first and second bidder role instance shown in Listings 5.37 and 5.38 respectively.

```
des(0,162,70)
(0,tau,1)
(1,tau,2)
(1,tau,3)
(3,tau,4)
(3,tau,5)
(3,(b,register,a),69)
...
```

Listing 5.37: Role-specific representation for the first bidder

```
des(0,162,70)
(0,tau,1)
(1,tau,2)
(1,tau,3)
(3,tau,4)
(3,(b,register,a),5)
(3,tau,69)
...
```

Listing 5.38: Role-specific representation for the second bidder

5.5.5 Checking the protocol role instances for bisimilarity

We can, now, run step two (line 9) of Listing 4.6, *branching bisimulation.* The *ltscompare* tool from the μCRL2 tool reports that indeed the two LTSs are *branching bisimilar* (step 3, line 10 of Listing 4.6) and the *check_role_automata* algorithm returns true.

This means that the two bidders exhibit the same behaviour. If this was not the case, the business protocol should explicitly specify it. As an example, consider the case that two bidder place the same bid and that is the final bid of the auction. The auction house might have a rule specifying that the bidder who is registered with them wins the item. This should have been explicitly specified in the protocol specification.

The representation for the first bidder instance is shown in Listing 5.39. It now contains sixty-one transitions and twenty-six states rather than the starting a hundred and sixty-two and seventy respectively. Any silent actions that were not repaired remain in the bidder's file as τ actions and indicate that further repair might be needed on the auction protocol.

```
des (0,61,26)
(1,"(b,bid,a)",14)
(4,"(a,acceptregistration,b)",1)
(14,"(a,rejectbid,b)",1)
(3,"(a,inform,b)",1)
(2,"(a,inform,b)",1)
(17,"tau",2)
...
```

Listing 5.39: Final LTS for the first bidder

As the representation contains *tau* actions, we need to repair the role specification to make it implementable. As we already know that the LTSs are bisimilar we only need to repair one of the two; we choose to do it for the first bidder. The repair is done using the approach in Section 4.3.3 on page 71.

After repairing the role LTS for the first bidder, we get the representation shown in Listing 5.40 where some moves have been extended to include the bidder agent among their recipients so as to provide it with extra knowledge about the current state of the protocol.

```
des(0, 162, 70)
(53,"(a,rejectbid1,[b1, b2])",52)
(68,"(a,acceptregistration1,[b1, b2])",52)
(52,"(b1,bid1,[a, b2])",53)
(21,"(b1,cancelbid1,[a, b2])",6)
(68,"(a,rejectregistration1,[b1, b2])",6)
(6,"(b1,register1,[a, b2])",68)
...
```

Listing 5.40: Protocol representation after the repair of silent transitions

For example the result of move *register1* from the first bidder is now sent to the second bidder as well or the fact that the first bidder placed a bid is now communicated to the second bidder apart from just the auctioneer.

Following that, we need to check if further repairs of the protocol are required. Thus, we create the new role-specific LTS for the first bidder from the updated protocol by replacing any action it is not involved in by a silent τ action, run branching bisimulation on the resulting protocol LTS and check if it still contains silent actions. The generated data for the protocol after the addition of recipients to certain moves because of the repair algorithm is displayed in Listing 5.41 and is the file for the first bidder generated, this time, from the updated protocol.

```
des(0, 162, 70)
(53,"(a,rejectbid1,[b1, b2])",52)
(68,"(a,acceptregistration1,[b1, b2])",52)
(52,"(b1,bid1,[a, b2])",53)
(21,"(b1,cancelbid1,[a, b2])",6)
(68,"(a,rejectregistration1,[b1, b2])",6)
(6,"(b1,register1,[a, b2])",68)
...
```

Listing 5.41: The first bidder after making repair changes

The LTS produced after running branching bisimulation on the repaired LTS of Listing 5.41 is shown in Listing 5.42.

```
des(0,61,26)
(1,(b1, bid1, a),14)
(4,(a, acceptregistration1, b1),1)
(14,(a, rejectbid1, b1),1)
(3,(a, inform1, b1),1)
(2,(a, inform1, b1),1)
(17,tau,2)
...
```

Listing 5.42: First bidder after repairs and *bbe*

As we see from the first bidder's LTS, there are still some remaining silent actions; however, they do not influence the decision-making process of the first bidder. If it goes from state 10 to state 3, then it can only receive a message from the auctioneer about another bidder placing a bid so there is no conflict.

5.6 Summary

In this chapter, we provided examples for the framework representation and algorithms described in Chapters 3 and 4. We looked at how protocols can be described as games in both *Situation Calculus* and *Event Calculus* notation, as well as demonstrated how cyclic patterns can be established so that we can filter them out of protocol runs. These processes can be used to prove (or refute) competence of an agent with regards to a specific protocol. We also showed how to provide an agent with the minimal necessary information for it to play its role in a protocol as well as decide on the correctness of a protocol description if it involves multiple instances of the same role. We accommodate this by creating the role-specific LTSs for the different instances, renaming the role and action names so that we have equal grounds for comparing them and demanding that they are **branching bisimilar**, as they exhibit the same behaviour. If that is not the case, then the protocol is not correct in that different instances of the same role do not exhibit the same behaviour. The next chapter will be comparing and contrasting our approach to other approaches to *agent competence* as well as *protocol decomposition* for providing the agent with the role-specific information it should receive.

Chapter 6

Discussion

6.1 Introduction

In this chapter, we review other approaches on *competence checking* and *protocol decomposition* and compare them with our approach that was covered in Chapters 3, 4 and 5. We focus on these two concepts as they are the main contributions of the work and briefly look at different ways for describing agent interaction either in the form of games or not. The approaches are grouped by the main contributors. In order to make the presentation in this chapter self-contained, we summarise the necessary features of the approaches to aid the comparison with our work.

6.2 Competence Checking

In the area of *competence checking*, dealing with whether an agent can enact a given protocol or not, there seems to be a general consent that if an agent is found to be *competent* (able to enact it), then it should be found to be *interoperable* as well (if enacting the protocol, it should be able to reach the terminating states without causing deadlocks) [4, 7, 10, 24]on the grounds of the other agents acting exactly as prescribed by the protocol. If two agents can engage in conversation that can possibly lead to a terminating state, they should be able to actually reach that state when they take on their roles in the context of the protocol specified. This, in general, is achieved by looking at the agent's private strategy (essentially, the way it selects its next move), as well as the protocol. The protocol is, then, transformed to take into account actions from the private strategy of the agent likely to prevent the agents from terminating the protocol, which are discarded. Any extra actions that are not going to cause any deadlocks are incorporated into the protocol by creating *trap* states, i.e., states from which the agent can only transition to the same state. The final protocol that is given to the agents is the modified one so the agents are made interoperable *by design.*

6.2.1 Singh et al.

In [24], Singh et al. address the issue of *competence* as well as that of *conformance.* The idea is to examine whether the agent in consideration can produce some (or all) of the computations predicted by the protocol. The representation of the protocol is that of a *transition system* from which *paths* are defined as a series of transitions that take an agent from an initial state to a final one and a *run* as only the sequence of states in the aforementioned transition. Furthermore, the *t-span* of a transition system T, $[\mathcal{T}]$, is defined as the set of all paths in the transition system, i.e., all sequences of paths that take the agent from an initial state to a final one.

They also address the issue of how to deal with cases where the agent does not follow the rules of the protocol strictly, but is allowed some deviations from it as long as the main goals of the protocol are not affected; e.g. in a purchase protocol, if the customer requests an invoice before making payment and the original protocol does not prescribe this behaviour. Rather than disallowing this action on the basis that it is not supported by the protocol, they choose to allow it as the basic commitment that the customer pays and then the merchant ships the goods is not affected. On the other hand, if the customer requested the goods to be shipped before payment is made, it would have been a major violation of the pay-before-ship commitment and not allowed.

As the authors allow for deviations from the original protocol, a decision has to be made as to what constitutes a minor (major) violation. The idea is that we want to allow for deviations in which the current and the resulting states of the protocol remain *unchanged* on the grounds of a similarity function - in this case, the *state similarity* one, which, as an example, can be based on the notion of commitments [71]. Formally, we say that $s_i \approx_f s_j$ if the same set of commitments holds in both states; if the non-anticipated move does not cause any change in what both agents are supposed to deliver, then it is deemed to be a minor violation and, thus, allowed.

Furthermore, a temporal relation is defined as follows: $s \succ_\tau s'$ if s occurs before s' in the run τ. In this context, we would say that run τ_j subsumes run τ_i ($\tau_j \gg_f \tau_i$) if for every state s_i in τ_i there is a state s_j in τ_j such that $s_j \approx s_i$ and for all $s_i{}'$ in τ_i if it is true that $s_i \succ_{\tau_i} s_i{}'$ then there is $s_j{}'$ in τ_j such that $s_j \succ_{\tau_j} s_j{}'$ and $s_j{}' \approx_f s_i{}'$. In other words, a run subsumes another one if we can relate them through a state-similarity function and we can rank the similar states in the same order using the temporal relation $\succ$ as in the original run.

The other important element is the concept of *closure* for a protocol which is defined as $[[\mathcal{P}]]_f = \{\tau | \forall \tau' \in [\mathcal{P}]: \tau >>_f \tau'\}$, i.e., all the runs that subsume the runs provided by the set of original runs for the protocol so that we can extend the initial protocol rules. The span of the agent α ($[\alpha]$) will be the paths in the protocol transition system that the agent can realise.

Following that, we define the terms *conformance* and *coverage* of an agent α with respect to a protocol $\mathcal{P}$ and a state-similarity function f:

- the agent is *conformant* with the protocol if $[a] \subseteq [[\mathcal{P}]]_f$, i.e., the agent cannot produce interactions that break the commitments holding in similar states.
- the agent is *covering* the protocol if for every run in the protocol span ($\tau \in [\mathcal{P}]$), there is another run τ' in the agent's span ($\tau' \in [\alpha]$) that subsumes it ($t' >>_f t$), i.e., the agent can subsume all protocol runs, at least matching them or adding extra states without modifying the commitments).

Following from these definitions, all we have to do to make sure that two agents will be *interoperable* is work out the parallel composition of their individual specifications and rule out, by the use of trap states, the paths that can cause problems - these are:

- *deadlock* paths, i.e., paths when both agents have to execute a *receive* action but the send actions for both messages have not been taken;
- *blocking* paths, i.e., paths when some receive actions have to be performed by the agent, but the send action for that message has not been performed;

- *out-of-order* paths, i.e., the agent receives the messages in a different order than the one they were sent.

As there is no control over how the interactions will be co-ordinated, no restriction is placed on them (e.g. forcing rendezvous style or any other).

In order to develop interoperable agents, we need to take their individual transition systems and rule out any *problematic* combinations, i.e., deadlock, blocking and out-of-order paths.

In another paper [112], it is claimed that a big class of protocols - especially those used in electronic commerce applications, can be looked at from the viewpoint of commitments and operations (creation, manipulation, delegation) on them.

They claim that in order to check compliance one cannot adopt a local view, due to the heterogeneity of the agents that comprise the system, but has to look at it externally as an observer (as not knowing the internal workings of the member agents will make only observable behaviour meaningful). In order to observe the moves of the agents, a central point of reference is needed (in the case of a distributed system no local view would be reliable). Thus, a global clock is used with its starting value set to $\overrightarrow{0} \triangleq \langle 0, \ldots, 0 \rangle$.

It considers *local models*, which are nothing more than a collection of messages sent and received by the agent (the timestamp will reference the time for each agent that the message was sent or received).

In this model, every state is identified with a message and at every state a set of messages holds, i.e., $Q = \{m{:}\,\text{m is a message}\} \cup \{\overrightarrow{0}\}$, i.e., all the messages that have been sent in the protocol so far as states are identified with messages, as well as $\overrightarrow{0}$, which is the starting point.

6.2.1.1 Comparison

In [24] the notion of conformance used could correspond to our notions of either *Competent under Adversity* or *Competent under Co-operation* assuming that the agent does not do any action that breaks the protocol. This is the case as in all degrees of competence, we require that the agent selects valid moves. Regarding the properties of the agent, assuming it selects only valid moves, if the agent is *conformant* then it would be at least *Competent under Co-operation*; the agent span is a subset of the protocol span and the agent can not realise all protocol actions. If the agent is *covering* the protocol (again on the assumption that it is always selecting valid moves), then it is *Fully Competent* as it can match all protocol runs.

This approach has a number of limitations, namely (these constraints are mainly imposed due to the approach for checking interoperability):

- it accepts only one action per transition;
- an agent is not allowed to play twice ;
- we cannot have two different transitions that take you to the same state;
- actions can only occur once in every path.

In our approach, we can represent concurrent actions with the use of *Event Calculus.*

In [112] our concept of *competence* in this approach relates to that of *compliance* to protocol rules, as long as the agent does not select a move that is not specified by the protocol (regardless of whether any base-level commitment is broken or not). They are, also, making assumptions about the agents' behaviour, i.e., agents are benevolent and they do not forge message timestamps. Our approach makes no such assumptions.

The use of the global clock for ensuring a central point of reference is similar to our approach where the Authority Agent of the society has full knowledge of the protocol and does the checking on the basis of the information submitted by individual agents. However, although they present a technique for flagging violations based on *commitments*, they offer no classification of the agents in terms of compliance levels. Some suggestions are adopted, but no formal classification is made.

The way they represent states is equivalent to the way we represent them in *Situation Calculus* (set of all actions happened so far in the game) and *Event Calculus*(set of properties that can be verified by looking at the actions so far carried out in the game). It is a run-time approach in contrast with ours, which checks competence *at design time.* This means that they check the agent's competence when the protocol is actually running, without having any knowledge of the agent strategies or any other information regarding their skills.

6.2.2 Baldoni et al.

Baldoni et al. in [7] recognise the similarities between web services and multi-agent systems in that they both comprise of heterogeneous, proactive and independent components operating in open, dynamic and unpredictable environments with no form of central control. Therefore rules are required in order to govern the interactions between the involved parties - these would be *protocols* in the case of agent societies and *choreographies* in the case of web services.

Moreover, due to the heterogeneity of the components as well as the dynamic nature of the environment in which they interact we cannot rely on internal constructs such as beliefs and intentions to do the verification. Instead, we will have to use the observable behaviour of the parties involved.

As the membership of the MAS or web services will be changing constantly, we need to reason about which applicant will be allowed entry and whose application should be rejected. This reasoning can be performed either *a priori* or at *runtime* and will be associated with whether the agent's behaviour is in line with the behaviour specified by the protocols for the role(s) it is interested in taking on in the new societal settings.

The authors expect the agents (web services) to fully respect the rules of the protocol (choreography) and not deviate from the specified interactions. They are, also, expecting two agents who have independently be proven *conformant* to the protocol to be *interoperable* as well - if we can prove that they can follow the protocol, then we expect them to do so if asked to participate in it.

They restrict their attention to protocols that can be represented by Finite State Automata; in fact it is required that both the specification of the protocol as well as its implementation (policy) is specified by a Finite State Automaton. These protocols can easily be described by our approach as we are using the same representation. This is a property that many protocols from the area of Multi-Agent Systems satisfy so it is a general approach in this respect. Furthermore, the expression of the agent's policy rules in a logic-based declarative language makes the transformation to a Finite State Automaton even easier.

The basic concept is to look at the problem as a problem of relating two languages: that of the protocol $\mathcal{P}_{spec}$ and that of the agent's policy $\mathcal{P}^{ag}_{lang}$

(where *lang* is the language in which the policy is specified and *ag* is the identifier of the agent).

The protocol specification defines a set of *legal* conversations - as a result, a conversation is *legal* if it belongs in the language defined by the protocol specification language. In our approach, this conversation will be a sequence of *valid* moves. Thus, the agents will be labelled *conformant* if they can produce a legal conversation with respect to an extended automaton that is derived from $\mathcal{P}^{ag}_{lang}$ and $\mathcal{P}_{spec}$.

The protocol specification defines a set of *legal* conversations - as a result, a conversation is **legal** if it belongs in the language defined by the protocol specification language. Thus, the agents will be labelled *conformant* if they can produce a legal conversation.

The proposed algorithm aims at fulfilling the following expectations:

- the policy should be able to deal with any possible incoming message that the protocol foresees and do nothing if the policy foresees a message that is not supposed to be received at that point according to the protocol;

- the agent's policy should not utter a message that is not expected at this stage according to the protocol;

- the agent's policy allows it to utter at least one of the messages foreseen by the protocol (it is not necessary to cater for all legal messages, but at least one should be catered for);

The solution proposed is to start with the automaton that accepts the intersection of the two languages (agent and protocol) and then extend it so that it includes all conversations we want to allow. This is done via an *IO* automaton [70]. The automaton is defined as follows: E_q is the set that consists of messages that cause a transition from state p to state q; formally $E_q = \{m \mid \delta(p, m) = q\}$for p,q $\in Q$, where Q is the set of the automaton states. Then, the automaton from which the states p and q are drawn will be an *IO*-automaton for agent *ag iff* for every $q \in Q, E_q$ consists of only incoming or only outgoing messages with regards to agent *ag*.

Starting with the automaton that accepts the intersection of the languages accepted by the agent and the protocol specification, it is sufficient to apply the following transformations:

- a state q that has only incoming messages coming into it and leads to a final state for the agent's policy but does not for the protocol specification, is made a trap state (the transition function always stays in the same state) and is added to the final states of the automaton. This would cover the case in which the agent's policy allows, at a certain state, a message that is not prescribed by the protocol. Assuming that the other protocol participants are following the rules, any paths originating from that state can be safely ignored;

- a state that has only incoming messages coming into it and leads to a final state for the protocol specification but not for the agent's policy, is made a trap state (for every state, the resulting state is the same) and there is no change for the set of final states. This covers the case in which, at a certain state, the protocol allows for a message to be received that the agent can not receive. If the agent receives this message, all paths originating from that state will be rejected by M_{conf};

- a state that has only outgoing messages coming out and leads to a final state for the agent's policy but not for the protocol specification is made a trap state (all transitions are self-transitions) and there is no change to the set of final states. This rule handles the case where the agent can utter a message that is not allowed according to the specification of the protocol. As a result, all conversations originating from that state will not be accepted by M_{conf};

- a state that has only outgoing messages coming out and leads to a final state for the protocol specification but not for the agent's policy is made a trap state (all transitions lead to the same state) and is added to the set of final states. This is the case where the protocol allows for a message to be uttered at a certain state, but the agent policy does not. All conversations up to this state should be accepted by M_{conf}, but it will have to be turned into a final state as the conversation can not progress any further.

Before checking for conformance and interoperability, we need to introduce the concept of a *complete* automaton. An automaton is *complete* if for all the common states in the agent's policy and the protocol specification there is at least one message that will lead to an alive state (state leading to a final state). If that message (from the agent policy) is substituted in the specification policy it is still possible to reach an alive state in the specification automaton.

In terms of *conformance* an agent is labelled conformant to a protocol specification if the automaton, whose construction is described above, is *complete*, i.e., a message will always be uttered, when one is expected) and it should accept the union of the two languages (the agent's $\mathcal{L}(P^{ag}_{lang})$ and the protocol's specification $\mathcal{L}(P_{spec})$).

Their approach works using the same concept as Singh in that they want to turn the states resulting from moves that can prevent the agents from terminating the protocol into trap states (states that are not final and the agent can only transition to the same state). However, what they are really interested in is to show that **conformant agents will also be interoperable**. According to the authors, for every common conversation (viewed as a *path* in the Finite State Automaton) t', there is another one t'' such that the resulting conversation $t = t'\mathrm{t}''$ (effectively the path that results from joining the two initial paths) is in the intersection of the two agent policy languages.

This approach entails *interoperability* by design without imposing restrictions on the conversations allowed, i.e., agents need not alternate or an action can be used more than once in a dialog, unlike the approach in Section 6.2.1. On the other hand, it links interoperability with conformance, while the approach in Section 6.2.3 does not (in fact, they discuss conformance without making any reference to the concept of interoperability).

In [10], the authors look at conformance verification of logic-based agents whose policies are specified in a Prolog-like language (in this case DyLOG) and distinguish between three degrees of conformance as shown below. The problem of verifying conformance is treated (for the first two degrees) as a inclusion problem between the set of conversations that can be generated from the agent's specification in DyLOG given an initial state of beliefs for the agent (Σ_0) (for the first case) or the set of conversations independent of the initial state of agent beliefs and the language generated by the AUML specification of the protocol (second case). In the third degree we want the agent's conformance to be proved independent of the speech acts semantics and their implementation (e.g. an *inform* message might mean that once an agent knows about a fact, it informs the others immediately or it does that only if its belief base contains the proposition that the other agent does not know about that fact; we would like to be able to prove conformance no matter what the semantics of the *inform* message are for the agent).

In this context, the following degrees of conformance are defined (higher versions imply the lower level ones):

a. an agent is *conformant* if any conversation generated by the policy language of the agent on the basis of an initial state of beliefs Σ_0 (equivalent to the initial state in our approach) is in the set of languages generated by the grammar that the AUML specification of the protocol is transformed in.

b. an agent is *strongly conformant* if any conversation generated by the policy specification of the agent independent of its belief base is also generated by the language generated by the grammar of the AUML specification of the protocol (definition a. is obviously satisfied, so a strongly conformant agent is also conformant).

c. an agent is *protocol conformant* if the language generated by the agent policy specification is a subset of the language generated by the AUML protocol specification (regardless of how the *inform* message, for example, is realised). Again, due to the definition of protocol conformance implies strong conformance and conformance.

The problem of verifying conformance is treated (for the first two degrees) as an inclusion problem. In the case of conformance, it will be between the set of conversations generated from the agent's DyLOG specification and the protocol's AUML specification, given Σ_0. In the second one we check the protocol specifications against the agent conversations independent of the initial state of the agent beliefs Σ_0. In the protocol conformance case, the agent's conformance has to be proved independent of the speech acts semantics and their implementation (e.g. an *inform* message might mean that once an agent knows about a fact, it informs the others immediately or it does that only if its knowledge tells it that the other agent does not know about this fact).

Finally, in [11], the authors talk about open systems that can be realised by new components been dynamically added to the system. Before that, however, we need to perform some sort of reasoning regarding the behaviour of the new component once inserted into the system. Finally, they make the analogy between agents and Multi-Agent Systems on one hand and web services and choreographies on the other. Agents and web services form the local view of the interaction, they are the component units. Multi-Agent Systems and choreographies are about the global viewpoint, specifying how the building blocks of the interactions should behave. A variety of representations exist for each viewpoint and the key question that both have to answer is the following: *given a choreography (MAS) can a web service (agent) join the system*? The decision will be based on whether the agent (or the web service) can participate in conversations with the other member agents or services; if it is capable of doing that, then membership is granted.

The authors also consider *bisimulation* [111] as a means for testing *conformance* of an agent wishing to interact with other agents on the basis of a choreography and the agent's policy. In [8], they conclude that this cannot be treated as a test of whether the conversations generated by the policy are a subset of what the choreography allows (as it could be the case that the agent is able to handle more messages than the choreography prescribes) or a test of whether the automata of the role as specified by the choreography and the agent's policy are *bisimilar* (as the presence of extra messages or messages with different names will generate a negative result). On the basis of these observations, they require that the bisimulation restriction is replaced with another simulation relation (*conformant simulation*) that treats incoming and outgoing messages in different ways. This captures the intuition that the policy should be able to handle all incoming messages according to the choreography and utter at least one of the messages that the choreography expects the role to utter in any given situation over the course of the protocol. Thus, it will correspond to a *Competent under Adversity* agent in our approach.

The conformant simulation relation is defined as follows: Assuming two FSA's A_1 and A_2, we say that A_1 is a conformant simulation of A_2 and write it as $A_1 < A_2$, *iff* there is a binary relation $\mathcal{R}$ between A_1 and A_2 such that (we consider the case that A_1 is the agent's private policy and A_2 is the specification of the role in the protocol that we compare the policy against):

a. $A_1.s_0 \mathcal{R} A_2.s_0$, i.e., the two initial states of the automata should be linked by the relationship $\mathcal{R}$);

b. for every outgoing message $m! \in A_1.L$ and for every state $s_i \in A_1.S$, for every $s_j \in A_2.S$ such that $s_i \mathcal{R} s_j$ and $(s_i, m!, s_{i+1}) \in A_1.T$, then there is a state $s_{j+1} \in A_2.S$ such that $(s_j, m!, s_{j+1}) \in A_2.T$ and $s_{i+1} \mathcal{R} s_{j+1}$. This means that for every outgoing message predicted by the agent's policy that is related to the role specification via $\mathcal{R}$ there is a corresponding state in the policy's automaton that is also related to the initial state of the transition. The same transition exists in the role automaton and the end states are related through $\mathcal{R}$ - this ensures that the outgoing messages which are also prescribed by the role specification would be legal;

c. for every incoming message $m? \in A_2.L$ and for every state $s_j \in A_2.S$, for every $s_i \in A_1.S$ such that $s_i \mathcal{R} s_j$ and $(s_j, m?, s_{j+1}) \in A_2.T$, then there is a state $s_{i+1} \in A_1.S$ such that $(s_i, m?, s_{i+1}) \in A_1.T$ and $s_{i+1} \mathcal{R} s_{j+1}$. This means that for every incoming message that the role specification accounts for and for all states from the agent's policy automaton that are linked to the initial state of the incoming message in the role specification, the final states of the corresponding transitions should also be linked via the relation $\mathcal{R}$.

6.2.2.1 Comparison

In [7] the authors are concerned with *conformance* interpreted as *Competent under Co-operation*. The agents are expected to fully respect the rules of the protocol and not deviate from the specified interactions. This is achieved by comparing the agent's policy language and the protocol specification against a specially constructed specification that takes both the protocol and the policy into account. They require that the agent's policy should be able to deal with all incoming messages and plan a response for at least one of them.

They make the following assumptions:

- protocols with concurrent operations are not supported - this is because the approach relies on finite state automata that do not accommodate concurrency;
- the conversation will be between two agents.

The completeness of the automaton (for every state with outgoing messages there is at least one message that leads to a path with a terminating state) they produce from the protocol specification language and the agent policy's language guarantees the property of *competence*. At least the agent is *Competent under Co-operation* due to the process of constructing the protocol automaton). This is the case as no matter what state the agents are in, a path taking them to a terminating state can always be reached. In this case, we only know that the agent is at least *Competent under Co-operation* as the other participant might select a move for which the agent in consideration has no response.

In [10], we have the following associations between the degrees of conformance specified by the authors and the ones in our approach:

- a *conformant* agent would be equivalent to an *Competent under Adversity* agent in our approach, as all generated conversations (on the basis of the agent's selection process) will be in the set of conversations that the protocol allows;
- a *strongly conformant* agent can be either *Competent under Adversity* as now there is no assumption on Σ_0 or even *fully competent* according to whether it can realise all actions or part of them;
- a *protocol conformant* agent implies a *fully competent* agent if the agent policy can generate the full protocol specification or a *Competent under Adversity* if that is not the case.

Furthermore, in this approach, the Initial Beliefs set Σ_0 can be linked to the initial state of the game. However, the higher degrees of conformance that relate to the agent's internal policies, cannot be represented in our approach.

In [11], the addition of an agent (web service) to the system (choreography) will only take place on the basis of respecting the rules of the society - this is the same idea behind our motivation in [105]. Furthermore, a notion similar to the notion of *agent skills* is used in [9] where a web service is considered to be a component available over the web that has several *interaction capabilities* that it makes available through its interface. Finally, the notion of *conformance* that they use is equivalent to the notion of *competence* we are using.

6.2.3 Endriss et al.

Endriss and colleagues in [36–38] are looking at the problem of conformance from the viewpoint of Multi-Agent Systems. They look at agent communication protocols as specifying *rules of encounter* and been designed independently of the agent policies. If we want to truly verify that an agent follows a protocol, we would need to have access to the internal mental states of the agent, which is not possible. This means that we need to use the agent's *observable behaviour* in order to do the verification.

The protocols that the authors look at are protocols that can be represented by deterministic finite state automata where the next state depends solely on the previous one (*shallowness*), mainly to avoid concurrency. The representation of the moves in general is of the form *act(Utterer,Receiver,DialogueID,Subject,Time).*

Further to shallowness, the following constraints apply to the protocols:

1. at least one rule has the special performative $START$ that signals the beginning of the interaction and $START$ can never occur on the right-hand side of a rule. This means that we cannot start nested executions of the protocol and the protocol specification does not cover sub-protocols;

2. any move - other than $STOP$ that signals the end of the dialogue - that occurs on the right-hand side also occurs on the left-hand side of another rule. This means that there are no dead states; no matter what state we are in, we can always make a move;

3. it is not possible to perform two different moves in the same dialogue at the same time; formally $tell(X,Y,S_1,T,D) \wedge tell(X,Y,S_2, T,D) \wedge S_1 \neq S_2 \Rightarrow \perp$. This means that no concurrency is allowed;

4. for every rule in the interaction other than $START$ and $STOP$, if on the left-hand side X is the utterer and Y is the receiver on the right-hand side Y should be the utterer and X the receiver. This means that all messages are used;

5. all dialogue moves on the left-hand side, i.e., the triggers of the dialogue moves, should be distinct from one another (so that non-deterministic automata are excluded).

In terms of conformance levels, a distinction is made between:

- *weak conformance*; the agent should never utter any dialogue move that is not expected for the state of the dialogue it is in. The protocol, however, might not allow the agent to not make a move and prescribe that the agent will always have to choose a move to make;

- *exhaustive conformance*; the agent should be *weakly conformant* and it should utter at least one legal move that is expected for the state of the dialogue it is in.
- *robust conformance*; the agent is *exhaustively conformant* and if the other agent utters any move that is not expected in the state of the dialogue it is in, it should react by uttering a special dialogue move (e.g. *not-understood*).

They, also, identify the decomposition of a protocol into *role skeletons*, i.e., the agent's view of the protocol, although the discussion is by example.

The authors do not deal with the issue of *interoperability* as it would involve agents revealing their private strategy.

6.2.3.1 Comparison

In [36–38] the authors look at *shallow* protocols, i.e., protocols in which the next state depends only on the previous one. We make no such assumption and our approach can accommodate any type of protocol as long as it is represented by an LTS.

Furthermore, they impose a set of rules that allows no concurrency and discards non-deterministic automata. We make no such assumptions; concurrent moves are accommodated by the use of *Event Calculus* for the representation of the game state and non-deterministic automata can be represented as a sequence of different *valid* moves.

Regarding conformance levels, the following equivalence exists between their approach and ours:

- *weak conformance* would correspond to *Competent under Adversity* or *Competent under Co-operation* in our approach, as long as the agent does not make a move that is not valid. In the case that it chooses not to do anything (or make an invalid move) there is no comparison with our competence levels.
- *exhaustive conformance* would correspond to *Competent under Adversity* or *Fully Competent* as it will always make a valid move.
- *robust conformance* would correspond to either a *Competent under Adversity* or *Fully Competent* agent depending on whether it can

make all protocol prescribed moves or not. This applies as long as the *not-understood* reply to invalid moves is part of the protocol (otherwise there is no comparison).

6.2.4 Alberti et al.

Alberti and colleagues in [3] observe the similarities between Multi-Agent Systems and Service Oriented Computing in that they both require the co-operation of autonomous and independent components (same observation as the one made by Baldoni). These components are developed independently of one another and they have to work together according to rules specified in the protocol (in the case of Multi-Agent Systems) or the choreography (in the case of web services composition).

In order for this approach to work, we need to ensure that, if combined, the agents (web services) will be able to work together and produce meaningful output. This is achieved by checking the interaction capabilities of the agents/web services with reference to the protocol/choreography that they want to join. The authors make use of a formal language for defining multi-agent protocols specified in [2, 4] and the proof-procedure *g-SCIFF* [1] to deal with protocol properties.

In this context, the *Abductive Logic Webservice Specification* ($A^{\iota}LoWS$) framework represents the specifications of both the choreography and the web service as abductive logic programs [50] with happened events (**H**) and expectations (**E**) as the abducibles. The idea is that if the specifications of the web service and the choreography are put together, we should be able to work out the set of all possible interactions between the web service and the choreography ($\mathbf{HAP}^{\star}$) on the basis of the following assumptions:

- any message that the web service is expected to send, it will actually send it;
- any message that the choreography expects to be sent from other web services (not the one we are checking for conformance), will actually be sent as well.

On the basis of that set, we can distinguish between two types of conformance:

1. *feeble conformance* - in this case the events in the set $\mathbf{HAP}^{\star}$ will be such that put together with the expectations and the knowledge base of both the choreography and the web service will model the goal and the integrity constraints of the two components. Furthermore, the happened events along with the expectations will model every expectation that the web service has as well as any expectation that the choreography has. This method of conformance ensures that all messages from the web service will be expected by the choreography, but does not cover the ones that the choreography is not prescribing.

2. *strong conformance* - for that, the interaction will have to be feeble conformant, but this time all the events should be expected by both the web service and the choreography.

This work is inspired by the work of Baldoni in that they look at the problem of conformance as a problem of comparing two formal specifications; finite state automata in the case of Baldoni and abductive logic programs in the case of $A^{\iota}LoWS$.

6.2.4.1 Comparison

The model of Alberti et al. in [3] reflects the social context of our model in terms of the requirement for adding a web service to a choreography. It will need to meet the expectations, i.e., be able to understand and utter a set of messages pertinent to the role it wants to occupy in the choreography.

They describe two types of conformance, which correspond to our approach as follows:

- *feeble conformance* could correspond to a *Fully Competent*, *Competent under Adversity* or *Competent under Co-operation* agent, as we know nothing about whether the agent can realise all messages or whether at any point it depends on the other web services to send it a specific message, so that the choreography can progress. This applies in the case that no invalid moves are made, as this would render the comparison impossible.

- *strong conformance* could correspond, again, to a *Fully Competent*, *Competent under Adversity* or *Competent under Co-operation*, as we do not have enough information to classify it in one of the categories. In this case no invalid moves are allowed by the definition.

6.2.5 Giordano et al.

Another popular approach for representing protocols is that of temporal logic [34, 42, 45]. Giordano et al. are using this representation to reason about *agent's interoperability* [43], i.e., they maintain that if a number of agents A_1, A_2,$\cdots$,A_k are proved interoperable then they should be able to produce traces that are foreseen by the protocol. Their notion of agent interoperability entails *weak conformance* - as described by Endriss in Section 6.2.3 - at least. They make use of temporal logic operators like $\bigcirc$ for the *next* operator and $\Diamond$ for the eventually one.

A protocol is described by the following components:

- action laws (AL_{ag}); these represent the effects of execution of actions on the state.
- causal laws (CL_{ag}); these represent dependencies between fluents.
- precondition laws (PL_{ag}); these represent conditions about when an action can run.

They, also, define an initial state that provides the initial values for all fluents.

The criterion which is used for deciding whether the protocol run is a good one or not is whether all commitments - used in the same way as in Section 6.2.1 - are fulfilled or not. The protocol itself is composed by the constituent protocols via synchronous composition and the benefit of using temporal logic (more specifically Dynamic Linear Time Temporal Logic) is that infinite protocols can be represented. In fact, all protocols are considered to run infinitely by setting the agent to perform a special operation *noop* that has no effect on the running of the protocol.

Every agent participating in the protocol will have a *choice* function, through which it will be making the choice of which move to select at any given state. The notion of interoperability used is the same as in the other approaches - one of the participating agents should be able to send a message and another one should be able to receive it.

The notion of *conformance* is built on top of that of *interoperability.*

The agents will have to be interoperable **and** the runs that the protocol composition will produce should all be runs allowed by the protocol P.

Finally, for an agent a to be conformant with a protocol P the following three requirements have to be met:

1. the agent should be *interoperable* with the other agents that take part in the protocol, i.e., no deadlocks should be possible, no situation in which the agent supposed to be making a move cannot make one;

2. *correctness* - any runs produced are actual runs of the protocol. In other words, if the agent can send messages that are not allowed by the protocol, then these messages should not be sent as they will create paths outside of the protocol (even if another agent happens to be able to understand and receive the message). This means that an agent can select any message as long as it is supported by the protocol (consistent with the idea that the agent needs not be able to entertain all outgoing messages, but at least one of them).

3. *completeness* - any message that an agent chooses to send and is allowed by the protocol, the agent(s) that communicate with should be able to receive it. In other words, the agent should be able to do at least all the receptions that the protocol prescribes (doing more is not a problem, as if the agents are interoperable, no message not supported by the protocol will be uttered). This is consistent with the idea that the agent can be able to receive more messages than the protocol allows for. If the other agents participating in the protocol are interoperable, there will not be a case where a message not expected will be uttered.

The test of whether the agents are interoperable or not is based on moving from a Büchi automaton to a finite automaton that is checked against the existence of at least one alive state, i.e., a state that lies on a path to a final state. That state also belongs to the original automaton $\mathcal{M}$ and the cut-down version has at least one outgoing message that is a valid choice according to the protocol. Furthermore, the tests for correctness and completeness will happen in a similar way but this time looking at two different automata. The first one is the protocol where the participants are the role specifications. The second one is the automaton where the i^{th} agent, whose properties we are checking, has been replaced by the specification of the agent itself rather than its role specification.

The protocol itself can be composed from a number of protocols via synchronous composition.

6.2.5.1 Comparison

The benefit of this approach is that it can represent protocols with infinite (non-terminating) paths. In fact, all protocols are considered to run infinitely by setting the agent to perform a special operation *noop* at the final state of the protocol. This is something that can easily be replicated in our approach by introducing the *noop* at the terminating states of the protocol, although it will not make any difference with regards to the agent's competency classification or its ability to join a society.

The notion of interoperability used is the same as in the other approaches. At each state, at least one agent can send and at least one can receive. This would correspond to a *Competent under Adversity* (or even *fully competent*) agent in our approach.

6.3 Protocol Decomposition

6.3.1 Desai et al.

In [30], Desai et al. identify the dangers of moving from the global view of a choreography (or protocol) to a local view of a single role (or agent) in either web service or multi-agent systems applications. This is important as the shift of viewpoint and the respective limitations on what the web service (or agent) can observe might mean that in the isolated agent view, there might be not enough information to implement their role specification in the choreography (or protocol). As an example, they provide the rule $\alpha \Rightarrow \beta$ where α and β are known to roles ρ_1 and ρ_2 but not to role ρ_3. If ρ_3 is supposed to send message β, then it will not be possible to send the message as it will not have knowledge of α. As a result, certain constraints need to be put in place so that this will not happen.

These constraints are as follows (every rule will be of the form $\alpha \Rightarrow \beta$ with α been the antecedent and β been the consequent):

- for any rule other than the beginning of the protocol, every preposition on the consequent of the rule should be part of another rule's antecedent (so that no rule can be used without been asserted before);

- every proposition in any rule's head should be reachable from the beginning of the protocol (so that it can be used as an enabler for a message exchange without deadlocks);

- for every rule of the protocol and every proposition α in the body of the rule, there should be a proposition β in the head such that if a role ρ initiates β, then it should send or receive α as well. In other words, the occurrence of β would mean that the same role is able to observe α as well.

Because of these properties, there will be no missing information for any of the roles (since each time a role will have to decide whether to send a message or not, it will be able to verify all the information it needs to make the decision). As a result, all protocols will be *enactable*, i.e., there will be no case where any of the protocol roles will have to make a decision about which path to take and not know the truth value of any of the conditions.

Furthermore, since they look at protocols as distributed entities and as a composition of roles, they provide an algorithm for deriving a *role skeleton*, i.e., the local view of the interaction that a role will have of the protocol including its own message exchanges. As their description of protocols is based on commitments, the role will need to know the messages it can send and receive, as well as any facts that enable them and lead to the creation (or discharge) of commitments. The main idea in the algorithm for working out the role skeleton for a certain role is that if the role does not have knowledge of the immediate proposition needed to make a decision as to how to proceed, it should be possible to backtrack and find another one that leads with certainty to the one been examined, i.e. if the role needs to know α but it does not, then the role should go back in history and find β so that the role knows it and $\beta \rightarrow \alpha$.

The process for deriving the skeleton of a role in this process is as follows:

- find all messages that role ρ is sending or receiving and save them in a set $\mathcal{S}$;

- the rules for sending a message $m \in \mathcal{S}$ would involve checking if a number of propositions α hold or not. This means that the role in question should be able to observe these propositions (e.g. if a shipping agent cannot observe - or be notified - that payment has been made it cannot ship the goods to the customer). As a result, we have to look for another proposition α' that the role knows about and its observance will definitely lead to occurrence of α. This would involve looking at all the protocol rules that generate the proposition α and looking at what the role in question knows that, if added to the knowledge base, it will guarantee occurrence of the proposition α.

6.3.1.1 Comparison

The algorithm by Desai et al. works as the protocol is supposed to be *enactable*. However, in the case it is not there are no steps in the algorithm to remedy the situation. Furthermore, it does not take into account any branching structure of the resulting protocol and just goes back until a suitable match is found, resembling a $\tau^{\star}\alpha$ bisimulation process. We do not assume that and provide algorithms for repairing the protocol (see Section 4.3) if that is not the case. The algorithm they propose does take into account the branching structure of the protocol. This is because when looking for the proposition β if there is a branch that knowledge of β could lead to α or γ, it would not meet our requirements. We require that knowledge of β leads with certainty to knowledge of α and it would not be possible to conclude that in the previous case.

6.3.2 Bouaziz

Bouaziz [18] uses XML and XSD schemas to describe a protocol ontology and views role as a component that can be fully specified by the *Role Profile* and *Role Behaviour* elements, as specified in [17].

A role is represented in a meta-model by its *profile* and *behaviour*. The behaviour comprises of the *allowed* actions it can assume. In terms of describing an action, we will need to specify its *type*, *data field*, a *set of input events* and the *output events*.

In this model, an action will be executed only if an event occurs. Moreover, the new state of the role will be determined by looking at the relevant *transition* in the meta-model. Bouaziz accommodates *conditional transitions*, i.e., transitions with different effects depending on certain conditions.

In order to provide a full description of a role in the form of an XML document, the following process is used:

1 find all actions from the protocol that involve role R - either incoming or outgoing messages;

2 for every action from the set created in step 1:

   (a) find the current node in the role XML document and create a new node with a new element;

   (b) if the action activates the initial state in the role schema, then this element becomes the initial element.

   (c) if the action activates the final state in the role schema, then this element becomes the final element;

   (d) append the node just created to the current node;

   (e) find all actions that succeed action a;

   (f) while the set created in the previous step is not empty

      i. fetch the next element a';

      ii. find the current node and create a new node that contains a transition from a to a';

      iii. append that node as a child node to the current one;

   (g) create a new node in the role schema with the role states;

   (h) find the role state from the protocol ontology that activates action a;

   (i) create a new state node with the information and append it to the current node.

First, the actions relating to the role in question are compiled. The authors check to see whether each action from the set containing the role actions activate the initial (final) state, so that they can be placed as the top (bottom) element in the role XML description.

6.3.2.1 Comparison

Step 2b of the algorithm for etxracting the role skeleton is equivalent to our bisimulation algorithm setting as initial state of the new automaton the equivalent state of the initial state of the original automaton (same for the final state). Also, the role schema will contain actions that the role in question is not directly involved in as in step 2e we are just selecting all actions succeeding action a from the protocol ontology, rather than the set of actions that the role is involved in. Only if the protocol is not enactable, additional knowledge will have to be inserted.

6.3.3 Blanc and Haumerlain

Blanc and Haumerlain [99] tackle the issue of generating a role skeleton for the purposes of separating the participatory from the strategic part of the agent. In their approach, a *moderator* agent will regulate all interactions between the agents engaged in the course of the protocol. They raise the issue of the agent been overloaded with big protocols if all the information is provided, and suggest the separation of knowledge in two different aspects. These would be the *strategic* aspect which is generated by the agent itself and consists of generating a strategy for the protocol (e.g. in an auction how should the agent bid) and the *participation* aspect that is about the agent actually participating in the protocol. The *Participation* component handles the protocol conversation, so that the agent's strategy can actually be realised. This is the aspect that the moderator agent will be seeing and interacting with as it will be receiving its utterances.

They define a protocol in terms of the following components:

- variables making up the conversation objective or characterising its state with respect to the objective;
- roles the agents can assume in the protocol;
- types of intervention the agents can use in a conversation ;
- initial state of the conversation;
- the final state of the conversation;
- casting constraints - these are the requirements that the agent should fulfill to take on the role in that protocol;
- behaviour constraints that specify when an agent can carry out an intervention.

The authors appoint a *moderator* agent to be an arbitrator for a conversation, once it begins. This means that every protocol run will need a new instance of the moderator agent that will be overlooking the conversation and making sure that all moves are in accordance with the rules of the protocol. Effectively, the moderator acts as an umpire making sure that actions taken would follow the rules of the protocol. Furthermore, it will decide whether a conversation has ended or not, thus terminating the protocol.

In terms of the actual protocol run, each agent is represented by a *participation* component. This component will have the responsibility of choosing amongst the protocol rules, those that implement best the agent's strategy. Every participation in a protocol will require a new instance of the participation component. The authors also look at the issues behind the agent's selection - why the agent would choose to participate in one conversation and not another.

This means that in order to study the behaviour of the agent, we need to look at the participation component. It is represented as a transition system and every time a choice is made, the result is sent back from the moderator.

The task in hand, now, is to extract the rules relevant to the strategy of the agent from the protocol rules. The first step will be to get anything related to the role from the protocol rules (these are messages that the role in question is either the sender or amongst the recipients). However, as the participation component will be implementing a particular strategy set by the strategy component, some rules although pertinent to the role should not be included in it (e.g. if a purchase protocol gives the option of paying using either Visa or Mastercard and the strategy of the agent says that Visa should always be preferred, then there is no reason for the participation component to know about the Mastercard option - strategy dictates that it will never be uttered).

The protocol rules are defined as a *Petri Net* [76]; in order to retrieve the rules pertinent to the role, we replace any actions in which the role is not involved with ϵ. The idea is that every state in the Petri net will be characterised by a marking, i.e., the number of tokens on each place of the Petri net. This way we do not look to places and transitions as the building blocks of the net, but to the changes they bring to token distribution.

The following steps are, then, applied to the resulting Petri Net for producing the final role transition system:

- the initial markings will make up the initial state and the transition relation is an empty set ($\varnothing$);
- apply the algorithm *Graphe* with the first argument been an empty list and the second one been the initial state.

The *Graphe* algorithm works as follows:

1 if the second argument M is equal either to the first item of the list representing the first argument (as in this case we will have the formation of an ϵ-loop) or it has already been looked at ($M_1 = \{\mu_1, \mu_2, \cdots, \mu_n\}$ and $M = \mu_1$ or $M \in M_1$), then unify M with the corresponding M_1 (as we use the Petri Net's markings as the criterion for deciding on the states, if the state in question has the same markings, then it is the same state) and exit;

2 if the state represents a dead transition, i.e., a transition that takes you nowhere, then we are ignoring it and exiting the algorithm. Even if the participation component selected this move, no change is going to happen to the course of the protocol;

3 in any other case find all t such that $M \xrightarrow{t} M'$

   (a) for each t identified in step 3

      i. if the label of the marking is not ϵ, $\lambda(t) \neq \epsilon$, i.e., it is pertinent to the role we are looking at, then add the resulting marking as a new state and add the transition to the transitions set. Then, go back to the top and re-run the *Graphe* algorithm;

      ii. if that is not the case, then add M to the list of states reached by an ϵ transition and re-run the *Graphe* algorithm with M' as the second argument and the first argument will comprise of all the states previously making the first argument with the addition of M.

6.3.3.1 Comparison

Blank and Haumerlain in [99] produce a role skeleton that might not include all the actions pertinent to the role. This is the case as the *strategic* aspect of the agent might discard some actions not fitting the agent's strategy. If, for example, the agent's role can make a payment using Paypal or Visa and its strategy dictates a preference for card payments, then the action of paying by Paypal will not be included in the role skeleton.

They represent actions not involving the role (after the initial filtering by the *strategic* aspect) with ϵ. This is equivalent to the τ actions in our framework.

In the Graphe algorithm they use markings as the equivalence criterion. If two states have the same markings, then they are the same. It is a similar concept to bisimulation, in which we demand equivalence over a relation $\mathcal{R}$.

Finally, their method seems to be similar to $\tau^\star\alpha$ bisimulation. They are merging all ϵ-transitions, as they bring no change to the markings of a state.

6.3.4 Giordano et al.

Giordano et al. [43] consider the representation of a local view (or role skeleton), as they look at the alphabet of each agent (Σ_i) separately. They are specifically interested in the actions that agent i can understand (send or receive). Any other action taken in the protocol will have a local equivalent that might be the empty action if the agent in question is not involved in it, either as a sender or a receiver. Also, the way that the local view of the agent is constructed is essentially by the use of $\tau^\star\alpha$ bisimulation, as any actions not relevant to the agent are discarded. This leads to problems, especially for protocols with a branching structure as it is not taken at all into consideration (see Section 4.2.3).

6.4 Summary of Competence Checking and Role Skeleton approaches

Table 6.1 summarises the approaches mentioned in Sections 6.2 and 6.3 regarding the concepts of *competence* viewed in the domains of *Multi-Agent Systems* and *Web Services Choreography and Orchestration* in terms of the approach used for representing the protocol as well as for its support for *protocol decomposition*:

Name of the approach	Protocol Representation Method	Different Degrees of Competence	Role Skeleton
Singh et al.	Transition Systems & Temporal Logic	✓	✓
Baldoni et al.	Finite State Automata	✓	✗
Endriss et al.	Deterministic Finite State Automata	✓	✓
Alberti et al.	Abductive Logic Programming	✓	✗
Giordano et al.	Temporal Logic	✓	✓

Table 6.1: Comparison of different approaches to competence

As we can see, a number of ways have been used to represent protocols accommodating all different types, i.e., finite and infinite protocols. Furthermore, all approaches introduce a way of distinguishing between different types of agents on the basis of their ability to cope with the protocol. The term *competence* is not used in any of the works; instead the authors use terms such as *conformance*, *interoperability* and *coverage*. In the papers, however, they are used in the same way to look into what part of the protocol the agent can cover on the basis of the messages it can exchange. A subset of the approaches, also, use information from the agent's strategy in order to ensure that any additional moves that might be present in their strategy will not cause problems and will not be selected instead of a move that the protocol would allow. Finally, a few identify the need for a local view of the protocol rules by the agent, a *role skeleton* as it is called.

6.5 Protocol description languages for approaches supporting competence/protocol decomposition

6.5.1 Desai et al.

The way that Desai et al. represent protocols in [30] resembles our game-based representation in Section 3.6. In Sections 5.2 and 5.3 we only looked at cases where the last move was the only constraint on the state of the game, e.g. a *reply* move follows a *request* one. However, other constraints on the state of the game (e.g. the running price of an auction) can be easily accommodated by appropriate `holds` events. We are basing the protocol representation on the concept of validity of moves so that any constraints can be expressed (either on the last move or, in general, on the state of the game). As an example, we can use as a constraint the running price on an item in an auction; if it is above a threshold, the next bigger bid wins the item.

A move will be deemed *valid* as long as the properties specified are true ($valid(M_1) \leftarrow property_1 \wedge property_2 \wedge \ldots \wedge property_n$). A set of values from the properties will identify a state. The authors express the same properties in the form of *commitments*. A move might create - or discharge - a commitment and the protocol proceeds on the basis of existing commitments. This representation can be more flexible in dealing with *exceptions*, i.e., moves that are not prescribed by the protocol but do not break the commitments. The only requirement is that no commitments are broken in the course of the interaction. On the other hand, it could be the case that although no commitment is broken, there might be other problems, e.g. insufficient resources or access privileges. If that is the case, then the protocol will freeze, or the violation could be flagged and not allowed through a commitment operation.

6.5.2 Bouaziz

The behaviour comprises of the *allowed* actions a role can assume. These correspond to the available actions component in our framework. In terms of describing an action, one needs to specify:

- its *type* (this corresponds to the name of the action in the game-based framework);
- *data field* (no direct correspondence in our approach, but it could be incorporated if we allow for the name of the action to be parameterised);
- a *set of input events* (representing the constraints to the *valid* predicate in our approach); and the
- *output events* (depending on their nature, these could be either the *effects* of the action or a set of *initiates* and *terminates* events).

In this model, an action will be executed only if an event occurs. The possible sequence of actions that can follow a role action will be determined by looking at the relevant *transition* in the meta-model. In our model, there is no direct correspondence as that would involve revealing actions which are not actions of the role in question. Bouaziz accommodates *conditional transitions*, i.e., transitions with different effects depending on certain conditions. Our model can accommodate this by adding rules to *effects* predicates. Also, if the next move depends not only on the previous one but in parameters of the message itself - e.g. the value of the bid, we can represent it by adding further constraints to the *valid* predicate.

6.5.3 Blanc and Haumerlain

They define a protocol in terms of the following components:

- variables making up the conversation goal(s) or characterising its state with respect to the goal(s) (in our approach, these are the states of the LTS);
- roles the agents can assume in the protocol (same in our approach);
- types of intervention the agents can use in a conversation (these are the actions of the agent);
- initial state of the conversation (the initial values of certain predicates captured by *initially* statements);
- the final state of the conversation (final values of predicates captured in the *terminating* conditions of the protocol);
- casting constraints - these are constraints relating to role conflict or any other constraints relating to the agent playing the role in the protocol (e.g. the auctioneer cannot be a bidder in the same auction or a bidder participating in an auction should have a minimum credit rating). Our approach does not consider role conflict, but this forms part of future work. The remaining constraints can be expressed as part of the *social qualifications* information that the agent has to provide when applying for membership to a society;

- behaviour constraints that specify when an agent can carry out an intervention. These are represented by *valid* move predicates in the game-based approach.

The authors appoint a *moderator* agent to be an arbitrator for a conversation, once it begins. This means that every protocol run will need a new instance of the moderator agent that will be overlooking the conversation and making sure that all moves are in accordance with the rules of the protocol (essentially this is the *umpire* role in [104]). Effectively, the moderator plays the role of the *valid* component in the game-based framework. Furthermore, it will decide whether a conversation has ended or not, a role assumed by the *terminating* conditions in our framework.

In order to study the behaviour of the agent, we need to look at the participation component. It is represented as a transition system and an action on behalf of it equates to a *select* statement in our framework. Moreover, the result sent back from the moderator is the equivalent of an *effects* statement.

6.5.4 Baldoni et al.

In [7], the authors restrict their attention to protocols that can be represented by Finite State Automata. These protocols can easily be described by our approach as we are using the same representation.

In this case, the alphabet Σ will correspond to actions, s_0 will be the initial state, the transition function δ will be the effects relationship, states can be represented as a sequence of valid moves and the final state(s) F will correspond to the final state component.

6.5.5 Giordano et al.

In [43], there is a correspondence between the components they use to describe a protocol and our approach:

- action laws (AL_{ag}): these represent the effects of execution of actions on the state. In our theory, we use the *effects* predicate.
- causal laws (CL_{ag}): these represent dependencies between fluents. In our model these are represented by *terminates* or *initiates* predicates.

- precondition laws (PL_{ag}): these represent conditions about when an action can run. We use them as components of *valid* predicates.
- initial state that provides the initial values for all fluents: this corresponds to our *initially* predicate.

Every agent participating in the protocol has a *choice* function, which is equivalent to the *select* statements in our approach.

Finally, there is no direct computational equivalent of translating the logical formulae used in DLTL into a program, although for finite protocols (protocols with finite runs that do terminate) their expressive power is the same as our game-based approach. These protocols can, however, be converted in a *Büchi* automaton.

6.6 Roles & Skills

A role is viewed as a restriction on what an agent (or an actor, in general) can do in a given situation, namely the *expected behaviour within a given context* [13]. As a concept, it has been exhaustively used in an number of areas in Computing: an example is the field of *Agent-Oriented Software Engineering (AOSE)* dealing with how to engineer societies of agents where access to resources and coordination between member agents are important issues. There are methodologies that focus mainly on the agent level [100, 101], describing mainly the agent setup. On the other hand, we have methodologies that try to develop an integrated approach that can be used in both the analysis and design stages of the system [20, 22, 29, 44, 49, 51, 53, 55, 56, 58, 80, 81, 83, 100, 101, 117–122]. Finally, there are methodologies that promote the use of formal tools and techniques for the design of agent organisations such as role algebra [51]. Another field where role theory has proven popular with respect to defining access to resources is in the field of *Databases*, where role-based access control policies are specified [69].

In some methodologies (e.g. Gaia [119, 122] and MaSE [20, 29, 49, 58, 117, 118]) the concepts of role (or *compound role*, where a number of roles are composed to form a more complex one) and *social position* are taken to be the same, however there are objections to this. Skarmeas [100] differentiates between the two. The former is a computational concept that relates to groupings of roles that an agent needs to assume, while the latter is a social construct that encompasses features like culture, respect and so on. It describes more the perception that other member agents have for the given agent rather than simply the role that it has to assume. The majority of the methodologies, also, do not look in depth at the notion of skill and in some of them (e.g. Prometheus [83]), there is no distinction between *what* needs to be done (roles) and *how* it is going to be done (skills or competencies).

Table 6.2 summarises the different approaches mentioned so far, in terms of their support for the concepts of role, skill and protocol as well as the definition of a role. We should note here that most of these works refer to full methodologies for developing mutli-agent systems and, as such, outside the scope of this work.

Name	Role	Skill	Protocol	Remarks
Gaia [119, 122]	✓	✓	✓	Role is seen as a 1st class entity and equates to social position. It is defined in terms of responsibilities ($\simeq$ skills), permissions, activities and protocols.
MaSE [20, 29, 49, 58, 117, 118]	✓	✓	✓	Roles are mapped to one or more system goals and are attached to agent classes . The skills are similar to the concept of simple messages in tasks and protocols are present in the form of conversations. However, skills are defined within the roles.
Skarmeas [100, 101]	✓	✓	✓	Skarmeas is mainly looking at case studies of office environments. Skills can be thought of as simple, i.e. atomic, tasks - they can not be further broken down into smaller (simpler) tasks. Protocols can be related to contracts & there is no analytical definition of role (maybe as the approach is geared towards implementation). However, the concept is present in the approach and a number of different types of roles are considered.

Name	Role	Skill	Protocol	Remarks
Kendall [53–56]	✓	✓	✓	Skills can be related to the agent's actions for achieving a goal & protocols can be linked to co-ordination. A role is described by the role model which it is made of, responsibilities, collaborator roles, external interfaces for access to resources, relationship to other role models, expertise of the agent (skill evaluation), co-ordination and collaboration as well as how fast the agent can learn about the environment and the actions required for achieving its goal ($\simeq$ skills). However, all descriptions are in natural language.
RoMAS [120, 121]	✓	✓	✗	Skills are defined as behaviours bound to the agent. The concept of role is defined in the *Z* language as a set of goals, attributes and services. The importance of protocols is recognised (in [120] it seems to be represented by use cases as there is no reference to it by name). There is no formal definition as the paper focuses on the definition of agents, role and role organisations.

Name	Role	Skill	Protocol	Remarks
SODA [80, 81]	✓	✓	✓	SODA is a full life-cycle methodology for engineering agents, agent societies and agent environments. Roles are associated with tasks. A task is defined by responsibilities, competencies ($\simeq$ skills) and resources (although at the analysis stage only the responsibilities item is defined). The protocols are defined in interaction models. Roles are mapped to agent classes at the design stage.
Prometheus [83]	✓	✓	✓	This is a full life-cycle methodology as well. Roles are defined in terms of functionalities, i.e., by the following attributes: name, short natural language descriptor, list of relevant percepts, data used, data produced and interactions with other functionalities. These are been assigned to agent classes at the architectural design state. Skills are present as agent capabilities, described by a name, input events, events produced, natural language descriptor, interactions with other capabilities and reference to data read and written by the capability. Protocols are described as interaction protocols at the Architectural Design stage.

Name	Role	Skill	Protocol	Remarks
TROPOS [22, 44]	✓	✓	✓	TROPOS is a full life-cycle methodology as well. The concept of role is covered by that of actor and the concept of capability is close to that of skill (also the individual actions from the agent's plan). Protocols can be represented by AUML Agent Interaction Diagrams in the detailed design phase [21].
ZEUS [51]	✓	✓	✓	ZEUS views an agent society as a collection of agents. The role is defined as a position, along with a set of characteristics (each of them has a set of attributes). These characteristics are: role model(s) that it belongs to ($\simeq$ protocol(s)), goals/responsibilities, tasks ($\simeq$ skills), capabilities/privileges and performance characteristics. The exchange of messages happens according to interaction protocols.

Table 6.2: Comparison of different AOSE approaches

In our research, we focus on the concept of role within *Multi-Agent Systems* applications and, more specifically, in the context of interaction protocols. In this context the messages than an agent can utter depends on the role it is assuming and the state of the protocol that it is in.

Once the role is described, we will need to look at the communicative acts of the agent within the context of the protocol. These communicative acts will determine the *competence* of the participating agents.

6.7 Summary

In this chapter, we considered different approaches to ours, concerning the concepts of *competence* and *protocol-to-role decomposition.* The focus was on assessing the agent's ability to join an artificial society on the basis of the conversations it *might* engage in given its communicative skills. Approaches in this area seem to be using the term *conformance* to describe different degrees of *competence.* There is also the requirement that *agents who are found to be conformant to a protocol, need to be able to use that effectively.* As a result, interactions between the agents are engineered in order to ensure that only conversations that do not lead to deadlocks can occur. This has the benefit of ensuring that there would be no problematic runs when the protocol executes at the expense of requiring the agent to reveal its private strategy which might not always be possible (or desirable).

On the other hand, our approach does not guarantee interoperability. It focuses on verifying that the agents *can* hold a conversation without making any claims that they *will* indeed have one. Furthermore, the idea of using the concept of bisimulation for breaking up a protocol into roles has not received much attention and was used in its simplest form ($\tau^\star\alpha$) under the assumption that the design of the protocol is correct. There are, however, a number of open issues regarding extensions of the research, most notably *subgames.* These are discussed in the following chapter, along with a short evaluation of the work and concluding remarks.

Chapter 7

Conclusions and Further Work

7.1 Introduction

This chapter briefly evaluates and summarises the research conducted in this work. It covers the game-based approach to evaluate an agent's competence to join an artificial agents society and the process followed to provide it with an enactable protocol description if it is allowed entry. Furthermore, we describe possible extensions and future directions of the concepts developed in the thesis.

7.2 Overall Evaluation

The way we use computers in applications such as those envisaged by ubiquitous computing raises a number of issues regarding how we design, specify and implement the interactions between people, devices and services. More specifically, there are important issues of how to structure the interactions and how do we provide access to the various services on offer.

We propose to organise the services around the concept of *semi-open artificial agent societies*. These are defined on the basis of the protocols they make available, which specify the rules of interaction as well as the roles participating in them. The agents, if interested in the protocol(s), apply for admittance to the society. They are admitted if they are *competent* users of the societal protocol(s) for the role(s) they apply for, either by been able to enact the full protocol or by been able to enact part of it, with or without the need for cooperation by the other agent participants.

We use the *game-based* metaphor as our conceptual framework because it is an intuitive way of describing interactions and most (if not all) protocol designers or end users are familiar with it. The representation framework resulting from the games metaphor is a game-based representation of the protocols. Every move in the protocol is treated as a move made in the context of a game with a number of constraints attached to it. This ensures it can be made at that state of the interaction, i.e., it is a *valid* move. The *effects* of the move are applied to the current state of the game to produce the new state and, unless it is a *terminating* state, the interaction continues in the same way. There is no winner or looser as in the traditional sense of a game, just the end of a business transaction.

As we look at protocols as games, we need a representation and implementation framework for the state of the game, which will integrate well with that for the protocol. In this work, we use the Event and Situation Calculi both as a representation framework and an implementation framework to describe the evolution of the state.

We should note the seamless integration between the different implementation frameworks for the protocol (game) and state (Event & Situation Calculi). We can keep the representation of the protocol the same, while changing the representation of the state from *Event Calculus* to *Situation Calculus* (or destructive assignment or commitments or any other state representation). This makes the approach modular and any other implementation framework for the evolution of the state can be chosen and used instead. All that is required to integrate it with the game-based representation of protocols is to rewrite the *holds* events so that they use the new state representation. In [102] the update of the state is done by *destructive assignment*, i.e., by *assert* and *retract* clauses. We could have followed this approach, explicitly adding (removing) information about what holds in the game, but that would have rendered our approach non-declarative.

Furthermore, the choice of Event and Situation Calculi was based on our choice of seeking *executable* specifications (Normal Logic Programs that can produce executable specifications). For this reason we provided our own version of the *Event* and *Situation* Calculi specified as *Normal Logic Programs*. We did not employ any other formalism (e.g. modal or temporal logic) as we focused on using a specification which is directly executable.

In addition, the formal framework defined in Section 3.3 is *methodological*, *easily extensible* and *facilitates* the addition of extra game moves. This can be done by:

- adding the action to the available actions (A) component of the protocol definition;
- adding the moves that this action is involved in to the moves (M) component;
- specifying the states for which the moves will be valid and adding them to the valid actions (V) component;
- updating the effects relationship, i.e., specifying what state the agent finds itself in after executing the move. This is done by updating the effects (E) relationship.

Our approach does not currently deal with complex interactions in the form of *subgames*. This affects the modularity of the protocols we can run it on as each protocol has to specify the full interactions between the participants and we can not make use of smaller protocols as building blocks. We plan to address this as part of our future work.

Furthermore, as our focus is on specifications that are directly executable, we did not employ formalisms with potentially higher expressivity, e.g., modal logic. However, the employed formalism is proven sufficient for the protocols we are addressing as well as for the high-level view of interactions that we are employing.

The two representations that we use for protocols, i.e., the game-based and the LTS one are strongly related and share the key concepts of the framework, e.g., the notions about *available* and *valid* moves. In addition, for small protocols or protocols whose FSP specification can be provided, a direct conversion between the two representations is possible. This fully unifies the whole process for assessing the agent's competence and providing it with the information it needs to enact its role in the protocol. For protocols with more states, or for protocols with no FSP representation there is no tool support at the moment, making the transition from one representation to the other time-consuming. This will be dealt with in our future work, as a number of ideas is explored regarding how to convert from a logic-based representation to an LTS representation.

The concept of bisimulation is well studied in a number of different areas, e.g., modal logic, calculus of communicating systems, formal verification and as a means of checking equivalence between transition systems. A number of tools, both commercial and open source, exist that can fully automate the process. The *CADP*[1] and *mCRL2*[2] are probably the two most notable examples. As a large number of protocols in the area of E-Commerce and Multi-Agent Systems can be described using Finite State Automata or formalisms that can eventually be transformed into a transition system, bisimulation is a good approach for minimising them. It is, also, a suitable approach when treating a role as a component that has to fit within a larger system - a protocol or a choreography, and we need to make sure that substituting the protocol with the bisimulated role will produce the same overall behaviour. In addition, as we are interested in the behavioural part of the societal components, transition systems are a well established mechanism for describing it. Behavioral (bisimulation) equivalences are usually employed to perform reasoning on LTS and determine their equivalence. The approaches for deriving the *role skeleton* for an agent that were reviewed in Section 6.3 use concepts very similar to bisimulation, but most of them make no direct reference to the concept.

7.3 Evaluation of the research aims

In this section, we review and evaluate the research aims of the work.

(RA 1) The first aim of the research was to establish a framework based on the games metaphor that will represent protocols in an artificial agent society and will use social concepts in this representation, namely the concepts of role and skills.

This aim was fully achieved by the games framework described in Section 3.3. The game is represented as an LTS and one of the components of the description is R that represents the roles participating in the protocol.

We treat protocols as first-class entities in which agents will have to participate assuming one of the roles that the protocol makes

[1]`http://www.inrialpes.fr/vasy/cadp/`
[2]`http://www.mcrl2.org/mcrl2/wiki/index.php/Home`

available. The protocol is defined as a set of messages passing between the participants, where each message will be characterised by a sender, a number of recipients and a move (the content of the message). The move component will comprise a *skill* of the agent, i.e. an utterance that it can send or understand. The representation of the move could be enhanced to include various parameters relevant to the message (e.g. the price of an auctioned item on which the agent bids as well as its id) or, even, the level at which the agent possesses the skill (e.g. an agent might be able to send call-for-bids message, but not in encrypted form).

(RA 2) The second aim dealt with the provision of a computational framework for assessing the application of an agent wishing to join an artificial agent society. This is achieved by the computational version of the LTS for the representation of protocols and follows closely the framework of [102].

The protocol is represented as a `game/2` predicate in Prolog, which we initially check if it is in a terminating state or not. If it is, the game ends and the result is returned. If not, the next *valid* move is chosen, its effects are assumed and we follow the cycle again to check if that move terminated the game.

It amounts to verifying the runs that the agent can produce on the basis of its competency profile are all the possible runs the protocol prescribes (in the case of *fully competent* agents) or the runs to meet the minimum requirements set by the *Authority Agent* (for *Competent under Adversity* or *Competent under Co-operation* agents). The current framework works well for *atomic* - i.e., without subgames, games but it would be interesting to extend it to introduce a specification for *compound* games that include subgames. In this richer framework, we would be able to express constraints like label some sub-protocols as essential or some others as indifferent. It will also provide a more abstract and fine-grained way of representing protocols as well as help building a library of protocols that can be re-used and form more complex ones through the process of composition.

(RA 3) The provision of a framework for decomposing the protocol into smaller ones that contain the *minimal* amount of information pertaining to all of its roles was the next aim of the work. As the representation of the protocol is that of an LTS, we chose to use the concept of bisimulation for breaking the protocols.

Every action that the role does not "understand", i.e. it is neither the sender nor amongst the recipients of the message, is replaced by a silent (τ) action. Branching bisimulation is run on the resulting LTS and, if no τ actions remain or those that remain do not make the protocol non-enactable, the *role skeleton* LTSs are generated.

In the light of the previous aim, it would be interesting to look at the properties of *compound* protocols. As an example, if an agent is found to be competent, would the *role skeleton* that it should receive be the product (with the appropriate constraints in place) of the role skeleton of the protocols that make up the compound one? Or, the inverse question, if the agent is found to be competent in the smaller constituent protocols, can we gather that it will be competent in the compound one as well?

(RA 4) The final aim refers to ways of repairing a protocol in the case that is not enactable. In this case, we will need to give certain roles access to additional information than the one they already have - they will need to be added as recipients to certain messages that they do not know about in the current implementation of the protocol.

We present three different approaches to repairing protocols, ranging from repairing every move that relates to a state that is the originator of a silent (τ) action to repairing only selected τ moves and only if they are causing the protocol to be non-enactable. This significantly reduces the volume of additional information that the agent assuming the role receives, a very important property for protocols containing sensitive information.

Again, it would be interesting to research the case of compound protocols. If repairs are made on the constituent protocols, will the composition of the repaired protocols be the same protocol as if we took the original compound protocol and repaired it?

7.4 Thesis Summary

In this thesis, we looked at the various phases that an agent has to go through when applying for a position in an artificial agent society. This application will be driven by its need to fulfill a goal - resources which are available within the society in consideration - and its societal life cycle (if admitted) will be divided in three phases. These would be *application*, *evolution* whilst a member of the society and *exiting* the society. We do not look at the processes of how the agent evolves once in the society or how it leaves it (whether having its goal fulfilled or been expelled from it) in detail other than simply sketching how they should be processed in terms of activities. UML activity diagrams are used for this purpose.

Instead, we concentrate on the part where the agent applies for entry and provides the society's Authority Agent with its communication skills, i.e., the communicative acts that it can understand and engage in. This forms the *competency profile* of the agent, which is then incorporated It does not provide any other aspect of its personal strategy, only that it can understand certain messages and it is able of sending or receiving them. We then present an algorithm with the use of which the agent's *competency* can be assessed both in cases where we focus on changes of the global state (with the use of *Situation Calculus*) as well as in cases where concurrency is a domain requirement (with the use of *Event Calculus*). The choice of *Event Calculus* is done on the basis that it is a natural way of representing time, although versions of *Situation Calculus* that can deal with time and concurrent moves have been developed. The *Situation Calculus* and *Event Calculus* frameworks (used to describe the state of the protocol) are combined with the *game* representation of the protocol (used to present the protocol evolution) in order to describe how it evolves from its initial state to a terminating one. The evolution happens through selection of a series of *valid* moves from the moves that the protocol makes *available* by checking constraints on the current state of the game to ensure the agent can make the move. The move will, in turn, change the state of the protocol. The changes are *assumed* via *effects* rules and a check is made to see if the game has reached a *terminating* state or not. If not, the same process is followed again.

In the case studies considered in Chapter 5, we look at both acyclic and cyclic protocols, as well as cases where the protocol might involve more than a single instance of the same role (e.g. an auction with two bidders). The conditions we impose on a move for it to be *valid* in these examples refer to the last move made in the game, leaving out any additional constraints relating to other elements of the game state. We can add more constraints to a move by extending the *valid* predicate. The Authority Agent of the society will use these descriptions (as well as the initial set-up of the protocol) in order to make the decision of whether to allow entry to the applicant agent or not, on the basis of the **competence requirements** of the society. In other words, the *Authority Agent* will have a set S of paths that the applicant agent will need to be able to follow and generate the paths Q that it can assume on the basis of its *competence profile*. If $Q \subseteq P$, then the agent is accepted into the society.

Assuming the agent is accepted into the society, the next step is to consider what kind of information to provide it with. The full protocol might be too big to be treated by the agent or there could be security concerns for which we do not wish to reveal the whole protocol. Thus, the best solution is to provide the applicant agent with exactly the information that it needs to participate in the protocols prescribed by the role(s) it subscribes to. In other words, we need a new protocol that can simulate the behaviour of the original one minus the unknown actions to the agent. These would comprise the messages that the agent is not involved in, either as a sender or as a recipient.

As our initial game-based approach is resulting into an LTS for representing the different paths that the protocol execution can take, we want an LTS that simulates the original one discarding (possibly) the actions that the agent is not engaged in. Effectively, we want two LTSs that one simulates the other; thus, we employ *bisimulation* to achieve the reduction by looking at the $\tau^{\star}\alpha$ and *branching* bisimulation equivalence relations. The first one is unsuitable as it does not take into account the branching structure of the protocol. It reduces all sequences of unknown actions before a known one, without reasoning about the effects this will have, while the latter reduces only those unknown actions that do not break the branching structure. We represent the protocol in the form of an LTS, which is translated into a Prolog program for the implementation. The unknown actions are substituted by silent ones (τ) and the branching bisimulation algorithm is run on the resulting protocol. The resulting protocol should have no silent actions or those present should not affect the outcome by creating non-enactable branching structures. These are structures where the agent has a choice of taking two actions but not the knowledge to decide on what to do.

By the use of bisimulation, we can also check if a protocol decomposition into role behaviours for a protocol containing multiple role instances of the same role is correct or not. If it is, then the LTSs that we produce for those instances should be *bisimilar*, as they represent the same role, only different instances. This is assuming the protocol does not differentiate between different role instances, but prescribes the same behavior for all of them.

7.5 Further Research

7.5.1 Framework Extension

A number of possibilities exist regarding the extension of the work presented here both at the theoretical and the implementation levels - we present them in relevance with the issues mentioned in Chapter 1.

The first issue to consider concerning the game-based representation of the competence framework is that of protocol representation in the competence assessment and role allocation phases. At the moment, the protocol representation for checking whether the agent will be admitted to the society and decomposing the protocol to its role(s) are different (although they are different viewpoints of looking at the same information). A better approach would be to have a unique representation or a way of converting between the two. This is the case as some protocols (e.g. chess) are easier to describe through valid moves, as the state space is huge whereas others (e.g. netbill) can easily be described using their entire state space.

One way of doing that could be to decide on a set of properties that will uniquely characterise a state (in the chess protocol, this could be the positions of the various pieces on the chess board). Once the set of properties is decided, we use these values in the *valid* predicate to decide on the next state. The value of the next state will be the collective value of all these properties. This will give us the LTS representation needed for running the bisimulation algorithm.

If we have an FSP model of the protocol roles, another way of dealing with the same issue is to compose and convert them into a format that is accepted by the `protocol` predicate in our approach. This will be done using the steps in the *preparation* phase in Section 5.5.3.

The concept of **subgames** [104] could, also, be explored in the representation of protocols for the application phase when the agent requests access to an artificial agent society as it allows for higher modularity and granularity when describing protocols. As an example, if we look at a protocol describing a Dutch auction and the result is a tie, then the auctioneer will need to run another auction that follows the English auction format to decide the winner. These are two separate protocols that can exist independently of one another, but in this context one can be embodied in the other. This is an example of a sub-protocol that can only start at a specific point in the current protocol and has to complete before returning to the main protocol and the main protocol cannot terminate if the sub-protocol does not terminate. On the other hand, if the agent is involved in a number of auctions running at the same time independently of one another, then the moves can *interleave*. This means that they can be executed in any order and we can get in and out of the different sub-protocols at will. The termination of one of the protocols might have implications for the others, e.g. if the agent is bidding for a similar item in all running auctions, or not, e.g. if the bids are placed for different items. This change in representation will not affect the bisimulation algorithm as it can be applied to any transition system - in fact, for non-interleaving protocols, we might be able to run bisimulation separately on the protocols involved. It will, also, allow us to run multiple instances of the same protocol (with the necessary amendments in the computational representation, e.g. the addition of an *id* to be able to differentiate between different runs of the same protocol).

The work presented in this thesis could, also, be extended in new directions to cover issues that were not considered here. A question that we did not look at in this work has been the case of a conditional reply from the society to the agent. Our model needs to be augmented to cover conditional criteria on which to base the agent's admission to the society. As an example, the society might consider that an agent accepted for a role involving protocols $p_1, p_2, \ldots, p_n$ shall be able to do all moves in p_1, reach certain terminating states of p_2 and we are not concerned with what it can do for p_n. Alternatively, this issue can be resolved by on the spot training of the agent, i.e., ad hoc addition of the missing protocol rules to its knowledge base.

The second and third phase of the agent's life cycle within the society will involve activities like revising its status and (possibly) dynamic update of the *protocols* and *roles* it will be involved in. In our current approach, the assignment of roles to the agent is done in a static way at entry time. We need a way of modelling the dynamic re-assignment of roles, including dealing with any potential role conflicts that might arise. This issue has been tackled to some extent in [100], but in a centralised way with an agent acting as a database of all other member agents abilities and roles. Any request for a particular service could be redirected through that agent. This is an approach that is computationally expensive and error-prone as a failure of the database agent will result in a complete failure of the whole system.

We could, also, look into how our approach can be enriched and what is the minimum extra set of information that needs to be requested in order to ensure and enforce competence between agents when they are found in the same society. At the moment, our test is solely based on the skills of the agents, but we do not look at the element of their private strategies. As an example, we might have two agents who both have the skill to greet, but that is only invoked if the agent is greeted first by another agent. If these two agents are put together in a society, their engagement would end up in a deadlock situation in which no agent can (or wants to) make a move.

The idea of linking the skills of the agent with part of its selection strategy in order to classify it with regards to a competence level is present in [35]. However, the authors look at a specific class of protocols (*shallow*). We intend to look at this as part of further work.

Another area of improvement could be the phase where the agent gets the protocol pertinent to its role. The agent could get a *personalised* version of its role LTS rather than the generic role description. At the moment, we are using branching bisimulation on the original protocol ignoring any paths that the agent might not be able to make because of insufficient skills or because these paths are not in its individual maximal strategy. In this, after deploying the equation in Section 3.9 to compute its maximal strategy, the resulting LTS for the role could be further pruned to take that into account.

As an example, consider an agent A with the following set of skills $Ag(Skills) = \{order, acceptorder\}$ participating in the protocol in Figure 7.1.

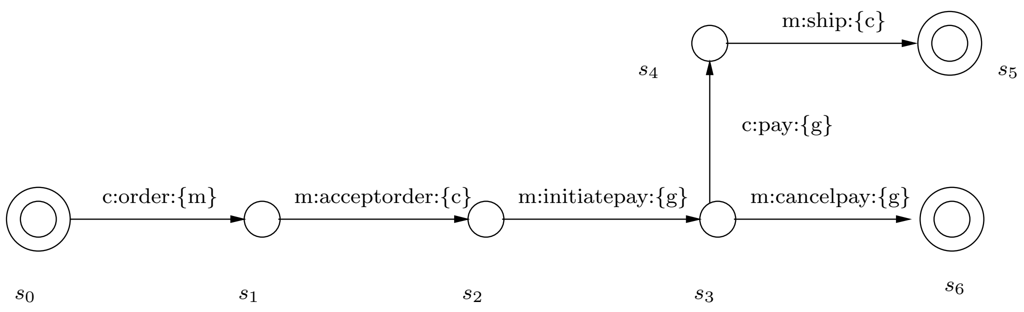


Figure 7.1: Order protocol

On the basis of the protocol rules and the agent's competencies, it is *competent under co-operation*. At the moment, the agent will receive a role protocol that contains actions *pay* and *ship*, which it cannot understand. A better approach would be to combine the protocol decomposition algorithm along with the agent's maximal strategy algorithm. This will provide it with a protocol that will contain actions and paths that it can reach ruling out the problematic ones either due to agent's incompetence or not been part of the agent's maximal strategy.

7.5.2 Technical Improvements

An issue with the Prolog representation of the process at the moment is that we need to calculate the truth value of a number of properties through `holds` and `happens` predicates repeatedly. In that process the same predicates would be called over and over again as the game progresses. The current implementation does not account for any caching or saving the information in any way and this can prove to be a performance bottleneck for big and complex protocols. An approach for tackling this issue could be to use a Prolog with abilities of caching and tabling results, like XSB-Prolog [108, 114]. We could, also, look at versions of *Event Calculus* that are designed to run with better efficiency [23].

The `protocol` predicate will need to be extended to include the structural relations between the top-level protocol and its sub-protocols. Also, the definition of a number of `initiates` and `terminates` predicates will need to change in order to account for the fact that some moves might be starting a new subgame or that moves can be made outside the subgame's context even if it is still active. Finally, all protocol-relating predicates will need to carry an id as they will be evaluated in multiple instances of protocol runs.

Bibliography

[1] Marco Alberti, Federico Chesani, Marco Gavanelli, Evelina Lamma, Paola Mello, and Paolo Torroni. Security protocols verification in abductive logic programming: A case study. In Oguz Dikenelli, Marie Pierre Gleizes, and Alessandro Ricci, editors, *ESAW*, volume 3963 of *Lecture Notes in Computer Science*, pages 106–124. Springer, 2005. ISBN 3-540-34451-9.
Cited on page(s): 124

[2] Marco Alberti, Anna Ciampolini, Marco Gavanelli, Evelina Lamma, Paola Mello, and Paolo Torroni. A social ACL semantics by deontic constraints. In Vladimír Maríк, Jörg P. Müller, and Michal Pechoucek, editors, *Multi-Agent Systems and Applications III, 3rd International Central and Eastern European Conference on Multi-Agent Systems, CEEMAS 2003, Prague, Czech Republic, June 16-18, 2003, Proceedings*, volume 2691 of *Lecture Notes in Computer Science*, pages 204–213. Springer, 2003. ISBN 3-540-40450-3.
Cited on page(s): 124

[3] Marco Alberti, Marco Gavanelli, Evelina Lamma, Federico Chesani, Paola Mello, and Marco Montali. An abductive framework for a-priori verification of web services. In Annalisa Bossi and Michael J. Maher, editors, *Proceedings of the 8th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 10-12, 2006, Venice, Italy*, pages 39–50. ACM, 2006. ISBN 1-59593-388-3.
Cited on page(s): 124, 125

[4] Marco Alberti, Marco Gavanelli, Evelina Lamma, Paola Mello, and Paolo Torroni. Specification and verification of agent interaction using social integrity constraints. In *LCMAS'03: Logic and Communication in Multi-Agent Systems*, volume 85(2) of *Electronic Notes in Theoretical Computer Science*, pages 94–116. El-

sevier, 2004.
Cited on page(s): 109, 124

[5] Alexander Artikis and Jeremy Pitt. A formal model of open agent societies. In *AGENTS '01: Proceedings of the fifth international conference on Autonomous agents*, pages 192–193. ACM, New York, NY, USA, 2001. ISBN 1-58113-326-X.
Cited on page(s): 17, 18

[6] A. Badica and C. Badica. Formalizing agent-based english auctions using finite state process algebra. *Journal of Universal Computer Science*, 14(7):1118–1135, 2008.
Cited on page(s): 96

[7] Matteo Baldoni, Cristina Baroglio, Alberto Martelli, and Viviana Patti. Verification of protocol conformance and agent interoperability. In Francesca Toni and Paolo Torroni, editors, *CLIMA VI*, volume 3900 of *Lecture Notes in Computer Science*, pages 265–283. Springer, 2005. ISBN 3-540-33996-5.
Cited on page(s): 109, 114, 120, 139

[8] Matteo Baldoni, Cristina Baroglio, Alberto Martelli, and Viviana Patti. A priori conformance verification for guaranteeing interoperability in open environments. In Asit Dan and Winfried Lamersdorf, editors, *Service-Oriented Computing - ICSOC 2006, 4th International Conference, Chicago, IL, USA, December 4-7, 2006, Proceedings*, volume 4294 of *Lecture Notes in Computer Science*, pages 339–351. Springer, 2006. ISBN 3-540-68147-7.
Cited on page(s): 119

[9] Matteo Baldoni, Cristina Baroglio, Alberto Martelli, and Viviana Patti. Reasoning about interaction protocols for customizing web service selection and composition. *Journal of Logic and Algebraic Programming*, 70(1):53–73, 2007.
Cited on page(s): 121

[10] Matteo Baldoni, Cristina Baroglio, Alberto Martelli, Viviana Patti, and Claudio Schifanella. Verifying protocol conformance for logic-based communicating agents. In João Alexandre Leite and Paolo Torroni, editors, *CLIMA V*, volume 3487 of *Lecture Notes in Computer Science*, pages 196–212. Springer, 2004. ISBN 3-540-28060-X.
Cited on page(s): 109, 117, 121

[11] Matteo Baldoni, Cristina Baroglio, Alberto Martelli, Viviana Patti, Claudio Schifanella, Laura Torasso, and Viviana Mascardi. Personalization, verification and conformance for logic-based communicating agents. In Flavio Corradini, Flavio De Paoli, Emanuela Merelli, and Andrea Omicini, editors, *WOA 2005: Dagli Oggetti agli Agenti. 6th AI*IA/TABOO Joint Workshop "From Objects to Agents": Simulation and Formal Analysis of Complex Systems, 14-16 November 2005, Camerino, MC, Italy*, pages 177–183. Pitagora Editrice Bologna, 2005. ISBN 88-371-1590-3.
Cited on page(s): 118, 121

[12] Bernhard Bauer, Jörg P. Müller, and James Odell. Agent UML: A formalism for Specifying Multiagent Interaction. In Paolo Cinancarini, Cristiano Castelfranchi, and Michael Wooldridge, editors, *Agent-Oriented Software Engineering*, pages 91–103. Springer–Verlag, Berlin, 2001.
Cited on page(s): 4

[13] B. J. Biddle. Recent developments in role theory. *Annual Review of Sociology*, 12(1):67–92, 1986.
Cited on page(s): 140

[14] Bruce J. Biddle. *Role Theory: Concepts & Research.* John Wiley & Sones, January 1979. ISBN 978-0471072157.
Cited on page(s): 20

[15] Stefan Blom, Wan Fokkink, Jan Friso Groote, Izak van Langevelde, Bert Lisser, and Jaco van de Pol. μCRL: A toolset for analysing algebraic specifications. In *CAV'01: Proceedings of the 13th International Conference on Computer Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, pages 250–254. Springer-Verlag, Paris, France, July 2001.
Cited on page(s): 92

[16] Benedikt Bollig. *Formal Models of Communicating Systems: Languages, Automata and Monadic Second-Order Logic.* Texts in Theoretical Computer Science. An EATCS series. Springer, 1st edition, October 2006. ISBN 978-3540329220.
Cited on page(s): 22

[17] Wassim Bouaziz. Une Ontologie de Protocoles pour la Coordination de Systèmes Distribués. In *Journées Francophones sur les Ontologies (JFO), Sousse, Tunisie, 18/10/07-20/10/07*, pages

231–246. Centre de Publication Universitaire, Octobre 2007.
Cited on page(s): 130

[18] Wassim Bouaziz and Eric Andonoff. Dynamic execution of coordination protocols in open and distributed multi-agent systems. In Anne Håkansson, Ngoc Thanh Nguyen, Ronald L. Hartung, Robert J. Howlett, and Lakhmi C. Jain, editors, *Agent and Multi-Agent Systems: Technologies and Applications, Third KES International Symposium, KES-AMSTA 2009, Uppsala, Sweden, June 3-5, 2009. Proceedings*, volume 5559 of *Lecture Notes in Computer Science*, pages 609–618. Springer, 2009. ISBN 978-3-642-01664-6.
Cited on page(s): 130

[19] Jeffrey M. Bradshaw. *An introduction to software agents*, pages 3–46. MIT Press, Cambridge, MA, USA, 1997. ISBN 0-262-52234-9. URL `http://portal.acm.org/citation.cfm?id=267985.267993`.
Cited on page(s): 2, 16, 17

[20] Frances M. T. Brazier, Catholijn M. Jonker, and Jan Treur. Principles of compositional multi-agent system development. In *Proceedings of the* 15th *IFIP World Computer Congress, WCC'98, Conference on Information Technology and Knowledge Systems, IT&KNOWS'98*, pages 347–360. Press, 1998.
Cited on page(s): 140, 142

[21] Paolo Bresciani, Anna Perini, Paolo Giorgini, Fausto Giunchiglia, and John Mylopoulos. A knowledge level software engineering methodology for agent oriented programming. In *AGENTS '01: Proceedings of the fifth international conference on Autonomous agents*, pages 648–655. ACM, 2001. ISBN 1-58113-326-X. URL `http://doi.acm.org/10.1145/375735.376477`.
Cited on page(s): 145

[22] Paolo Bresciani, Anna Perini, Paolo Giorgini, Fausto Giunchiglia, and John Mylopoulos. Modeling early requirements in tropos: A transformation based approach. *Lecture Notes in Computer Science*, 2222:151–168, 2002. ISSN 0302-9743.
Cited on page(s): 140, 145

[23] Luca Chittaro and Angelo Montanari. Efficient handling of context-dependency in the cached event calculus. In *Proc. of TIME'94 - International Workshop on Temporal Representation*

and Reasoning, pages 103–112, 1994.
Cited on page(s): 158

[24] Amit K. Chopra and Munindar P. Singh. Producing compliant interactions: Conformance, coverage, and interoperability. In Matteo Baldoni and Ulle Endriss, editors, *Declarative Agent Languages and Technologies IV, 4th International Workshop, DALT 2006, Hakodate, Japan, May 8, 2006, Selected, Revised and Invited Papers*, volume 4327 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2006. ISBN 3-540-68959-1.
Cited on page(s): 109, 110, 112

[25] Benjamin Cox, J. D. Tygar, and Marvin Sirbu. Netbill security and transaction protocol. In *Proceedings of the 1st conference on USENIX Workshop on Electronic Commerce - Volume 1*, volume 1, page 6. USENIX Association, Berkeley, CA, USA, July 1995. URL `http://portal.acm.org/citation.cfm?id=1267185.1267191`.
Cited on page(s): 29, 35, 89

[26] Paul Davidsson. Emergent societies of information agents. In Matthias Klusch and Larry Kerschberg, editors, *Proceedings of the 4th International Workshop on Cooperative Information Agents IV, The Future of Information Agents in Cyberspace*, volume 1860 of *Lecture Notes in Computer Science*, pages 143–153. Springer-Verlag, London, UK, July 2000. ISBN 3-540-67703-8. URL `http://portal.acm.org/citation.cfm?id=647785.735565`.
Cited on page(s): 17

[27] Paul Davidsson. Categories of artificial societies. In *Proceedings of the Second International Workshop on Engineering Societies in the Agents World II*, Lecture Notes in Computer Science, pages 1–9. Springer-Verlag, London, UK, July 2001. ISBN 3-540-43091-1. URL `http://portal.acm.org/citation.cfm?id=645380.651127`.
Cited on page(s): 3, 20, 22

[28] Paul Davidsson and Stefan Johansson. On the potential of norm-governed behavior in different categories of artificial societies. *Computational and Mathematical Organisational Theory*, 12(2-3):169–180, 2006. ISSN 1381-298X (print version), 1572-9346

(electronic version). URL `http://dx.doi.org/10.1007/s10588-006-9542-x`.
Cited on page(s): 18

[29] Scott A. DeLoach. Multiagent systems engineering: A methodology and language for designing agent systems. In *Proceedings of Agent-Oriented Information Systems '99 (AOIS '99)*, pages 45–57. Heidelberg, Germany, 14–15 June 1999 1999.
Cited on page(s): 140, 142

[30] Nirmit Desai, Ashok U. Mallya, Amit K. Chopra, and Munindar P. Singh. Interaction protocols as design abstractions for business processes. *IEEE Transactions on Software Engineering*, 31(12): 1015–1027, 2005.
Cited on page(s): 128, 136

[31] Frank Dignum and Mark Greaves. Issues in Agent Communication: An Introduction. In *Issues in Agent Communication*, pages 1–16. Springer–Verlag, London, UK, 2000. ISBN 3-540-41144-5.
Cited on page(s): 23

[32] V. Dignum, J-J. Meyer, H. Weigand, and F. Dignum. An organizational-oriented model for agent societies. In G. Lindemann, D. Moldt, M. Paolucci, and B. Yu, editors, *Proceedings of International Workshop on Regulated Agent–Based Social Systems: Theories and Applications (RASTA'02)*, pages 31–50. Hamburg, Germany: University of Hamburg, 16 July 2002.
Cited on page(s): 19

[33] J. E. Doran. The archaeology of artificial societies. In *Proceedings of the AISB'00 Symposium on Starting from Society - The Application of Social Analogies to Computational Systems*, pages 15–21. UK: AISB, Birmingham,UK, April 2000. ISBN 1 902956 13 8.
Cited on page(s): 18

[34] Ulle Endriss. Temporal logics for representing agent communication protocols. In Frank Dignum, Rogier M. van Eijk, and Roberto A. Flores, editors, *AC*, volume 3859 of *Lecture Notes in Computer Science*, pages 15–29. Springer, 2006. ISBN 978-3-540-68142-7.
Cited on page(s): 126

[35] Ulrich Endriss, Wenjin Lu, Nicolas Maudet, and Kostas Stathis. Competent agents and customising protocols. In Andrea Omicini,

Paolo Petta, and Jeremy Pitt, editors, *ESAW*, volume 3071 of *Lecture Notes in Computer Science*, pages 168–181. Springer, 2003. ISBN 3-540-22231-6.
Cited on page(s): 78, 84, 157

[36] Ulrich Endriss, Nicolas Maudet, Fariba Sadri, and Francesca Toni. Aspects of protocol conformance in inter-agent dialogue. In *AAMAS*, pages 982–983. ACM, 2003. ISBN 1-58113-683-8.
Cited on page(s): 121, 123

[37] Ulrich Endriss, Nicolas Maudet, Fariba Sadri, and Francesca Toni. Protocol conformance for logic-based agents. In Georg Gottlob and Toby Walsh, editors, *IJCAI-03, Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, Acapulco, Mexico, August 9-15, 2003*, pages 679–684. Morgan Kaufmann, 2003.
Cited on page(s): 121, 123

[38] Ulrich Endriss, Nicolas Maudet, Fariba Sadri, and Francesca Toni. Logic-based agent communication protocols. In F. Dignum, editor, *Advances in agent communication*, volume 2922, pages 91–107. Springer Verlag, 2004. ISBN 3-540-20769-4.
Cited on page(s): 121, 123

[39] Thomas Erl. *Service-Oriented Architecture (SOA): Concepts, Technology, and Design.* Prentice Hall PTR, Upper Saddle River, NJ, USA, August 12, 2005. ISBN 0131858580.
Cited on page(s): 2

[40] Kathi Fisler and Moshe Y. Vardi. Bisimulation minimization in an automata-theoretic verification framework. In Ganesh Gopalakrishnan and Phillip J. Windley, editors, *FMCAD*, volume 1522 of *Lecture Notes in Computer Science*, pages 115–132. Springer, 1998. ISBN 3-540-65191-8.
Cited on page(s): 63

[41] Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3 edition, 2003. ISBN 0321193687.
Cited on page(s): 3

[42] Dov M. Gabbay, Ian Hodkinson, and Mark Reynolds. *Temporal logic (vol. 1): mathematical foundations and computational aspects.* Oxford University Press, Inc., New York, NY, USA, 1994.

ISBN 0-19-853769-7.
Cited on page(s): 126

[43] Laura Giordano and Alberto Martelli. Verifying agents' conformance with multiparty protocols. In Michael Fisher, Fariba Sadri, and Michael Thielscher, editors, *CLIMA IX*, volume 5405 of *Lecture Notes in Computer Science*, pages 17–36. Springer, 2008. ISBN 978-3-642-02733-8.
Cited on page(s): 126, 135, 139

[44] Fausto Giunchiglia, John Mylopoulos, and Anna Perini. The tropos software development methodology: processes, models and diagrams. In *AOSE'02: Proceedings of the 3rd international conference on Agent-oriented software engineering III*, pages 162–173. Springer-Verlag, Berlin, Heidelberg, 2003. ISBN 3-540-00713-X.
Cited on page(s): 140, 145

[45] Robert Goldblatt. *Logics of time and computation*. Center for the Study of Language and Information, Stanford, CA, USA, 1987. ISBN 0-937073-12-1.
Cited on page(s): 126

[46] Vaishali Goradia, Bob Mowry, Porlin Kang, Michelle Panjwani, David Lowe, Alexander Somogyi, Patrick Magruder, Thomas Wagner, David McNeil, Catherine Yang, William Arms, Marvin Sirbu, and Doug Tygar. Netbill 1994 prototype. TR 1994-11, Information Networking Institute, Carnegie Mellon University, 1994.
Cited on page(s): 29, 35, 89

[47] Stephane Grumbach. DATALOG^{rew}: DATALOG with rewriting rules. In *Cinquièmes Journées Bases de Données Avancées (BDA), 26-28 Septembre 1989, Genève, Suisse (Informal Proceedings)*, pages 87–108. INRIA, 1989.
Cited on page(s): 23

[48] Bryan Horling and Victor Lesser. A survey of multi-agent organizational paradigms. *Knowl. Eng. Rev.*, 19(4):281–316, 2004. ISSN 0269-8889.
Cited on page(s): 19

[49] Carlos Angel Inglesias, Mercedes Garijo, and José Centeno-González. A survey of agent-oriented methodologies. In Jörg P. Müller, Munindar P. Singh, and Anand S. Rao, editors, *ATAL*,

volume 1555 of *Lecture Notes in Computer Science*, pages 317–330. Springer, 1998. ISBN 3-540-65713-4.
Cited on page(s): 140, 142

[50] A. C. Kakas, R. A. Kowalski, and F. Toni. Abductive Logic Programming. *Journal of Logic and Computation*, 2(6):719–770, 1993.
Cited on page(s): 124

[51] Anthony Karageorgos, Simon Thompson, and Nikolay Mehandjiev. Semi-automatic design of agent organisations. In *SAC '02: Proceedings of the 2002 ACM symposium on Applied computing*, pages 306–313. ACM, New York, NY, USA, 2002. ISBN 1-58113-445-2. URL `http://doi.acm.org/10.1145/508791.508853`.
Cited on page(s): 140, 145

[52] Krishna Kavi, David C. Kung, Hitesh Bhambhani, Gaurav Pancholi, and Marie Kanikarla. Extending UML for modeling and design of multi-agent, May 3–10, 2003.
Cited on page(s): 3

[53] Elizabeth A. Kendall. Partitioning goals with roles. In Serge Demeyer and Jan Bosch, editors, *ECOOP Workshops*, volume 1543 of *Lecture Notes in Computer Science*, pages 240–241. Springer, 1998. ISBN 3-540-65460-7.
Cited on page(s): 140, 143

[54] Elizabeth A. Kendall. Role model designs and implementations with aspect-oriented programming. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 353–369, 1999. ISBN 1-58113-238-7 (ACM), 0-201-48561-3 (Addison Wesley Longman).
Cited on page(s): 143

[55] Elizabeth A. Kendall. Role modeling for agent system analysis, design, and implementation. In *ASA/MA*, pages 204–218. IEEE Computer Society, Octomber 1999. ISBN 0-7695-0340-3.
Cited on page(s): 140, 143

[56] Elizabeth A. Kendall. Agent software engineering with role modelling. In Paolo Ciancarini and Michael Wooldridge, editors, *AOSE*, volume 1957 of *Lecture Notes in Computer Science*, pages

163–170. Springer, 2000. ISBN 3-540-41594-7.
Cited on page(s): 140, 143

[57] Bakhadyr Khoussainov and Anil Nerode. *Automata Theory and its Applications*. Progress in Computer Science and Applied Logic (PCS). Birkhǎuser Boston, 1^{st} edition, June 2001. ISBN 978-0817642075.
Cited on page(s): 22

[58] David Kinny, Michael P. Georgeff, and Anand S. Rao. A methodology and modelling technique for systems of BDI agents. In *Proceedings of MAAMAW 1996*, pages 56–71, 1996.
Cited on page(s): 140, 142

[59] Christos Kloukinas, George Lekeas, and Kostas Stathis. From agent game protocols to implementable roles. In *EUMAS 08, Sixth European Workshop on Multi-Agent Systems, Bath, UK*, pages 1–15, December 2008.
Cited on page(s): 14

[60] R. Kowalski and M. Sergot. A logic-based calculus of events. *New Generation Computing*, 4(1):67–95, 1986.
Cited on page(s): 45, 47

[61] Robert Kowalski and Fariba Sadri. The situation calculus and event calculus compared. In *Proceedings of the 1994 International Symposium on Logic programming*, ILPS '94, pages 539–553. MIT Press, Cambridge, MA, USA, 1994. ISBN 0-262-52191-1. URL `http://portal.acm.org/citation.cfm?id=200616.200660`.
Cited on page(s): 46

[62] Robert A. Kowalski and Fariba Sadri. Reconciling the event calculus with the situation calculus. *Journal of Logic Programming*, 31(1–3):39–58, April – June 1997.
Cited on page(s): 46

[63] Peep Küngas and Mihhail Matskin. Web services roadmap: The semantic web perspective. In *AICT/ICIW*, page 130. IEEE Computer Society, 2006. ISBN 0-7695-2522-9.
Cited on page(s): 2

[64] Mark V. Lawson. Finite automata. In Dimitrios Hristu-Varsakelis and William S. Levine, editors, *Handbook of Networked and Em-*

bedded Control Systems, pages 117–144. Birkhäuser, 1st edition, 2005. ISBN 0-8176-3239-5.
Cited on page(s): 22

[65] George Lekeas and Kostas Stathis. An agent development framework based on social positions. In *SMC (6)*, pages 5461–5466. IEEE, 2004. ISBN 0-7803-8566-7.
Cited on page(s): 14

[66] George K. Lekeas and Kostas Stathis. Agents acquiring resources through social positions: An activity-based approach. In O. de Bruijn and K. Stathis, editors, *Proceedings of the 1st International Workshop on Socio-Cognitive Grids.* Santorini, Greece, June 2003.
Cited on page(s): 4, 14

[67] Hector J. Levesque, Fiora Pirri, and Raymond Reiter. Foundations for the situation calculus. *Electron. Trans. Artif. Intell*, 2:159–178, 1998.
Cited on page(s): 35

[68] Nathaniel Love, Timothy Hinrichs, David Haley, Eric Schkufza, and Michael Genesereth. General game playing: Game description language specification. Technical Report March 4 2008, Stanford University, 2008.
Cited on page(s): 23

[69] Emil Constantin Lupu. *A Role-Based Framework For Distributed Systems Management.* PhD thesis, Department of Computing, Imperial College of Technology, Science and Medicine, United Kingdom, 2000.
Cited on page(s): 140

[70] N.Ã. Lynch and M.R̃. Tuttle. Hierarchical correctness proofs for distributed algorithms. Technical Report MIT/LCS/TR-387, MIT, 1987.
Cited on page(s): 115

[71] Ashok U. Mallya and Munindar P. Singh. An algebra for commitment protocols. *Autonomous Agents and Multi-Agent Systems*, 14 (2):143–163, 2007.
Cited on page(s): 110

[72] J. McCarthy and P. J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. pages 26–45, 1987. ISBN 0-934613-45-1.
Cited on page(s): 35, 38, 40

[73] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982. ISBN 0387102353.
Cited on page(s): 63, 64

[74] R. Milner. *Communication and concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989. ISBN 0-13-115007-3.
Cited on page(s): 63

[75] Laurent Mounier. *Verification Methods For Behavioral Specifications*. PhD thesis, L' Universitè Joseph Fourier - Grenoble I, 1992.
Cited on page(s): 63

[76] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989. ISSN 0018-9219.
Cited on page(s): 133

[77] Paul Muschamp. An introduction to web services. *BT Technology Journal*, 22(1):9–18, January 2006.
Cited on page(s): 2, 8

[78] Hyacinth S. Nwana. Software agents: An overview. *Knowledge Engineering Review*, 11(3):205–244, 1996.
Cited on page(s): 2, 16

[79] James Odell, H. Van, Dyke Parunak, and Bernhard Bauer. Extending UML for agents. In Gerd Wagner, Yves Lesperance, and Eric Yu, editors, *Proc. of the Agent-Oriented Information Systems Workshop at the 17th National conference on Artificial Intelligence*, pages 3–17. AOIS Workshop at AAAI 2000, Austin, TX, August 2000.
Cited on page(s): 3, 4

[80] Andrea Omicini. From objects to agent societies: Abstractions and methodologies for the engineering of open distributed systems. In Antonio Corradi, Andrea Omicini, and Agostino Poggi, editors, *WOA 2000: Dagli Oggetti agli Agenti. 1st AI*IA/TABOO Joint Workshop "From Objects to Agents": Evolutive Trends of Software Systems, 29-30 May 2000, Parma, Italy*, pages 29–34. Pitagora

Editrice Bologna, 2000. ISBN 88-371-1195-9.
Cited on page(s): 140, 144

[81] Andrea Omicini. SODA: Societies and infrastructures in the analysis and design of agent-based systems. In Paolo Ciancarini and Michael Wooldridge, editors, *AOSE*, volume 1957 of *Lecture Notes in Computer Science*, pages 185–193. Springer, 2000. ISBN 3-540-41594-7.
Cited on page(s): 140, 144

[82] Andrea Omicini, Paolo Petta, and Robert Tolksdorf, editors. *Engineering Societies in the Agents World II, Second International Workshop, ESAW 2001, Prague, Czech Republic, July 7, 2001, Revised Papers*, volume 2203 of *Lecture Notes in Computer Science*. Springer, 2001. ISBN 3-540-43091-1.
Cited on page(s): 2, 17

[83] Lin Padgham and Michael Winikoff. Prometheus: a methodology for developing intelligent agents. In *AOSE'02: Proceedings of the 3rd international conference on Agent-oriented software engineering III*, pages 174–185. Springer-Verlag, Berlin, Heidelberg, 2003. ISBN 3-540-00713-X.
Cited on page(s): 140, 141, 144

[84] David Park. Concurrency and automata on infinite sequences. In Peter Deussen, editor, *Theoretical Computer Science*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer Berlin / Heidelberg, 1981. 10.1007/BFb0017309.
Cited on page(s): 63

[85] Simon Parsons. Knowledge in action: Logical foundations for specifying and implementing dynamical systems by raymond reiter, mit press, 0-262-18218-1, 448 pp. *Knowledge Eng. Review*, 20(4):431–432, 2005. URL `http://dx.doi.org/10.1017/S0269888906210749`.
Cited on page(s): 35

[86] Javier Pinto and Raymond Reiter. Reasoning about time in the situation calculus. *Ann. Math. Artif. Intell*, 14(2-4):251–268, 1995.
Cited on page(s): 46

[87] Javier A. Pinto. *Temporal Reasoning in the Situation Calculus.* Ph.D. dissertation, Department of Computer Science, University

of Toronto, Toronto, 1994.
Cited on page(s): 46

[88] Jeremy Pitt, Lloyd Kamara, and Alexander Artikis. Interaction patterns and observable commitments in a multi-agent trading scenario. In *AGENTS '01: Proceedings of the fifth international conference on Autonomous agents*, pages 481–488. ACM, New York, NY, USA, 2001. ISBN 1-58113-326-X.
Cited on page(s): 3

[89] Jeremy Pitt and Abe Mamdani. A protocol-based semantics for an agent communication language. In *IJCAI '99: Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, pages 486–491. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999. ISBN 1-55860-613-0.
Cited on page(s): 23

[90] Jeremy Pitt, Abe Mamdani, and Patricia Charlton. The open agent society and its enemies: a position statement and research programme. *Telematics and Informatics*, 18(1):67–87, 2001.
Cited on page(s): 2, 17

[91] Alferes Renwei, R. Li, and Moniz Pereira. Concurrent actions and changes in the situation calculus. In Hector Geffner, editor, *Proc. of IBERAMIA 94*, pages 93–104. McGraw-Hill, March 25 1994.
Cited on page(s): 46

[92] Davide Sangiorgi. On the origins of bisimulation and coinduction. *ACM Trans. Program. Lang. Syst.*, 31:15:1–15:41, May 2009. ISSN 0164-0925. URL `http://doi.acm.org/10.1145/1516507.1516510`.
Cited on page(s): 63

[93] Bastin Tony Roy Savarimuthu, Martin K. Purvis, Marcos De Oliveira, and Maryam Purvis. Towards secure interaction in agent societies. In *PST*, pages 143–148, 2004.
Cited on page(s): 17

[94] Christoph Schroth and Till Janner. Web 2.0 and SOA: Converging concepts enabling the internet of services. *IT Professional*, 9(3): 36–41, 2007.
Cited on page(s): 2

[95] Murray Shanahan. Event calculus planning revisited. *Lecture Notes in Computer Science*, 1348:390–402, 1997. ISSN 0302-9743.
Cited on page(s): 45

[96] Murray Shanahan. *Solving the frame problem - a mathematical investigation of the common sense law of inertia.* Artificial Intelligence. MIT Press, Cambridge, Massachusetts, USA, 1997. ISBN 978-0-262-19384-9. I-XXXIV, 1-407 pp.
Cited on page(s): 45

[97] Murray Shanahan. The event calculus explained. In *Artificial Intelligence Today*, pages 409–430. 1999.
Cited on page(s): 45, 47

[98] Murray Shanahan. The ramification problem in the event calculus. In Dean Thomas, editor, *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI-99-Vol1)*, pages 140–146. Morgan Kaufmann Publishers, S.F., July 31 – August 6 1999.
Cited on page(s): 45

[99] Christophe Sibertin-Blanc and Nabil Hameurlain. Participation components for holding roles in multiagent systems protocols. In Marie Pierre Gleizes, Andrea Omicini, and Franco Zambonelli, editors, *ESAW*, volume 3451 of *Lecture Notes in Computer Science*, pages 60–73. Springer, 2004. ISBN 3-540-27330-1.
Cited on page(s): 132, 134

[100] Nikolaos Skarmeas. A framework for multi-agent modelling and office automation. In Leon Sterling, editor, *Proceedings of the Second International Conference on Applications of Prolog*, pages 517–526, April 1994.
Cited on page(s): 140, 142, 156

[101] Nikolaos Skarmeas. Organizations through roles and agents. In *International Workshop on the Design of Cooperative Systems (COOP'95), Antibes-France*, pages 185–194, November 1995.
Cited on page(s): 140, 142

[102] Kostas Stathis. *Game–based development of interactive systems.* PhD thesis, Department of Computing, Imperial College London, November 1996.
Cited on page(s): 38, 148, 151

[103] Kostas Stathis. Towards a game-based architecture for developing complex interactive components in computational logic. In A. Brogi and P. Hill, editors, *Proceedings of the First International Workshop on Component-based Software Development in Computational Logic (COCL98).* Pisa, Italy, September 19, 1998.
Cited on page(s): 24, 35

[104] Kostas Stathis. A game-based architecture for developing interactive components in computational logic. *Journal of Functional and Logic Programming*, 2000(5):1–27, 2000.
Cited on page(s): 2, 9, 24, 35, 49, 139, 155

[105] Kostas Stathis, George Lekeas, and Christos Kloukinas. Competence checking for the global E-service society using games. In Gregory M. P. O'Hare, Alessandro Ricci, Michael J. O'Grady, and Oguz Dikenelli, editors, *ESAW*, volume 4457 of *Lecture Notes in Computer Science*, pages 384–400. Springer, 2006. ISBN 978-3-540-75522-7.
Cited on page(s): 14, 31, 38, 121

[106] Kostas Stathis and Marek J. Sergot. Games as a metaphor for interactive systems. In Martina Angela Sasse, Jim Cunningham, and Russel L. Winder, editors, *BCS HCI*, pages 19–33. Springer, 1996. ISBN 3-540-76069-5.
Cited on page(s): 23, 24, 35

[107] Andrew S. Tanenbaum. *Distributed Systems: Principles and Paradigms.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2^{nd} edition, October 2006. ISBN 0132392275.
Cited on page(s): 2

[108] XSB Team. Xsb prolog project on sourceforge. Online. URL `http://xsb.sourceforge.net/`. visited 21 May 2010.
Cited on page(s): 158

[109] Francesca Toni and Kostas Stathis. Access-as-you-need: A computational logic framework for accessing resources in artificial societies. In Paolo Petta, Robert Tolksdorf, and Franco Zambonelli, editors, *ESAW*, volume 2577 of *Lecture Notes in Computer Science*, pages 126–140. Springer, 2002. ISBN 3-540-14009-3.
Cited on page(s): 1, 3

[110] Kristof Van Belleghem, Marc Denecker, and Danny De Schreye. Combining situation calculus and event calculus. In *ICLP*, pages

83–97, 1995.
Cited on page(s): 46

[111] Robert Jan (Rob) van Glabbeek and W. Peter Weijland. Branching time and abstraction in bisimulation semantics. *Journal of the ACM (JACM)*, 43(3):555–600, 1996.
Cited on page(s): 64, 65, 66, 119

[112] Mahadevan Venkatraman and Munindar P. Singh. Verifying compliance with commitment protocols. *Autonomous Agents and Multi-Agent Systems*, 2(3):217–236, 1999.
Cited on page(s): 112, 113

[113] D. Vergamini. Verification by means of observational equivalence on automata. Report 501, Institut National de Recherche en Informatique et en Automatique (INRIA), Sophia-Antipolis, France, Valbonne Cedex, March 1986.
Cited on page(s): 63

[114] D. S. Warren. Memoing for logic programs. *Communications of the ACM*, 35(3):93–111, 1992.
Cited on page(s): 158

[115] Marc Weiser. The world is not a desktop. *ACM Interactions*, 1 (1):7–8, November 1994. ISSN 1072-5520.
Cited on page(s): 1

[116] Mark Weiser. The computer for the 21st century. *SIGMOBILE Mob. Comput. Commun. Rev.*, 3(3):3–11, 1999. ISSN 1559-1662. URL `http://doi.acm.org/10.1145/329124.329126`.
Cited on page(s): 1

[117] Mark F. Wood. Multiagent systems engineering: A methodology for analysis and design of multiagent systems. MSc Thesis, AFIT/GCS/ENG/00M-26, 2000.
Cited on page(s): 140, 142

[118] Mark F. Wood and Scott A. DeLoach. An overview of the multiagent systems engineering methodology. In Paolo Ciancarini and Michael Wooldridge, editors, *AOSE*, volume 1957 of *Lecture Notes in Computer Science*, pages 207–222. Springer, 2000. ISBN 3-540-41594-7.
Cited on page(s): 140, 142

[119] Michael Wooldridge, Nicholas R. Jennings, and David Kinny. The gaia methodology for agent-oriented analysis and design. *Autonomous Agents and Multi-Agent Systems*, 3:285–312, 2000. ISSN 1387-2532.
Cited on page(s): 140, 142

[120] Q. Yan, L.J. Shan, and X.J. Mao. Romas: A role-based modelling method for multi-agent system. In Jian Ping Li, Jing Zhao, Jiming Liu, Ning Zhong, and John Yen, editors, *Proceedings of the* 2nd *International Conference on Active Media Technology (World Scientific Publishing)*, pages 156–161. Chongqing, PR China, May 29–31, 2003.
Cited on page(s): 140, 143

[121] Qi Yan, XinJun Mao, and ZhiChang Qi. Modelling distributed agent systems with RoMAS. *Sozionik aktuell*, 4(3):46–55, September 2003. ISSN 1617-2477.
Cited on page(s): 140, 143

[122] Franco Zambonelli, Nicholas R. Jennings, and Michael Wooldridge. Developing multiagent systems: The Gaia methodology. *ACM Transactions on Software Engineering and Methodology*, 12(3):317–370, July 2003. ISSN 1049-331X.
Cited on page(s): 140, 142

Appendix A

Prolog Code for the Situation Calculus Representation

```
:- use_module(library(lists)).
:- use_module(library(system)).

% Frame axioms
holds(sit(Name,Id,[]),F):-
    initially(F,sit(Name,Id,[])).
holds(sit(Name,Id,[Move|Moves]),F):-
    effect(F,Move,sit(Name,Id,Moves)).
holds(sit(Name,Id,[Move|Moves]),F):-
    holds(sit(Name,Id,Moves),F),
    \+ abnormal(F,Move,sit(Name,Id,Moves)).


% What holds initially at s0
initially(role_of(kostas,initiator),sit(query,s0,[])).
initially(role_of(george,client),sit(query,s0,[])).

% Available Moves
available(sit(query,_,_),select(_,query,_)).
available(sit(query,_,_),select(_,inform,_)).
available(sit(query,_,_),select(_,failure,_)).
available(sit(query,_,_),select(_,refuse,_)).
available(sit(query,_,_),select(_,notunderstood,_)).

%Effect axioms
effects(sit(Name,Id,Ms),M,sit(Name,Id,[M|Ms])).

effect(last_move(M), M, sit(query,_,_)).

% Abnormality conditions in various situations
abnormal(last_move(M_old), M_new, sit(_, _, _)).

% What players can do
can(sit(query, _, _), select(john,query,_)).
can(sit(query, _, _), select(paul,inform,_)).
can(sit(query, _, _), select(paul,failure,_)).
can(sit(query, _, _), select(paul,refuse,_)).
can(sit(query, _, _), select(paul,notunderstood,_)).

```

```
% run it as  game(sit(query,s0,[]),R).
game(Sit, Result):-
    terminating(Sit, Result).
game(Sit, Result):-
    \+ terminating(Sit, _),
    valid(Sit,Move),
    effects(Sit,Move,NewSit),
    game(NewSit, Result).

% terminating conditions
terminating(Sit, Sit):-
    holds(Sit,last_move(select(_,X,_))),
    member(X, [inform,failure,refuse,notunderstood]).

valid(Game, Move):-
    available(Game, Move),
    legal(Game, Move),
    can(Game, Move).

legal(sit(query,Id,N), select(P1, query,_)):-
    holds(sit(query,Id,N),role_of(P1,initiator)),
    \+ holds(sit(query,Id,N),last_move(_)).
legal(sit(query,Id,N), select(P1,inform,_)):-
    holds(sit(query,Id,N),role_of(P1,client)),
    holds(sit(query,Id,N),last_move(select(_,query,_))).
legal(sit(query,Id,N), select(P1, failure,_)):-
    holds(sit(query,Id,N),role_of(P1,client)),
    holds(sit(query,Id,N),last_move(select(_,query,_))).
legal(sit(query,Id,N), select(P1,refuse,_)):-
    holds(sit(query,Id,N),role_of(P1,client)),
    holds(sit(query,Id,N),last_move(select(_,query,_))).
legal(sit(query,Id,N), select(P1,notunderstood,_)):-
    holds(sit(query,Id,N),role_of(P1,client)),
    holds(sit(query,Id,N),last_move(select(_,query,_))).
```

Listing A.1: Situation Calculus Prolog Program

Appendix B

Prolog Code for the Event Calculus Representation

```prolog
:- use_module(library(lists)).
:- use_module(library(system)).

holds(sit(N,Id,Tn,Nn), P):-
    0 =< Tn,
    initially(sit(N,Id,Ti,Ni), P),
    \+ clipped(P, sit(N,Id,Ti,Ni), sit(N,Id,Tn,Nn)).
holds(sit(N,Id,Tn,Nn), P):-
    happens(E, Ti, Ni, Nn),
    Ti < Tn,
    initiates(E, P, sit(N,Id,Ti,Ni)),
    \+ clipped(P, sit(N,Id,Ti,Ni), sit(N,Id,Tn,Nn)).

clipped(P, sit(N,Id,Ti,_), sit(N,Id,Tn,Nn)):-
    happens(Estar, Tj, Nj, Nn),
    Ti < Tj,
    Tj < Tn,
    terminates(Estar, P, sit(N,Id,Tj,Nj)).

happens(at(En,Tn), Tn, [at(En,Tn)|Sn], [at(En, Tn)|Sn]).
happens(at(Ei,Ti), Ti, [at(Ei,Ti)|Si], [at(_, _)|Sn]):-
    happens(at(Ei, Ti), Ti, [at(Ei,Ti)|Si], Sn).
happens(E, Tn, [at(En, Tn)|Sn], [at(En, Tn)|Sn]):-
    member(E, En).
happens(E, Ti, [at(Ei,Ti)|Si], [at(_, _)|Sn]):-
    happens(E, Ti, [at(Ei,Ti)|Si], Sn).

effects(sit(Name, Id, T, N),at(Es,T),sit(Name,Id,NewT,[at(Es,T)|N])):-
    T >= 0,
    NewT is T + 1.

% run it as game(sit(auc,s0,0,[]),R).
game(Sit, Result):-
    terminating(Sit, Result).
game(Sit, Result):-
    \+ terminating(Sit, _),
    Sit = sit(_,_,_,Nar),
    writelist([->, Nar],nl),
    assume(valid(Sit, Move), Sit, Move, Episode),
    effects(Sit, Episode, NewSit),
    game(NewSit, Result).

assume(Valid, sit(N,Id,T,Ns), E, at(Es,T)):-
    findall(E, Valid, All),
    sublist(Es, All),
    acceptable(sit(N,Id,T,Ns), Es).

acceptable(Sit, Es):-
    \+ cyclic(Sit, Es).

cyclic(sit(_,_,T,Ns), Es):-
    member(select(P,bid,_), Es),
    happens(select(_,cfp,_), Ti, _, Ns),
    Ti =< T,
    happens(select(P,bid,_), Tj, _, Ns),
    happens(select(P,bid,_), Tk, _, Ns),
    Tj < Ti,
    Tk < Ti,
    \+ Tj = Tk.

valid(sit(N,Id,T,Es), M):-
    available(sit(N,Id,T,Es), M),
    legal(sit(N,Id,T,Es), M),
    can(sit(N,Id,T,Es), M).

incompatible(select(P, bid,_), C):-
    member(select(P, nobid,_), C).
incompatible(select(P, nobid,_), C):-
    member(select(P, bid,_), C).

initially(sit(auc,s0,0,[]), role_of(john, auctioneer)).
initially(sit(auc,s0,0,[]), role_of(paul, bidder)).
initially(sit(auc,s0,0,[]), role_of(chris, bidder)).

terminating(sit(auc,Id,T,N), sit(auc, Id, T, N)):-
    holds(sit(auc,Id,T,N), last_moves([select(_,X,_)])),
    member(X, [adjudicate,withdraw]).

initiates(at(Es, T),  last_moves(Es), sit(auc,_,T,_)).
terminates(at(_, T), last_moves(_), sit(auc,_,T,_)).

available(sit(auc,_,_,_),select(_,start,_)).
available(sit(auc,_,_,_),select(_,cfp,_)).
available(sit(auc,_,_,_),select(_,bid,_)).
available(sit(auc,_,_,_),select(_,adjudicate,_)).
available(sit(auc,_,_,_),select(_,withdraw,_)).

legal(sit(auc,Id,T,N), select(P1, start,_)):-
    holds(sit(auc,Id,T,N),role_of(P1,auctioneer)),
    \+ holds(sit(auc,Id,T,N),last_moves(_)).
legal(sit(auc,Id,T,N), select(P1,cfp,_)):-
    holds(sit(auc,Id,T,N),role_of(P1,auctioneer)),
    holds(sit(auc,Id,T,N),last_moves([select(P1,start,_)])).
legal(sit(auc,Id,T,N), select(P1,cfp,_)):-
    holds(sit(auc,Id,T,N),role_of(P1,auctioneer)),
    holds(sit(auc,Id,T,N),last_moves(Es)),
    once(member(select(P,bid,_), Es)),
    holds(sit(auc,Id,T,N),role_of(P,bidder)).


```

```
legal(sit(auc,Id,T,N), select(P1, bid,_)):-
    holds(sit(auc,Id,T,N),role_of(P1,bidder)),
    holds(sit(auc,Id,T,N),last_moves([select(P2,cfp,_)])),
    holds(sit(auc,Id,T,N),role_of(P2,auctioneer)).
legal(sit(auc,Id,T,N), select(P1,withdraw,_)):-
    holds(sit(auc,Id,T,N),role_of(P1,auctioneer)),
    holds(sit(auc,Id,T,N),last_moves([])),
        (
           \+ happens(select(Pi,bid,_), Ti, Ni, N)
         ;
           (once(happens(select(Pi,bid,_), Ti, Ni, N)),
            happens(select(Pj,bid,_), Ti, Ni, N),
            \+ Pi=Pj
            )
         ).
legal(sit(auc,Id,T,N), select(P1,adjudicate,_)):-
    holds(sit(auc,Id,T,N),role_of(P1,auctioneer)),
    holds(sit(auc,Id,T,N),last_moves([])),
    once(happens(select(Pi,bid,_), Ti, Ni, N)),
        forall(
               (holds(sit(auc,Id,T,N),role_of(Pj,bidder)), \+ (Pj = Pi))
                   ,
               \+ happens(select(Pj,bid,_), Ti, Ni, N)
        ).
legal(sit(auc,Id,T,N), select(P1,adjudicate,_)):-
    holds(sit(auc,Id,T,N),role_of(P1,auctioneer)),
    holds(sit(auc,Id,T,N),last_moves([])),
    once(happens(select(Pi,bid,_), Ti, Ni, N)),
        forall(
               (holds(sit(auc,Id,T,N),role_of(Pj,bidder)), \+ (Pj = Pi))
                   ,
               \+ happens(select(Pj,bid,_), Ti, Ni, N)
        ).

can(sit(auc, _, _, _), select(john,start,_)).
can(sit(auc, _, _, _), select(john,cfp,_)).
can(sit(auc, _, _, _), select(john,adjudicate,_)).
can(sit(auc, _, _, _), select(john,withdraw,_)).
can(sit(auc, _, _, _), select(paul,bid,_)).
can(sit(auc, _, _, _), select(chris,bid,_)).

forall(G1, G2):-
    \+ (call(G1),
    \+ call(G2)).

combinations(List, [H|T]):-
    append([H|T], _, List).
combinations(_, []).

writelist([],nl):-nl,!.

writelist([],sl):-!.

writelist([H|B],L):-
    write(H),
    write(' '),
    !,
    writelist(B,L).

sublist([H | T], [H | U]):-
    initialsublist(T, U).
initialsublist([], L).
initialsublist([H | T], [H | U]):-
    initialsublist(T, U).
```

Listing B.1: Prolog Program for Event Calculus Representation

Appendix C

Prolog Code for Bisimulation

```
:- use_module(library(lists)).
:- use_module(library(system)).
:- set_prolog_flag(double_quotes,string).
% allow the redefinition of the role_def predicate.
:- dynamic role_def/3.
% load the file with the protocol information.
:-consult('protocolfile.pl').

% run it as protocol_to_roles(auction,[], Z).
protocol_to_roles(Name, BisimulatedList, FinalProtocol):-
    protocol(Name,Roles,States,_,_,_,_,_,OriginalProtocol),
    create_aut(Roles, Name,OriginalProtocol, States),
    shell("./runbisimulation"),
    process_roles(Roles, Name, OriginalProtocol, NewProtocol,BisimulatedList),
    protocol_to_roles_internal(Roles,NewProtocol,NewBisimulatedList,FinalProtocol,
                               States,Name),
    count(FinalProtocol, Y),
    max_list(States, MaxV),
    MaximumValue is MaxV + 1,
    append([des(0, Y, MaximumValue)], FinalProtocol, LastProtocol),
    write_list_to_file(protocol, LastProtocol),
    !.

protocol_to_roles_internal(Roles, FinalProtocol, RoleList,FinalProtocol,_,_):-
    \+ RoleList = [],
    !.
protocol_to_roles_internal(Roles,InitialProtocol,BisimulatedRoleList,FinalProtocol,
                           States,Name):-
    shell("./cleanup"),
    create_aut(Roles, Name, InitialProtocol, States),
    shell("./runbisimulation"),
    process_roles(Roles, Name, InitialProtocol, NewProtocol, UpdatedList),
    \+ BisimulatedRoleList = [],
    \+ BisimulatedRoleList = UpdatedList,
    protocol_to_roles_internal(Roles,NewProtocol,UpdatedList,NewUpdatedList,FinalProtocol,
                               States,Name).

process_roles([], _, EndProtocol, EndProtocol, NewList).
process_roles([RoleOne|RoleRest],Name,Protocol,NewProtocol,EndBisimRoleList):-
    process_role(Name,Protocol,RoleOne,NewRoleOneList,NewWholeProtocol),
    append(NewRoleOneList,[],BisimListOne),
    !,
    process_roles(RoleRest,Name,NewWholeProtocol,NewProtocol,NewListRest),
    append(BisimListOne,NewListRest,EndBisimRoleList).

process_role(ProtocolName, ProtocolTransitions,Role,NewRoleList,NewProtocol):-
    atom_concat('bb',Role,NewRole),
    read_content_file(NewRole,RoleTransitionList),
    process_role_transitions(ProtocolTransitions,Role,RoleTransitionList,NewRoleList,
                         NewProtocol).

process_role_transitions(Final, _, [], _, Final):-!.
process_role_transitions(Protocol,Role,[RoleTransitionListOne|RoleTransitionListRest],
                     NewRoleProtocol,FinalProtocol):-
   process_role_transitions_one(Protocol,Role,RoleTransitionListOne,NewRoleProtocolOne,
                          InterimProtocol),
   process_role_transitions(InterimProtocol,Role,RoleTransitionListRest,NewRoleProtocolRest,
                        FinalProtocol),
   append(NewRoleProtocolOne, NewRoleProtocolRest, NewRoleProtocol).

process_role_transitions_one(Protocol, Role, Transition, [],Protocol):-
   Transition = des(_,_,_).
process_role_transitions_one(Protocol, Role,Transition,Transition],Protocol):-
   Transition = (InitialState, (Role, Action, Receiver), FinalState).
process_role_transitions_one(Protocol,Role,Transition,[Transition],Protocol):-
   Transition = (InitialState, (Sender, Action, Role), FinalState).
process_role_transitions_one(Protocol,Role,Transition,[Transition],Protocol):-
   Transition = (InitialState, (Sender, Action, Receivers), FinalState),
   is_list(Receivers),
   member(Role, Receivers).
process_role_transitions_one(Protocol,Role,Transition,NewRoleTransition,NewProtocol):-
   repair_transition(Protocol, Role, Transition, NewRoleTransition,Protocol).

backwards_tau_transitions(State, TransitionList, AllStateList):-
   calculate_incoming_list_transitions(TransitionList,State,IncomingListTransitions),
   process_incoming_list_transitions([IncomingListTransitionsOne|IncomingListTransitionsNext],
                                AllStateList).

process_incoming_list_transitions([IncomingListTransitionsOne|IncomingListTransitionsNext],
                             States,RoleTransitionList):-
   process_incoming_list_transitions_one(IncomingListTransitionsOne,State1,
                                   RoleTransitionList),
   process_incoming_list_transitions(IncomingListTransitionsRest,StatesRest,
                                RoleTransitionList),
   append(State1, StatesRest, States).
process_incoming_list_transitions([], [], []).

process_incoming_list_transitions_one(TransitionOne,StateOne,TransitionList):-
   Transition1 = (InitialState, 'tau', FinalState),
   backwards_tau_transitions(InitialState, RoleTransitionList,AllStates).

repair_transition(Protocol, Role, Transition, NewRoleTransition,NewProtocol):-
   Transition=(InitialState,tau,FinalState),
   calculate_outgoing_list_transitions(Protocol,InitialState,ListOfTransitions),
   fix_transitions(Protocol,ListOfTransitions,Role,NewRoleTransition,NewProtocol).

calculate_outgoing_list_transitions([], _, []).
calculate_outgoing_list_transitions([ProtocolOne|ProtocolRest],StateOne,ListOfTransitions):-
   calculate_outgoing_list_transitions_state_one(ProtocolOne,StateOne,ListOfTransitionsOne),
   calculate_outgoing_list_transitions(ProtocolRest,StateOne,ListOfTransitionsRest),
   append(ListOfTransitionsRest, ListOfTransitionsOne, ListOfTransitions).

calculate_outgoing_list_transitions_state_one([], _, []).
calculate_outgoing_list_transitions_state_one(ProtocolOne,State,[ProtocolOne]):-
   ProtocolOne = (State, (_, _, _), _).
calculate_outgoing_list_transitions_state_one(ProtocolOne,State,[ProtocolOne]):-
   ProtocolOne = (State, tau, _).
calculate_outgoing_list_transitions_state_one(ProtocolOne, State,[]):-
   ProtocolOne = (X, (_, _, _), _),
   \+ X = State.
calculate_outgoing_list_transitions_state_one(ProtocolOne,State,[]):-
   ProtocolOne = (X, tau, _),
   \+ X = State.

calculate_incoming_list_transitions([], _, []).
calculate_incoming_list_transitions([ProtocolOne|ProtocolRest],StateOne,ListOfTransitions):-
  calculate_incoming_list_transitions_state_one(ProtocolOne,StateOne,ListOfTransitionsOne),
  calculate_incoming_list_transitions(ProtocolRest,StateOne,ListOfTransitionsRest),
  append(ListOfTransitionsRest, ListOfTransitionsOne, ListOfTransitions).

calculate_incoming_list_transitions_state_one([], _, []).
calculate_incoming_list_transitions_state_one(ProtocolOne,State,[ProtocolOne]):-
  ProtocolOne = (_, (_, _, _), State).
calculate_incoming_list_transitions_state_one(ProtocolOne,State,[ProtocolOne]):-
  ProtocolOne = (_, tau, State).
calculate_incoming_list_transitions_state_one(_, _, []).

find_problematic_tau_actions(Role,RoleTransitionList,
                             [ListOfTransitionsOne|ListOfTransitionsRest],
                             ProblematicTransitions):-
  find_problematic_tau_actions_one(Role,RoleTransitionList,ListOfTransitionsOne,
                                   ProblematicTransitionsOne),
  find_problematic_tau_actions(Role,RoleTransitionsList,ListOfTransitionsRest,
                               ProblematicTransitionsRest),
  append(ProblematicTransitionsOne,ProblematicTransitionsRest,ProblematicTransitions).

find_problematic_tau_actions_one(Role,RoleTransitionList,RoleList,Transition,[Transition]):-
  Transition = (InitialState, 'tau', FinalState),
  calculate_outgoing_list_transitions(RoleTransitionList,FinalState,OutGoingTransitionsList),
  member(Transition1, OutGoingTransitionsList),
  Transition1 = (FinalState, (Role, Action1, Receiver1), NewFinalState1),
  Transition2 = (InitialState, (Role, Action2, Receiver2), NewFinalState2),
  \+ Action1 = Action2.
find_problematic_tau_actions_one(Role,RoleTransitionList,RoleList,Transition,[Transition]):-
  Transition = (InitialState, 'tau', FinalState),
  calculate_outgoing_list_transitions(RoleTransitionList,FinalState,OutGoingTransitionsList),
  member(Transition1, OutGoingTransitionsList),
  Transition1 = (FinalState, (Role, Action1, Receiver1), NewFinalState1),
  Transition2 = (InitialState, (Role, Action2, Receiver2), NewFinalState2),
  \+ Receiver1 = Receiver2.
find_problematic_tau_actions_one(Role,RoleTransitionList,RoleList,Transition,[Transition]):-
  Transition=(InitialState,'tau',FinalState),
  calculate_outgoing_list_transitions(RoleTransitionList,FinalState,OutGoingTransitionsList),
  member(Transition1, OutGoingTransitionsList),
  Transition1=(FinalState,(Role,Action1,Receiver1),NewFinalState1),
  member(Transition2, OutGoingTransitionsList),
  Transition2 = (FinalState, 'tau', NewFinalState2),
  Transition3 = (NewFinalState2, (Role, Action2, Receiver2),NewFinalState2),
  \+ Action1 = Action2,
  \+ Receiver1 = Receiver2.
find_problematic_tau_actions_one(Role,RoleTransitionList,RoleList,Transition,[Transition]):-
  Transition=(InitialState,'tau',FinalState),
  calculate_outgoing_list_transitions(RoleTransitionList,FinalState,OutGoingTransitionsList),
  member(Transition1, OutGoingTransitionsList),
  Transition1=(FinalState,(Role,Action1,Receiver1),NewFinalState1),
  member(Transition2, OutGoingTransitionsList),
  Transition2 = (FinalState, 'tau', NewFinalState2),
  Transition3 = (NewFinalState2, (Role, Action2, Receiver2),NewFinalState2),
  \+ Action1 = Action2.
find_problematic_tau_actions_one(Role,RoleTransitionList,RoleList,Transition,[Transition]):-
  Transition=(InitialState,'tau',FinalState),
  calculate_outgoing_list_transitions(RoleTransitionList,FinalState,OutGoingTransitionsList),
  member(Transition1, OutGoingTransitionsList),
  Transition1=(FinalState,(Role,Action1,Receiver1),NewFinalState1),
  member(Transition2, OutGoingTransitionsList),
  Transition2 = (FinalState, 'tau', NewFinalState2),
  Transition3 = (NewFinalState2, (Role, Action2, Receiver2),NewFinalState2),
  \+ Receiver1 = Receiver2.

repair_transition3(Protocol, Role, Transition,NewRoleTransition,NewProtocol):-
  Transition = (InitialState, tau,FinalState),
  equivalent_states(InitialState,ListOfStates,Role),
  calculate_list_transitions(Protocol,ListOfStates,ListOfTransitions),
  fix_transitions(Protocol,ListOfTransitions,Role,NewRoleTransition,NewProtocol).

repair_transition2(Protocol, Role, Transition,NewRoleTransition,NewProtocol):-
  Transition = (InitialState, tau,FinalState),
  equivalent_states(InitialState,InitialStateEquivalents,Role),
  equivalent_states(FinalState,FinalStateEquivalents,Role),
  calculate_list_transitions2(Protocol,InitialStateEquivalents,FinalStateEquivalents,
                              ListOfTransitions),
  fix_transitions(Protocol,ListOfTransitions,Role,NewRoleTransitions,NewProtocol).

equivalent_states(OriginalState, ListOfStates, Role):-
  atom_concat('r',Role,NewResultsFile),
  read_content_file(NewResultsFile, AllEquivalentStates),
  find_equivalent_states(OriginalState, AllEquivalentStates, ListOfStates).

find_equivalent_states(_, [], []).
find_equivalent_states(State, [ PairOne | PairRest], FinalList):-
  find_equivalent_states_one(State, PairOne, FinalListOne),
  find_equivalent_states(State, PairRest, FinalListTwo),
  append(FinalListTwo, FinalListOne, FinalList).

% The results file is in the form (OriginalAut,BisimulatedAut)
find_equivalent_states_one(BisimulatedState,(OriginalState,BisimulatedState),
                           [OriginalState]):-
    !.
find_equivalent_states_one(BisimulatedState, (AState, AnotherState), []).

calculate_list_transitions_one(ProtocolOne, State, [ProtocolOne]):-
 ProtocolOne = (State, (Sender, Action, Receiver), FinalState),
 !.
calculate_list_transitions_one(ProtocolOne, State, []).

calculate_list_transitions2([], _,_, []).
calculate_list_transitions2([ProtocolOne|ProtocolRest],InitialEquivalentStates,
                             FinalEquivalentStates, ListOfTransitions):-
   calculate_list_transitions2_protocol_one(ProtocolOne,InitialEquivalentStates,
                                            FinalEquivalentStates,ListOfTransitionsOne),
   calculate_list_transitions2(ProtocolRest,InitialEquivalentStates,FinalEquivalentStates,
                               ListOfTransitionsRest),
   append(ListOfTransitionsRest, ListOfTransitionsOne, ListOfTransitions).

calculate_list_transitions2_protocol_one([],_, _, []).
```

```
calculate_list_transitions2_protocol_one([ProtocolOne|ProtocolRest],InitialStateEquivalent,
                                          FinalStateEquivalent,ListOfTransitions):-
    calculate_list_transitions2_one(ProtocolOne,InitialListStates,FinalListStates,
                                    TransitionsListOne),
    calculate_list_transitions2_protocol_one(ProtocolRest,InitialListStates,FinalListStates,
                                             NewTransitionList),
    append(NewTransitionList, TransitionsListOne, ListOfTransitions).


calculate_list_transitions2_one(ProtocolOne,InitialStateList,FinalStateList,[ProtocolOne]):-
 ProtocolOne = (State, tau, FinalState),
 member(State, InitialStateList),
 member(FinalState, FinalStateList),
 !.

calculate_list_transitions2_one(ProtocolOne, _, _, []).

update_transitions(Protocol,[OriginalTransitionOne|OriginalTransitionRest],Role,
                   NewTempRoleProtocol):-
   update_transitions_one(Protocol, OriginalTransitionOne,Role,TempProtocol),
   update_transitions(Protocol,OriginalTransitionRest,Role,RestTempProtocol),
   append(TempProtocol,RestTempProtocol , NewTempRoleProtocol).


fix_transitions(Protocol, [], _, _, Protocol).
fix_transitions(Protocol,[OriginalTransition|Rest],Role,NewRoleTransition,FinalProtocol):-
   fix_transitions_one(Protocol,OriginalTransition,Role,NewRoleTransitionOne,InterimProtocol),
   fix_transitions(InterimProtocol,Rest,Role,NewRoleTransitionRest,FinalProtocol),
   append(NewRoleTransitionOne, NewRoleTransitionRest, NewRoleTransition).

fix_transitions_one(Protocol, (_, (_, _, Role),_), Role, _, Protocol).
fix_transitions_one(Protocol, (_, (Role, _, _),_), Role, _, Protocol).
fix_transitions_one(Protocol,OriginalTransitionOne,Role,NewTransition,InterimProtocol):-
   OriginalTransitionOne=(InitialState,(Sender,Action,Receiver),FinalState),
   is_list(Receiver),
   \+ member(Role, Receiver),
   append([Role], Receiver, NewReceiver),
   sort(NewReceiver,SortedReceiver),
   append([(InitialState,(Sender,Action,SortedReceiver),FinalState)],[],NewTransition),
   delete(Protocol,OriginalTransitionOne,IntermediateProtocol),
   append(IntermediateProtocol, NewTransition, InterimProtocol).

fix_transitions_one(Protocol, OriginalTransitionOne, Role, [], Protocol):-
   OriginalTransitionOne =(InitialState,(Sender,Action,Receiver),FinalState),
   is_list(Receiver),
   member(Role, Receiver).

fix_transitions_one(Protocol,OriginalTransitionOne,Role,NewTransition,InterimProtocol):-
   OriginalTransitionOne=(InitialState,(Sender,Action,Receiver),FinalState),
   \+ is_list(Receiver),
   append([Role], [Receiver], NewReceiver),
   sort(NewReceiver,SortedReceiver),
   append([(InitialState,(Sender,Action,SortedReceiver),FinalState)],[],NewTransition),
   delete(Protocol,OriginalTransitionOne,IntermediateProtocol),
   append(IntermediateProtocol, NewTransition, InterimProtocol).


write_list_to_file(FileName, [ListOne | Rest]):-
   tell(FileName),
   write_list_to_file_one(ListOne),
   write_list_to_file_rest(Rest),
   told.

write_list_to_file_one(Content):-
   Content = des(_, _, _),
   write(Content),
   nl.

write_list_to_file_one(Content):-
   Content = (InitialState, (Sender, Message, Receiver), FinalState),
   write('('),
   write(InitialState),
   write(',\"('),
   write(Sender),
   write(','),
   write(Message),
   write(','),
   write(Receiver),
   write(')\",'),
   write(FinalState),
   write(')'),
   nl.

write_list_to_file_rest([]).

write_list_to_file_rest([RestOne | Remaining]):-
  write_list_to_file_one(RestOne),
  write_list_to_file_rest(Remaining).

read_content_file(File, List):-
  open(File, read, Str),
  list_of_terms(Str, List).


list_of_terms(Str, []):-
  at_end_of_stream(Str) .
list_of_terms(Str, List):-
  read_line_to_codes(Str,Codes),
  string_to_list(String, Codes),
  string_to_atom(String,Atom),
  term_to_atom(Term, Atom),
  list_of_terms(Str, NewList),
  append([Term], NewList, List),
  !.

create_aut([RoleHead | RoleTails], Name, Protocol, States):-
  create_aut_one(RoleHead, Name, Protocol, States),
  create_aut(RoleTails, Name, Protocol, States).
create_aut([], _, _,_):-!.

create_aut_one(X, Name, Protocol, States):-
  pl2aut(Name,X,X, Protocol, States).

pl2aut(ProtocolName,Role,FileName, Protocol, States):-
  count(Protocol,X),
  count(States,Y),
  tell(FileName),
  write('des(0,'),
  write(X),
  write(','),
  write(Y),
  write(')'),
  nl,
  check_transition(Protocol, Role, FileName),
  change_filename(FileName).

change_filename(FileName):-
  file_name_extension(FileName,aut,X),
  rename_file(FileName,X).

check_transition([Head |Tails], Role, FileName):-
  check_one(Head, Role, FileName),
  !,
  check_transition(Tails, Role, FileName).
check_transition([],_,_):-
  told.

check_one(Transition, Role, FileName):-
  Transition = (InitialState,Move, EndState),
  Move = (Sender, _, _),
  Role = Sender,
  tell(FileName),
  write('('),
  write(InitialState),
  write(',"('),
  write(Move),
  write(')"'),
  write(','),
  write(EndState),
  write(')'),
  nl.

check_one(Transition, Role, FileName):-
  Transition = (InitialState,Move, EndState),
  Move = (_, _, Receiver),
  Role = Receiver,
  tell(FileName),
  write('('),
  write(InitialState),
  write(',"('),
  write(Move),
  write(')"'),
  write(','),
  write(EndState),
  write(')'),
  nl.

check_one(Transition, Role, FileName):-
  Transition = (InitialState,Move, EndState),
  Move = (_, _, Receiver),
  member(Role,Receiver),
  tell(FileName),
  write('('),
  write(InitialState),
  write(',"('),
  write(Move),
  write(')"'),
  write(','),
  write(EndState),
  write(')'),
  nl.

check_one(Transition, Role, FileName):-
```

```
  Transition = (State1,_, State2),
  tell(FileName),
  NewTransition = (State1,'tau',State2),
  write('('),
  write(State1),
  write(',"tau",'),
  write(State2),
  write(')'),
  nl.

count(List,N):-
  length(List,N).

writelist([],nl):-nl,!.
writelist([],sl):-!.
writelist([H|B],L):-
  write(H),
  write(' '),
  !,
  writelist(B,L).

runaut(Name):-
  protocol(Name,Roles,States,_,_,_,_,_,Protocol),
  create_aut(Roles, Name, Protocol, States).
```

Listing C.1: Prolog Code For Bisimulation

Appendix D

Event Calculus Implementation for the Mail Order Protocol

```
holds(sit(N,Id,Tn,Nn), P):-
    0 =< Tn,
    initially(sit(N,Id,Ti,Ni), P),
    \+ clipped(P, sit(N,Id,Ti,Ni), sit(N,Id,Tn,Nn)).
holds(sit(N,Id,Tn,Nn), P):-
    happens(E, Ti, Ni, Nn),
    Ti < Tn,
    initiates(E, P, sit(N,Id,Ti,Ni)),
    \+ clipped(P, sit(N,Id,Ti,Ni), sit(N,Id,Tn,Nn)).

clipped(P, sit(N,Id,Ti,_), sit(N,Id,Tn,Nn)):-
    happens(Estar, Tj, Nj, Nn),
    Ti < Tj,
    Tj < Tn,
    terminates(Estar, P, sit(N,Id,Tj,Nj)).

happens(at(En,Tn), Tn, [at(En,Tn)|Sn], [at(En, Tn)|Sn]).
happens(at(Ei,Ti), Ti, [at(Ei,Ti)|Si], [at(_, _)|Sn]):-
    happens(at(Ei, Ti), Ti, [at(Ei,Ti)|Si], Sn).
happens(E, Tn, [at(En, Tn)|Sn], [at(En, Tn)|Sn]):-
    member(E, En).
happens(E, Ti, [at(Ei,Ti)|Si], [at(_, _)|Sn]):-
    happens(E, Ti, [at(Ei,Ti)|Si], Sn).

effects(sit(Name,Id,T,N),at(Es,T),sit(Name,Id,NewT,[at(Es,T)|N])):-
    T >= 0,
    NewT is T + 1.

game(Sit, Result):-
    terminating(Sit, Result).
game(Sit, Result):-
    \+ terminating(Sit, _),
    Sit = sit(_,_,_,Nar),
    assume(valid(Sit, Move), Sit, Move, Episode),
    effects(Sit, Episode, NewSit),
    game(NewSit, Result).
```

```
assume(Valid, sit(N,Id,T,Ns), E, at([Es],T)):-
    findall(E, Valid, All),
    sublist2(Es, All),
    acceptable(sit(N,Id,T,Ns), [Es]).

acceptable(Sit, Es):-
    \+ cyclic(Sit, Es).

cyclic(sit(mail_order, _, T, Ns), Es):-
    Es=[select(paul,reorder,_)],
        sublist([at([select(john,order,_)],_),
                 at([select(paul,reorder,_)],_),
                 at([select(john,order,_)],_)],Ns).

valid(sit(N,Id,T,Es), M):-
   available(sit(N,Id,T,Es), M),
   legal(sit(N,Id,T,Es), M),
   can(sit(N,Id,T,Es), M).

incompatible(select(P, bid,_), C):-
   member(select(P, nobid,_), C).
incompatible(select(P, nobid,_), C):-
   member(select(P, bid,_), C).

initially(sit(mail_order,s0,0,[]), role_of(john, merchant)).
initially(sit(mail_order,s0,0,[]), role_of(paul, supplier)).

terminating(sit(mail_order,Id,T,N), sit(mail_order, Id, T, N)):-
   holds(sit(mail_order,Id,T,N), last_moves([select(P,X,_)])),
   holds(sit(mail_order,Id,T,N), role_of(P,supplier)),
   member(X, [notify,refuse]).
terminating(sit(mail_order,Id,T,N), sit(mail_order, Id, T, N)):-
   holds(sit(mail_order,Id,T,N), last_moves([select(P,X,_)])),
   holds(sit(mail_order,Id,T,N), role_of(P,merchant)),
   member(X, [withdraw]).

initiates(at(Es, T),  last_moves(Es), sit(mail_order,_,T,_)).
terminates(at(_, T), last_moves(_), sit(mail_order,_,T,_)).

available(sit(mail_order,_,_,_),select(_,order,_)).
available(sit(mail_order,_,_,_),select(_,reorder,_)).
available(sit(mail_order,_,_,_),select(_,refuse,_)).
available(sit(mail_order,_,_,_),select(_,confirm,_)).
available(sit(mail_order,_,_,_),select(_,withdraw,_)).
available(sit(mail_order,_,_,_),select(_,accept,_)).
available(sit(mail_order,_,_,_),select(_,notify,_)).

legal(sit(mail_order,Id,T,N), select(P1, order,_)):-
   holds(sit(mail_order,Id,T,N),role_of(P1,merchant)),
   \+ holds(sit(mail_order,Id,T,N),last_moves(_)).
legal(sit(mail_order,Id,T,N), select(P1, order,_)):-
   holds(sit(mail_order,Id,T,N),role_of(P1,merchant)),
   holds(sit(mail_order,Id,T,N),last_moves([select(P2,reorder,_)])),
   holds(sit(mail_order,Id,T,N),role_of(P2,supplier)).
legal(sit(mail_order,Id,T,N), select(P1,reorder,_)):-
   holds(sit(mail_order,Id,T,N),role_of(P1,supplier)),
   holds(sit(mail_order,Id,T,N),last_moves([select(P2,order,_)])),
   holds(sit(mail_order,Id,T,N),role_of(P2,merchant)).
legal(sit(mail_order,Id,T,N), select(P1,confirm,_)):-
   holds(sit(mail_order,Id,T,N),role_of(P1,supplier)),
   holds(sit(mail_order,Id,T,N),last_moves([select(P2,order,_)])),
   holds(sit(mail_order,Id,T,N),role_of(P2,merchant)).
legal(sit(mail_order,Id,T,N), select(P1, refuse,_)):-
   holds(sit(mail_order,Id,T,N),role_of(P1,supplier)),
   holds(sit(mail_order,Id,T,N),last_moves([select(P2,order,_)])),
   holds(sit(mail_order,Id,T,N),role_of(P2,merchant)).
legal(sit(mail_order,Id,T,N), select(P1, withdraw,_)):-
   holds(sit(mail_order,Id,T,N),role_of(P1,merchant)),
   holds(sit(mail_order,Id,T,N),last_moves([select(P2,confirm,_)])),
   holds(sit(mail_order,Id,T,N),role_of(P2,supplier)).
legal(sit(mail_order,Id,T,N), select(P1, accept,_)):-
   holds(sit(mail_order,Id,T,N),role_of(P1,merchant)),
   holds(sit(mail_order,Id,T,N),last_moves([select(P2,confirm,_)])),
   holds(sit(mail_order,Id,T,N),role_of(P2,supplier)).
legal(sit(mail_order,Id,T,N), select(P1, notify,_)):-
   holds(sit(mail_order,Id,T,N),role_of(P1,supplier)),
   holds(sit(mail_order,Id,T,N),last_moves([select(P2,accept,_)])),
   holds(sit(mail_order,Id,T,N),role_of(P2,merchant)).

can(sit(mail_order, _, _, _), select(john,order,_)).
can(sit(mail_order, _, _, _), select(john,withdraw,_)).
can(sit(mail_order, _, _, _), select(john,accept,_)).
can(sit(mail_order, _, _, _), select(paul,reorder,_)).
can(sit(mail_order, _, _, _), select(paul,confirm,_)).
can(sit(mail_order, _, _, _), select(paul,refuse,_)).
can(sit(mail_order, _, _, _), select(paul,notify,_)).

writelist([],nl):-nl,!.
writelist([],sl):-!.
writelist([H|B],L):-
   write(H),
   write(' '),!,
   writelist(B,L).

sublist([H | T], [H | U]):-
   initialsublist(T, U).

initialsublist([], _).
initialsublist([H | T], [H | U]):-
   initialsublist(T, U).

sublist2(X, Y):-
   member(X,Y).
sublist2([],[]).
```

Listing D.1: Event Calculus Representation for the Order Protocol

Appendix E

Mail Order Protocol in Situation Calculus

```
:- use_module(library(lists)).
:- use_module(library(system)).

holds(sit(Name,Id,[]),F):-
    initially(F,sit(Name,Id,[])).
holds(sit(Name,Id,[Move|Moves]),F):-
    effect(F,Move,sit(Name,Id,Moves)).
holds(sit(Name,Id,[Move|Moves]),F):-
    holds(sit(Name,Id,Moves),F),
    \+ abnormal(F,Move,sit(Name,Id,Moves)).

initially(role_of(john,merchant),sit(mail_order,s0,[])).
initially(role_of(paul,supplier),sit(mail_order,s0,[])).

available(sit(mail_order,_,_),select(_,order,_)).
available(sit(mail_order,_,_),select(_,confirm,_)).
available(sit(mail_order,_,_),select(_,refuse,_)).
available(sit(mail_order,_,_),select(_,withdraw,_)).
available(sit(mail_order,_,_),select(_,accept,_)).
available(sit(mail_order,_,_),select(_,notify,_)).

effects(sit(Name,Id,Ms),M,sit(Name,Id,[M|Ms])).

effect(last_move(M), M, sit(mail_order,_,_)).

abnormal(last_move(M_old), M_new, sit(_, _, _)).

can(sit(mail_order, _, _), select(john,order,_)).
can(sit(mail_order, _, _), select(john,withdraw,_)).
can(sit(mail_order, _, _), select(john,accept,_)).
can(sit(mail_order, _, _), select(paul,reorder,_)).
can(sit(mail_order, _, _), select(paul,confirm,_)).
can(sit(mail_order, _, _), select(paul,refuse,_)).
can(sit(mail_order, _, _), select(paul,notify,_)).


% run it as  game(sit(mail_order,s0,[]),R).
game(Sit, Result):-
    terminating(Sit, Result).
```

```
game(Sit, Result):-
    \+ terminating(Sit, _),
    valid(Sit,Move),
    effects(Sit,Move,NewSit),
    game(NewSit, Result).

terminating(Sit, Sit):-
    holds(Sit,last_move(select(_,X,_))),
    member(X, [refuse,notify,withdraw]).

valid(Game, Move):-
    available(Game, Move),
    legal(Game, Move),
    can(Game, Move),
    \+ cyclic(Game, Move).

cyclic(sit(_,_,Moves), Move):-
    Move = select(paul,reorder,_),
    sublist([select(john,order,_),select(paul,reorder,_),
             select(john,order,_)],Moves).

legal(sit(mail_order,Id,N), select(P1, order,P2)):-
    holds(sit(mail_order,Id,N),role_of(P1,merchant)),
    holds(sit(mail_order,Id,N),role_of(P2,supplier)),
    \+ holds(sit(mail_order,Id,N),last_move(_)).
legal(sit(mail_order,Id,N), select(P1,confirm,P2)):-
    holds(sit(mail_order,Id,N),role_of(P1,supplier)),
    holds(sit(mail_order,Id,N),last_move(select(P2,order,P1))),
    holds(sit(mail_order,Id,N),role_of(P2,merchant)).
legal(sit(mail_order,Id,N), select(P1, refuse,P2)):-
    holds(sit(mail_order,Id,N),role_of(P1,supplier)),
    holds(sit(mail_order,Id,N),last_move(select(P2,order,P1))),
    holds(sit(mail_order,Id,N),role_of(P2,merchant)).
legal(sit(mail_order,Id,N), select(P1, withdraw,P2)):-
    holds(sit(mail_order,Id,N),role_of(P1,merchant)),
    holds(sit(mail_order,Id,N),last_move(select(P2,confirm,P1))),
    holds(sit(mail_order,Id,N),role_of(P2,supplier)).
legal(sit(mail_order,Id,N), select(P1, accept,P2)):-
    holds(sit(mail_order,Id,N),role_of(P1,merchant)),
    holds(sit(mail_order,Id,N),last_move(select(P2,confirm,P1))),
    holds(sit(mail_order,Id,N),role_of(P2,supplier)).
legal(sit(mail_order,Id,N), select(P1, notify,P2)):-
    holds(sit(mail_order,Id,N),role_of(P1,supplier)),
    holds(sit(mail_order,Id,N),last_move(select(P2,accept,P1))),
    holds(sit(mail_order,Id,N),role_of(P2,merchant)).

sublist([H | T], [H | U]):-
    initialsublist(T, U).

initialsublist([], _).
initialsublist([H | T], [H | U]):-
    initialsublist(T, U).
```

Listing E.1: Situation Calculus Representation of the Order Protocol

Appendix F

Prolog code in Situation Calculus for a protocol with loops

```
:- use_module(library(lists)).
:- use_module(library(system)).

holds(sit(Name,Id,[]),F):-
    initially(F,sit(Name,Id,[])).
holds(sit(Name,Id,[Move|Moves]),F):-
    effect(F,Move,sit(Name,Id,Moves)).
holds(sit(Name,Id,[Move|Moves]),F):-
    holds(sit(Name,Id,Moves),F),
    \+ abnormal(F,Move,sit(Name,Id,Moves)).

initially(role_of(john,merchant),sit(negotiation,s0,[])).
initially(role_of(paul,customer),sit(negotiation,s0,[])).

available(sit(negotiation,_,_),select(_,request,_)).
available(sit(negotiation,_,_),select(_,accept,_)).
available(sit(negotiation,_,_),select(_,refuse,_)).
available(sit(negotiation,_,_),select(_,challenge,_)).
available(sit(negotiation,_,_),select(_,justify,_)).
available(sit(negotiation,_,_),select(_,retract,_)).
available(sit(negotiation,_,_),select(_,authenticate,_)).
available(sit(negotiation,_,_),select(_,certify,_)).
available(sit(negotiation,_,_),select(_,notunderstood,_)).
available(sit(negotiation,_,_),select(_,inform,_)).
available(sit(negotiation,_,_),select(_,reject,_)).

effects(sit(Name,Id,Ms),M,sit(Name,Id,[M|Ms])).

effect(last_move(M), M, sit(negotiation,_,_)).

abnormal(last_move(M_old), M_new, sit(_, _, _)).

can(sit(negotiation, _, _), select(john,request,_)).
can(sit(negotiation, _, _), select(john,justify,_)).
can(sit(negotiation, _, _), select(john,retract,_)).
can(sit(negotiation, _, _), select(john,authenticate,_)).
can(sit(negotiation, _, _), select(john,notunderstood,_)).
can(sit(negotiation, _, _), select(john,reject,_)).
can(sit(negotiation, _, _), select(paul,accept,_)).
can(sit(negotiation, _, _), select(paul,refuse,_)).
can(sit(negotiation, _, _), select(paul,challenge,_)).
can(sit(negotiation, _, _), select(paul,certify,_)).
can(sit(negotiation, _, _), select(paul,inform,_)).
% run it as  game(sit(negotiation,s0,[]),R).
game(Sit, Result):-
    terminating(Sit, Result),
    !.
game(Sit, Result):-
    \+ terminating(Sit, _),
    valid(Sit,Move),
    effects(Sit,Move,NewSit),
    game(NewSit, Result).

terminating(Sit, Sit):-
    holds(Sit,last_move(select(_,X,_))),
    member(X, [refuse,accept,retract,reject]),
    Sit = sit(_, _, Moves),
    writelist([->,Moves],nl).

valid(Game, Move):-
    available(Game, Move),
    legal(Game, Move),
    can(Game, Move),
    \+ cyclic(Game, Move).

cyclic(sit(_,_,Moves), Move):-
    Move = select(paul,challenge,john),
    sublist([select(john,justify,paul),select(paul,challenge,john),
             select(john,justify,paul),select(paul,challenge,john)],
             Moves).
cyclic(sit(_,_,Moves), Move):-
    Move = select(john,authenticate,paul),
    sublist([select(paul,challenge,john),select(john,justify,paul),
             select(paul,challenge,john),select(john,justify,paul)],
             Moves).
cyclic(sit(_,_,Moves), Move):-
    Move = select(paul,challenge,john),
    sublist([select(john,justify,paul),select(paul,certify,john),
             select(paul,certify,john)],
             Moves).
cyclic(sit(_,_,Moves), Move):-
    Move = select(paul,challenge,john),
    sublist([select(john,justify,paul),select(paul,certify,john),
             select(john,notunderstood,paul),select(paul,certify,john),
             select(john,authenticate,paul),select(paul,challenge,john),
             select(john,justify,paul)],
             Moves).
cyclic(sit(_,_,Moves), Move):-
    Move = select(paul,challenge,john),
    sublist([select(john,justify,paul),select(paul,inform,john),
             select(john,notunderstood,paul),select(paul,inform,john),
             select(john,authenticate,paul),select(paul,challenge,john),
             select(john,justify,paul)],
             Moves).
cyclic(sit(_,_,Moves), Move):-
    Move = select(paul,challenge,john),
    sublist([select(john,justify,paul),select(paul,inform,john),
             select(john,notunderstood,paul),select(paul,certify,john)],
             Moves).
cyclic(sit(_,_,Moves), Move):-
    Move = select(paul,challenge,john),
    sublist([select(john,justify,paul),select(paul,certify,john),
             select(john,notunderstood,paul),select(paul,inform,john)],
             Moves).
cyclic(sit(_,_,Moves), Move):-
    Move = select(paul,challenge,john),
    sublist([select(john,justify,paul),select(paul,certify,john),
             select(john,notunderstood,paul),select(paul,certify,john),
             select(john,notunderstood,paul),select(paul,inform,john)],
             Moves).
cyclic(sit(_,_,Moves), Move):-
    Move = select(paul,certify,john),
    sublist([select(john,notunderstood,paul),select(paul,certify,john),
             select(john,notunderstood,paul),select(paul,certify,john)],
             Moves).
cyclic(sit(_,_,Moves), Move):-
    Move = select(paul,inform,john),
    sublist([select(john,notunderstood,paul),select(paul,inform,john),
             select(john,notunderstood,paul),select(paul,inform,john)],
             Moves).
cyclic(sit(_,_,Moves), Move):-
    Move = select(paul,challenge,john),
    sublist([select(john,justify,paul),select(paul,inform,john),
             select(john,authenticate,paul)],Moves).
cyclic(sit(_,_,Moves), Move):-
    Move = select(paul,challenge,john),
    sublist([select(john,justify,paul),select(paul,certify,john),
             select(john,authenticate,paul)],Moves).
cyclic(sit(_,_,Moves), Move):-
    Move = select(paul,certify,john),
    sublist([select(john,notunderstood,paul),select(paul,inform,john),
             select(john,notunderstood,paul)],Moves).
cyclic(sit(_,_,Moves), Move):-
    Move = select(paul,inform,john),
    sublist([select(john,notunderstood,paul),select(paul,inform,john),
             select(john,notunderstood,paul)],Moves).

legal(sit(negotiation,Id,N), select(P1, request,P2)):-
    holds(sit(negotiation,Id,N),role_of(P1,merchant)),
    holds(sit(negotiation,Id,N),role_of(P2,customer)),
    \+ holds(sit(negotiation,Id,N),last_move(_)).

legal(sit(negotiation,Id,N), select(P1,accept,P2)):-
    holds(sit(negotiation,Id,N),role_of(P1,customer)),
    holds(sit(negotiation,Id,N),last_move(select(P2,X,P1))),
    member(X, [request,justify]),
    holds(sit(negotiation,Id,N),role_of(P2,merchant)).
legal(sit(negotiation,Id,N), select(P1,refuse,P2)):-
    holds(sit(negotiation,Id,N),role_of(P1,customer)),
    holds(sit(negotiation,Id,N),last_move(select(P2,X,P1))),
    member(X, [request,justify]),
    holds(sit(negotiation,Id,N),role_of(P2,merchant)).
legal(sit(negotiation,Id,N), select(P1, challenge,P2)):-
    holds(sit(negotiation,Id,N),role_of(P1,customer)),
    holds(sit(negotiation,Id,N),last_move(select(P2,X,P1))),
    holds(sit(negotiation,Id,N),role_of(P2,merchant)),
    member(X, [request, justify]).
legal(sit(negotiation,Id,N), select(P1, retract,P2)):-
```

```
    holds(sit(negotiation,Id,N),role_of(P1,merchant)),
    holds(sit(negotiation,Id,N),last_move(select(P2,challenge,P1))),
    holds(sit(negotiation,Id,N),role_of(P2,customer)).
legal(sit(negotiation,Id,N), select(P1, authenticate,P2)):-
    holds(sit(negotiation,Id,N),role_of(P1,merchant)),
    holds(sit(negotiation,Id,N),last_move(select(P2,challenge,P1))),
    holds(sit(negotiation,Id,N),role_of(P2,customer)).
legal(sit(negotiation,Id,N), select(P1, certify,P2)):-
    holds(sit(negotiation,Id,N),role_of(P1,customer)),
    holds(sit(negotiation,Id,N),last_move(select(P2,X,P1))),
    member(X,[authenticate,notunderstood]),
    holds(sit(negotiation,Id,N),role_of(P2,merchant)).
legal(sit(negotiation,Id,N), select(P1, inform,P2)):-
    holds(sit(negotiation,Id,N),role_of(P1,customer)),
    holds(sit(negotiation,Id,N),last_move(select(P2,X,P1))),
    member(X,[authenticate,notunderstood]),
    holds(sit(negotiation,Id,N),role_of(P2,merchant)).
legal(sit(negotiation,Id,N), select(P1, notunderstood,P2)):-
    holds(sit(negotiation,Id,N),role_of(P1,merchant)),
    holds(sit(negotiation,Id,N),last_move(select(P2,X,P1))),
    member(X,[inform,certify]),
    holds(sit(negotiation,Id,N),role_of(P2,customer)).
legal(sit(negotiation,Id,N), select(P1, reject,P2)):-
    holds(sit(negotiation,Id,N),role_of(P1,merchant)),
    holds(sit(negotiation,Id,N),last_move(select(P2,X,P1))),
    member(X,[inform,certify]),
    holds(sit(negotiation,Id,N),role_of(P2,customer)).
legal(sit(negotiation,Id,N), select(P1, justify,P2)):-
    holds(sit(negotiation,Id,N),role_of(P1,merchant)),
    holds(sit(negotiation,Id,N),last_move(select(P2,X,P1))),
    member(X,[challenge,inform,certify]),
    holds(sit(negotiation,Id,N),role_of(P2,customer)).

sublist([H | T], [H | U]):-
    initialsublist(T, U).
sublist([H | T], [A | U]):-
    sublist([H | T], U).
sublist([], []):-
    false.

initialsublist([], _).
initialsublist([H | T], [H | U]):-
    initialsublist(T, U).

writelist([],nl):-nl,!.
writelist([],sl):-!.
writelist([H|B],L):-
   write(H),
   write(' '),!,
   writelist(B,L).
```

Listing F.1: Situation Calculus Representation of the Order Protocol

Appendix G

Prolog Code in Event Calculus for a protocol with loops

```
:- use_module(library(lists)).
:- use_module(library(system)).

holds(sit(N,Id,Tn,Nn), P):-
    0 =< Tn,
    initially(sit(N,Id,Ti,Ni), P),
    \+ clipped(P, sit(N,Id,Ti,Ni), sit(N,Id,Tn,Nn)).
holds(sit(N,Id,Tn,Nn), P):-
    happens(E, Ti, Ni, Nn),
    Ti < Tn,
    initiates(E, P, sit(N,Id,Ti,Ni)),
    \+ clipped(P, sit(N,Id,Ti,Ni), sit(N,Id,Tn,Nn)).

clipped(P, sit(N,Id,Ti,_), sit(N,Id,Tn,Nn)):-
    happens(Estar, Tj, Nj, Nn),
    Ti < Tj,
    Tj < Tn,
    terminates(Estar, P, sit(N,Id,Tj,Nj)).

happens(at(En,Tn), Tn, [at(En,Tn)|Sn], [at(En, Tn)|Sn]).
happens(at(Ei,Ti), Ti, [at(Ei,Ti)|Si], [at(_, _)|Sn]):-
    happens(at(Ei, Ti), Ti, [at(Ei,Ti)|Si], Sn).
happens(E, Tn, [at(En, Tn)|Sn], [at(En, Tn)|Sn]):-
    member(E, En).
happens(E, Ti, [at(Ei,Ti)|Si], [at(_, _)|Sn]):-
    happens(E, Ti, [at(Ei,Ti)|Si], Sn).

effects(sit(Name,Id,T,N),at(Es,T), sit(Name,Id,NewT,[at(Es,T)|N])):-
    T >= 0,
    NewT is T + 1.

game(Sit, Result):-
    terminating(Sit, Result),
    !.
game(Sit, Result):-
    \+ terminating(Sit, _),
    Sit = sit(_,_,_,Nar),
    assume(valid(Sit, Move), Sit, Move, Episode),
    effects(Sit, Episode, NewSit),
    game(NewSit, Result).

assume(Valid, sit(N,Id,T,Ns), E, at(Es,T)):-
    findall(E, Valid, All),
    member(Es, All),
    acceptable(sit(N,Id,T,Ns),at(Es,T)).

acceptable(Sit,at(Es,T)):-
    \+ cyclic(Sit,at(Es,T)).

cyclic_pattern([at(select(john,justify,paul),_),
                at(select(paul,challenge,john),_),
                at(select(john,justify,paul),_),
                at(select(paul,challenge,john),_)],
                select(paul,challenge,john)).
cyclic_pattern([at(select(paul,challenge,john),_),
                at(select(john,justify,paul),_),
                at(select(paul,challenge,john),_),
                at(select(john,justify,paul),_)],
                select(john,authenticate,paul)).
cyclic_pattern([at(select(john,justify,paul),_),
                at(select(paul,certify,john),_),
                at(select(paul,certify,john),_)],
                select(paul,challenge,john)).
cyclic_pattern([at(select(john,justify,paul),_),
                at(select(paul,certify,john),_),
                at(select(john,notunderstood,paul),_),
                at(select(paul,certify,john),_),
                at(select(john,authenticate,paul),_),
                at(select(paul,challenge,john),_),
                at(select(john,justify,paul),_)],
                select(paul,challenge,john)).
cyclic_pattern([at(select(john,justify,paul),_),
                at(select(paul,inform,john),_),
                at(select(john,notunderstood,paul),_),
                at(select(paul,inform,john),_),
                at(select(john,authenticate,paul),_),
                at(select(paul,challenge,john),_),
                at(select(john,justify,paul),_)],
                select(paul,challenge,john)).
cyclic_pattern([at(select(john,justify,paul),_),
                at(select(paul,inform,john),_),
                at(select(john,notunderstood,paul),_),
                at(select(paul,certify,john),_)],
                select(paul,challenge,john)).
cyclic_pattern([at(select(john,justify,paul),_),
                at(select(paul,certify,john),_),
                at(select(john,notunderstood,paul),_),
                at(select(paul,inform,john),_)],
                select(paul,challenge,john)).
cyclic_pattern([at(select(john,justify,paul),_),
                at(select(paul,certify,john),_),
                at(select(john,notunderstood,paul),_),
                at(select(paul,certify,john),_),
                at(select(john,notunderstood,paul),_),
                at(select(paul,inform,john),_)],
                select(paul,challenge,john)).

```

```
cyclic_pattern([at(select(john,notunderstood,paul),_),
                at(select(paul,certify,john),_),
                at(select(john,notunderstood,paul),_),
                at(select(paul,certify,john),_)],
                select(paul,certify,john)).
cyclic_pattern([at(select(john,notunderstood,paul),_),
                at(select(paul,inform,john),_),
                at(select(john,notunderstood,paul),_),
                at(select(paul,inform,john),_)],
                select(paul,inform,john)).
cyclic_pattern([at(select(john,justify,paul),_),
                at(select(paul,inform,john),_),
                at(select(john,authenticate,paul),_)],
                select(paul,challenge,john)).
cyclic_pattern([at(select(john,justify,paul),_),
                at(select(paul,certify,john),_),
                at(select(john,authenticate,paul),_)],
                select(paul,challenge,john)).
cyclic_pattern([at(select(john,notunderstood,paul),_),
                at(select(paul,inform,john),_),
                at(select(john,notunderstood,paul),_)],
                select(paul,certify,john)).
cyclic_pattern([at(select(john,notunderstood,paul),_),
                at(select(paul,inform,john),_),
                at(select(john,notunderstood,paul),_)],
                select(paul,inform,john)).

cyclic(sit(_,_,_,Ns),at(Es,T)):-
    cyclic_pattern(CyclicPattern,Es),
    sublist(CyclicPattern, Ns).

valid(sit(N,Id,T,Es), M):-
    available(sit(N,Id,T,Es), M),
    legal(sit(N,Id,T,Es), M),
    can(sit(N,Id,T,Es), M).

initially(sit(negotiation,s0,0,[]), role_of(john, merchant)).
initially(sit(negotiation,s0,0,[]), role_of(paul, customer)).

terminating(sit(negotiation,Id,T,N), sit(negotiation, Id, T, N)):-
    holds(sit(negotiation,Id,T,N), last_moves([select(P1,X,_)])),
    holds(sit(negotiation,Id,T,N), role_of(P1, merchant)),
    member(X, [retract,reject]),
    writelist([->,N],nl).
terminating(sit(negotiation,Id,T,N), sit(negotiation, Id, T, N)):-
    holds(sit(negotiation,Id,T,N), last_moves([select(P1,X,_)])),
    holds(sit(negotiation,Id,T,N), role_of(P1, customer)),
    member(X, [accept,refuse]),
    writelist([->,N],nl).

initiates(at(Es, T),  last_moves([Es]), sit(negotiation,_,T,_)).
terminates(at(_, T), last_moves(_), sit(negotiation,_,T,_)).

available(sit(negotiation,_,_,_),select(_,request,_)).
available(sit(negotiation,_,_,_),select(_,accept,_)).
available(sit(negotiation,_,_,_),select(_,refuse,_)).
available(sit(negotiation,_,_,_),select(_,justify,_)).
available(sit(negotiation,_,_,_),select(_,challenge,_)).
available(sit(negotiation,_,_,_),select(_,retract,_)).
available(sit(negotiation,_,_,_),select(_,authenticate,_)).
available(sit(negotiation,_,_,_),select(_,inform,_)).
available(sit(negotiation,_,_,_),select(_,certify,_)).
available(sit(negotiation,_,_,_),select(_,notunderstood,_)).
available(sit(negotiation,_,_,_),select(_,reject,_)).

can(sit(negotiation,_,_,_), select(john,request,_)).
can(sit(negotiation,_,_,_), select(john,justify,_)).
can(sit(negotiation,_,_,_), select(john,retract,_)).
can(sit(negotiation,_,_,_), select(john,authenticate,_)).
can(sit(negotiation,_,_,_), select(john,notunderstood,_)).
can(sit(negotiation,_,_,_), select(john,reject,_)).
can(sit(negotiation,_,_,_), select(paul,challenge,_)).
can(sit(negotiation,_,_,_), select(paul,accept,_)).
can(sit(negotiation,_,_,_), select(paul,refuse,_)).
can(sit(negotiation,_,_,_), select(paul,inform,_)).
can(sit(negotiation,_,_,_), select(paul,certify,_)).

legal(sit(negotiation,Id,T,N), select(P1, request,P2)):-
    holds(sit(negotiation,Id,T,N),role_of(P1,merchant)),
    holds(sit(negotiation,Id,T,N),role_of(P2,customer)),
    \+ holds(sit(negotiation,Id,T,N),last_moves([_])).
legal(sit(negotiation,Id,T,N), select(P1, accept,P2)):-
    holds(sit(negotiation,Id,T,N),role_of(P1,customer)),
    holds(sit(negotiation,Id,T,N),last_moves([select(P2,X,P1)])),
    holds(sit(negotiation,Id,T,N), role_of(P2,merchant)),
    member(X, [request,justify]).
legal(sit(negotiation,Id,T,N), select(P1, refuse,P2)):-
    holds(sit(negotiation,Id,T,N),role_of(P1,customer)),
    holds(sit(negotiation,Id,T,N),last_moves([select(P2,X,P1)])),
    holds(sit(negotiation,Id,T,N), role_of(P2,merchant)),
    member(X, [request,justify]).
legal(sit(negotiation,Id,T,N), select(P1, justify,P2)):-
    holds(sit(negotiation,Id,T,N),role_of(P1,merchant)),
    holds(sit(negotiation,Id,T,N),last_moves([select(P2,X,P1)])),
    holds(sit(negotiation,Id,T,N), role_of(P2,customer)),
    member(X, [inform,certify,challenge]).
legal(sit(negotiation,Id,T,N), select(P1, challenge,P2)):-
    holds(sit(negotiation,Id,T,N),role_of(P1,customer)),
    holds(sit(negotiation,Id,T,N),last_moves([select(P2,X,P1)])),
    holds(sit(negotiation,Id,T,N), role_of(P2,merchant)),
    member(X, [request,justify]).
legal(sit(negotiation,Id,T,N), select(P1, retract,P2)):-
    holds(sit(negotiation,Id,T,N),role_of(P1,merchant)),
    holds(sit(negotiation,Id,T,N),last_moves([select(P2,X,P1)])),
    holds(sit(negotiation,Id,T,N), role_of(P2,customer)),
    X = challenge.
legal(sit(negotiation,Id,T,N), select(P1, authenticate,P2)):-
    holds(sit(negotiation,Id,T,N),role_of(P1,merchant)),
    holds(sit(negotiation,Id,T,N),last_moves([select(P2,X,P1)])),
    holds(sit(negotiation,Id,T,N), role_of(P2,customer)),
    X = challenge.
legal(sit(negotiation,Id,T,N), select(P1, certify,P2)):-
    holds(sit(negotiation,Id,T,N),role_of(P1,customer)),
    holds(sit(negotiation,Id,T,N),last_moves([select(P2,X,P1)])),
    holds(sit(negotiation,Id,T,N), role_of(P2,merchant)),
    member(X, [authenticate, notunderstood]).
legal(sit(negotiation,Id,T,N), select(P1, inform,P2)):-
    holds(sit(negotiation,Id,T,N),role_of(P1,customer)),
    holds(sit(negotiation,Id,T,N),last_moves([select(P2,X,P1)])),
    holds(sit(negotiation,Id,T,N), role_of(P2,merchant)),
    member(X, [authenticate, notunderstood]).

```

```
legal(sit(negotiation,Id,T,N), select(P1, notunderstood,P2)):-
    holds(sit(negotiation,Id,T,N),role_of(P1,merchant)),
    holds(sit(negotiation,Id,T,N),last_moves([select(P2,X,P1)])),
    holds(sit(negotiation,Id,T,N), role_of(P2,customer)),
    member(X, [inform, certify]).
legal(sit(negotiation,Id,T,N), select(P1, reject,P2)):-
    holds(sit(negotiation,Id,T,N),role_of(P1,merchant)),
    holds(sit(negotiation,Id,T,N),last_moves([select(P2,X,P1)])),
    holds(sit(negotiation,Id,T,N), role_of(P2,customer)),
    member(X, [inform, certify]).

writelist([],nl):-nl,!.
writelist([],sl):-!.
writelist([H|B],L):-
    write(H),
    write(' '),!,
    writelist(B,L).

sublist([H | T], [A |U]):-
    sublist([H | T],U).
sublist([], []):-
    false.
sublist([H | T], [H | U]):-
    initialsublist(T, U).

initialsublist([], _).
initialsublist([H | T], [H | U]):-
    initialsublist(T, U).

check_list([H | T], [H1 | T1]):-
    check_list_one(H, H1),
    check_list(T, T1).
check_list([H | T], [H1 | T1]):-
    check_list([H | T], T1),
    !.
check_list([],_):- !.

check_list_one(A, B ):-
    A = B.
```

Listing G.1: Loops Protocol in Situation Calculus