Runtime Verification of Deontic and Trust Models in Multiagent Interactions

Nardine Zoulfikar Osman

Doctor of Philosophy

Centre for Intelligent Systems and their Applications
School of Informatics
University of Edinburgh

2008

Abstract

In distributed open systems, such as multiagent systems, new interactions are constantly appearing and new agents are continuously joining or leaving. It is unrealistic to expect agents to automatically trust new interactions. It is also unrealistic to expect agents to refer to their users for help every time a new interaction is encountered. An agent should decide for itself whether a specific interaction with a given group of agents is suitable or not. This thesis presents a runtime verification mechanism for addressing this problem.

Verifying multiagent systems has its challenges. It is hard to predict the reliability of interactions, in systems that are heavily influenced by autonomous agents, without having access to the agent specifications. Available verification mechanisms may roughly be divided into two categories: (1) those that verify *interaction models* independently of specific agents, and (2) those that verify *agent models* whose constraints shape the interactions. Interaction models are not sufficient when verifying dynamic properties that depend on the agents engaged in an interaction. On the other hand, verifying agent specifications, such as BDI models, is extremely inefficient. Specifications are usually not explicit enough, resulting in the verification of a massive number of permissible interactions. Furthermore, in open systems, an agent's internal specification is usually not accessible for many reasons, including security and privacy.

This thesis proposes a model checker that verifies a combination of a global interaction model and local *deontic models*. The deontic model may be viewed as a list of agent constraints that are deemed necessary to share and verify, such as the inability of the buyer to pay by credit card. The result is a lightweight, efficient, and powerful model checker that is capable of verifying rich properties of multiagent systems without the need for accessing agents' internal specifications.

Although the proposed model checker has potential for addressing a variety of problems, the trust domain receives special attention due to the critically of the trust issue in distributed open systems and the lack of reliable trust solutions. The thesis illustrates how a dynamic model checker, using deontic/trust models, can help agents decide whether the scenarios they wish to join are trustworthy or not.

In summary, the main contribution of this research is in introducing *interaction time verification* for checking deontic and trust models multiagent interactions. When faced with new unexplored interactions, agents can verify whether joining a given interaction with a given set of collaborating agents would violate any of its constraints.

Acknowledgements

The single most influential source in shaping this thesis has been my supervisor Dave Robertson, to whom I am deeply grateful. He has been a great mentor and a main source of inspiration throughout my postgraduate years. His time, support, and encouragement, especially during the last and toughest months of writing this thesis, are greatly appreciated.

I am thankful to all my fellow CISA members, especially Michael Rovatsos, whose notes and comments sparked several interesting ideas in this thesis, and ex-CISA member Chris Walton for his co-supervision during my early years of PhD.

I am also thankful to Carles Sierra for his review and valuable comments on one of my early papers. I am especially grateful for his support during the last few months of writing this thesis.

The effort of George (Big G., my very first friend in Edinburgh) in reviewing a section of this thesis, while busy preparing for Clara Maria's birth, is greatly appreciated.

Were it not for my friends in Edinburgh, my stay in this city would have been as gloomy as its weather. I would like to thank Khodor, Sami, Teresa, Bassel, Ana, Sandy, Jorge (and others that I surely missed), for their sincere friendship and for making my stay in Edinburgh fun and exciting.

However, there is nothing like the support of family members. I am thankful to my siblings: Zanoubia, Rawand, and Mohammad Hussein. Each visit, each phone call, and each email has helped lift my spirit during the last three years. Our religious/philosophical discussions over the email provided an interesting and stimulating break for my mind. However, I am specially thankful to Rawand for having to put up with my numerous phone calls and inane questions.

My deepest gratitude goes to my cherished life companion Rida, who has not only managed to keep our long distance relationship burdenless, but cleverly turned such a potential emotional burden into a tremendous emotional support. He has been there for me every time I needed him. I would like to thank him for his patience, support, love, and respect.

Last, but not least, I will forever be indebted to those who have helped me be where I am today, my beloved Father and Mother, Zoulfikar and Wafaa. Their encouragement and support throughout the years, both financially and emotionally, have been unpreceded. Having them next to me has been a blessing.

A sincere 'thank you' to you all.

Declaration

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

(*Nardine Zoulfikar Osman*)

To those with pure hearts, clear minds, and blessed deeds,
To my beloved Mother and Father

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Computer systems have once been viewed as isolated dedicated systems. Nowadays, an isolated system is considered to be a powerless system with limited capabilities. The growth of the Internet has helped promote interconnection, resulting in the growth of distributed and concurrent systems. Hoping to achieve more intelligent systems, the latest trend is to move towards open distributed systems, such as multiagent systems[1]. However, the increased complexity of these open systems, which now rely on interacting autonomous agents, raises a crucial question on the verification issue: *"How can traditional verification mechanisms be applied to open distributed systems?"*

Formal verification is a well established field that has proven to be highly successful when applied to both hardware and software systems. However, this success has been limited when applied to multiagent systems. The reason is the dynamic nature of these systems. Traditionally verified systems, whether hardware or software, are static systems whose specifications are fixed and well defined. Multiagent systems, on the other hand, are highly dynamic. The entities that compose them, such as agents, peers, services, etc., may join or leave the system at anytime. New interactions are continuously appearing. As a result, traditional verification mechanisms seem no longer suitable to be directly applied to these systems.

This thesis proposes a dynamic approach for verification. The result is an automated verifier that may be invoked by the agents at interaction time when the conditions for verification are met. This is achieved by making use of a lightweight, efficient, and fully automated model checker.

[1] Although the research of this thesis is aimed at distributed open systems in general, the term multiagent will be used when referring to such systems to emphasise the autonomy of the different entities composing these systems.

Section 1.1 introduces the challenges of multiagent system verification, in view of the available literature. Section 1.2 presents the proposed approach for overcoming these challenges. Section 7.3 outlines our claims of novelty. Finally, the structure of this thesis is presented in Section 1.5, providing the reader with a glimpse of what is to be expected from each chapter.

1.1 The Challenge

One of the most fundamental issues of multiagent system engineering is to enable predictable and reliable interactions. The challenge is in achieving this without requiring a deep standardisation of the way in which the agents are engineered, yet preserving as much as possible their autonomy in individual reasoning. Two contrasting specification approaches have emerged for addressing this problem:

- **A top-down approach:** In this approach, the focus is on engineering *models of interactions* that may be described in a generic process or state-machine language. These models explicitly specify how the interaction may be carried out by the agents. They are detached from individual agents and are typically accessed when an agent anticipates it wants to initiate or join that type of interaction.

- **A bottom-up approach:** In this approach, the focus is on engineering highly *intelligent agents* in such detail that they become capable of shaping and directing interactions. Each agent specification is built locally and independently of other agents. This is sometimes referred to as the agent's BDI layer, which describes the agent's beliefs, desires, and intentions.

Following in the footsteps of specification, multiagent system verification mechanisms may roughly be divided into similar categories. Chapter 6 provides a thorough presentation of the available literature. However, in what follows, we give a brief overview to these verification mechanisms, which we roughly divide into four categories. In the first (e.g. Wen and Mizoguchi, 1999; Walton, 2004; Huget, 2002; Cliffe and Padget, 2002), the verifier is applied to a global interaction model and does not take into consideration any of the agents involved in such an interaction. In the second (e.g. Benerecetti et al., 1998; Wooldridge et al., 2002; Bordini et al., 2003b), the verifier is applied to a group of agents whose internal specifications are expected to shape the interaction that needs to be verified. In the third (Giordano et al., 2003, 2004), both the shared interaction model and the private agent models are used in verification. In the

fourth (e.g. Penczek and Lomuscio, 2003; Woźna et al., 2005; Kacprzak et al., 2004; Raimondi and Lomuscio, 2007), only the interaction model is verified. However, such interaction models are usually enriched with a variety of domain specific operators in order to verify properties about agents' knowledge, commitments, strategies, etc.

The advantage of the mechanisms of the first category is that the properties verified against a given interaction model are global properties that will hold for any group of agents. The problem, however, is that interactions in multiagent systems are strongly linked to the agents executing them. It is the agents that drive and direct these interactions. The verification of interaction models help ensure the correctness of these interactions by proving some of their properties. This proves to be very helpful in many applications, such as mechanism design. However, these properties are static properties of the interaction. When trying to prove whether a given interaction would succeed with a given group of collaborating agents, properties marking this success may be dynamic (by depending on the agents engaged in this given interaction). Verification mechanisms of the first category are limited to the class of static and agent independent properties. These usually describe liveness and safety properties of interactions, such as checking for deadlocks, reachability of states, etc.

In the second category, the verifier is applied to the agents' internal specification, or their BDI layer. These approaches tend to verify a much more interesting set of properties than those tackled by the verifiers of the first category. For example, one may verify properties about the change of agents beliefs within interactions. However, these verification mechanisms raise a serious concern by assuming that: (1) the internal specifications of agents are accessible by the verifier, and (2) these specifications are comprehensible to the verifier. Both of these assumptions are unrealistic and stand against the very nature of distributed open systems. In open systems, agents are not expected to allow access to their BDI layer, mainly for security, trust, and privacy issues. For instance, why would a bidding agent want to reveal its bidding strategy? Furthermore, even if an agent did permit some verifier to access its internal specification, this information is most likely incomprehensible to the verifier. The information required by the verifier is usually not specified declaratively, but rather implicitly in the procedural implementation of the agent. Also, agents in distributed systems are not expected to be engineered all in the same way. Each agent is built independently, most probably following different implementation designs and languages. One may argue that in closed systems, such strong assumptions could hold. This is true. However, even then, the agent constraints should be explicit enough to be able to specify and direct

the interaction. Usually, these techniques result in a massive number of permissible interactions, which raises severe efficiency issues. This leads model checkers to hit the state-space explosion problem fairly quickly.

The literature of the third category has simply combined both mechanisms of the first and second categories. The verifier can either prove properties of the interaction, disregarding the agents involved, or require internal agent specifications to prove that agents can actually perform the actions they are expected to perform.

Mechanisms of the fourth category have the ability to verify compelling properties of the agents without any need for accessing the agents' internal specifications. However, we believe the verification process of such mechanisms are overly complicated and inadequate for fully automated verification. For instance, the system model is not specified through a clear process calculus, but through a collection of states that are specified by hand. Also, transition functions should usually be specified to link the states of the system's state space together. Of course, this is not an issue when verifying the system offline. However, this complication in both the specification and verification process raises lots of concerns (including efficiency concerns) when verification needs to be performed by the agents at interaction time, as this thesis proposes.

In summary, verification in the field of multiagent systems has been suffering from various problems. Some verifiers lack the capability of verifying worthwhile properties of these dynamic systems. Others have struggled from severe efficiency issues. Many fell into the trap of over complicating the specification and verification process, by continuously adding new modalities to their specification languages for every new notion to be verified. We believe that the majority of these issues are directly linked to the main challenge presented above, and that may be described as the "interaction versus agent model" dilemma. The proposed verification mechanism of this thesis is aimed primarily at addressing this dilemma. The following section provides a gentle introduction to this proposed mechanism.

1.2 The Proposed Approach

Put differently, the "interaction versus agent model" dilemma of multiagent system verification may be redefined by the following question: *"How can a verifier prove compelling properties of interactions without relying on the agents specification?"* The verification mechanism of this thesis addresses this issue by distinguishing between an agent's internal specification and its deontic one. The deontic model is said to describe

agents' general permissions, prohibitions, and obligations. While an agent's internal specification cannot and should not be made public, agents might need to make some of their deontic constraints public. For example, no bidder would be interested in revealing its bidding strategy, yet it would be highly useful for both the bidder and the auctioneer to reveal that the bidder can only pay via *PayPal*.

Using this combination of interaction and deontic models implies that the verification mechanism can be more efficient than those that solely verify agent models. This is because the explicit models of interactions will considerably limit the interaction's state-space. Furthermore, the entire verification process is much more realistic. This is because it does not rely on the agents' internal specification, but on their deontic constraints that are made public by the agents and are believed to be crucial for the interaction. Moreover, the addition of this deontic layer enriches the properties that may be verified, leading to more predictable and reliable results. This is because the verifier does not completely neglect the agents involved and their effects on interactions. However, this combination of global interaction and local deontic models implies that verification will need to be performed at interaction time, when the conditions for verification are met and the related deontic models are obtained.

Interaction time verification is key. This requires a fully automated verification mechanism that is achieved by making use of automated model checking. In these highly dynamic systems, the automated verifier may be invoked by the agents at run time to help them decide whether the new interaction models they wish to join and the group of collaborating agents they wish to interact with are suitable or not. As an example, the verifier is applied to the field of trust. The goal is to let the agents verify whether the scenarios they are about to join could break any of their trust constraints.

1.3 Claims of Novelty

In summary, with the above proposed approach, this thesis presents two major novel contributions. First, it combines global interaction models to local deontic ones, resulting in a new system model architecture. It does this to address the interaction versus agent model dilemma. The goal is to be able to verify dynamic properties (that usually rely on the dynamic groups of agents engaged in an interaction) without the need for internal agent specifications, resulting in a verification mechanism suitable for open and distributed systems. Second, it introduces interaction time verification for multi-agent systems. For this to succeed, it is crucial that the verifier be fully automated,

lightweight, and efficient. The remaining chapters of this thesis illustrate how this may be accomplished.

1.4 Requirements and Assumptions

Several assumptions, in terms of both representation and computing resources, are made in this thesis for the proposed verification mechanism to succeed. In what follows, we give a list of these assumptions:

- The model checker of this thesis is intended to be used by agents in order to verify whether a given interaction model along with a given group of collaborating agents is suitable or trustworthy. We assume that the agent can search for available interaction models, possibly using discovery services (as illustrated by the OpenKnowledge project (Siebes et al., 2007)). As for the deontic and trust constraints, we currently assume agents to know the constraints they need to verify, along with the specific instantiations these constraints require. For example, if a specific auctioneer agent needs to verify whether the interaction protocol encourages truth telling on the bidders, the auctioneer should verify this trust constraint against an interaction model instantiated for one auctioneer agent and two bidder agents. Chapter 4 elaborates more on this issue.

- The input to the model checker should be specified in the languages of the proposed model checker. For instance, interaction models need to be specified in LCC (Section 2.2), deontic constraints in DPL (Section 3.4.2), trust constraints in TPL (Section 4.5), and temporal properties in the μ-calculus (Section 2.3.3). However, an interesting aspect of these languages is there basic and simple nature. For instance, LCC may be viewed as a basic process calculus (Section 2.2.5). The μ-calculus is a common and traditional temporal logic. DPL may be described as a policy language with a basic syntax defined by only two predicates: *must* and *can*. Similarly, TPL is a trust policy language with a basic syntax defined by the *trust* predicate, which is used for specifying whether a given interaction model or a given agent is trusted or not. Naturally, we assume agents to share the vocab of these languages. However, the vocab used in the constraints of these languages does not need to be shared; when necessary, matching or ontology mapping technologies could be used.

- For agents to be able to carry out the verification process, our proposed lightweight model checker, MCID, should be installed. The model checker is compact and written in less than 200 lines of Prolog code. However, to ensure termination, the model checker is built on top of the XSB tabled Prolog system (Sagonas et al., 1994). The XSB system requires a bit less than 20MB to install. Although we believe it is safe to assume that the majority of agents will have this commodity, agents that do not have the MCID and XSB systems installed may make use of model checking web services that receive verification requests and return the results of the verification process. Of course, this raises serious trust concerns. In such cases, traditional reputation mechanisms may be used for selecting the appropriate web service.

1.5 Thesis Structure

The rest of this thesis is divided as follows:

- Chapter 2 provides the background needed to attain a better understanding of the verification mechanism presented in Chapter 3. It provides an introduction to formal verification in general, and model checking in particular. It also provides the background needed for understanding the various specification languages of the proposed model checker.

- Chapter 3 introduces the proposed verification mechanism, resulting in the MCID model checker. The work presented in this chapter is largely based on our published papers in both the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems (Osman et al., 2006b) and the Fourth International Workshop on Declarative Agent Languages and Technologies (Osman et al., 2006a).

- Chapter 4 illustrates how the MCID system may be applied to the verification of trust in multiagent systems. The work presented in this chapter is largely based on our published papers in both the Twentieth International Joint Conference on Artificial Intelligence (Osman and Robertson, 2007) and the workshop on Trust in E-systems and the Grid (Osman, 2007).

- Chapter 5 demonstrates how the MCID model checker may be used in two different ways. In the first, the model checker is invoked offline from outside an

interaction model in order to verify the model, along with any other potential model the first model might link to. In the second, verification is performed at interaction time by an agent, and the model checker is invoked from within an interaction model to verify (at run time) potential models that may be executed from within the first model.

- ✦ Chapter 6 presents an overview of the available literature on multiagent system verification. This is followed by an analysis of the various mechanisms in comparison with the proposed mechanism of this thesis. Although it is a bit unusual to present the literature review towards the end of a thesis, we choose to do this because we find it more coherent to have one chapter presenting both the literature review along with a thorough analysis and comparison with our work. Furthermore, such analysis and comparison can only be fully understood after gaining an adequate understanding of our work.

- ✦ Chapter 7 draws the final conclusions and sets the plan for future work.

Chapter 2

Background

This thesis focuses on the verification of distributed open systems in general, and multiagent systems in particular. We propose a fully automated verification technique that may be carried out by the agents when needed. This could be used to aid autonomous agents in their decision process when answering questions such as: Should I engage in a given interaction? Whom should I interact with? How should I interact with them? etc. Formal verification mechanisms may be divided into two main categories: interactive deductive reasoning (or theorem proving) and automated methods (or model checking). In the first, although certain degrees of automation may exist, the verification is generally driven by the user and his understanding of the inference rules and the system being verified. In the second, the model checker is fed the system to be verified and the properties it is verified against, and the entire verification process is fully automated.

In this thesis, we choose model checking as our verification method due to its full automation. A model checker typically takes in a system model and a property specification as its input; it then proves whether the specified properties are satisfied in the given system model. However, for our system, we suggest the addition of yet a third input: the agent's deontic constraints. We believe that the verification of interactions, which heavily depend on the autonomous parties engaged in these interactions, should take into consideration the various parties' constraints.

This chapter aims at providing the background needed for understanding the technical details of this thesis: our model checking technique and its formal specification languages. Section 2.1 provides a general overview of the various verification techniques, focusing on model checking in particular. This is followed by a presentation of the lightweight coordination calculus (LCC), the process calculus used by our

model checker for the specification of multiagent systems, in Section 2.2. Section 2.3 introduces the μ-calculus temporal logic, our model checker's property specification language. Finally, Section 2.4 provides an overview of deontic logic and policy languages, which is crucial for understanding our proposed deontic based policy language of Section 3.4.2.

2.1 Formal Verification and Model Checking

This section starts with an overview of available verification methods in Section 2.1.1, before introducing the technical details of model checking in Section 2.1.2.

2.1.1 Verification Methods: an Overview

Verification methods are usually divided into two main categories: functional (or black-box) testing and formal verification. Figure 2.1 provides a categorisation of the most common verification methods. Functional testing usually disregards the internal structure and mechanism of a system and focuses solely on observing the output when providing certain valid or invalid input. The two most common techniques in this category are simulation and testing (Edmund M. Clarke et al., 1999). While simulation is performed on the model of a system, testing is performed on the actual system. Especially in software development, simulation and testing are the most widely used techniques for verification today. This is basically due to their cost-efficient approaches for discovering many errors. However, the problem with functional testing is that there is no guarantee that all the errors are discovered and that the system will behave as intended. Of course, this raises an alarm in more critical systems. For example, on the 4th of June, 1996, the Ariane 5 rocket exploded due to a software error: the 64-bit floating point number was too large to fit into a 16-bit signed integer, causing an operand error. To ensure the correctness of such critical systems, functional testing is clearly not a sufficient verification method. There certainly is a need for more formal methods that could prove the correctness of these systems.

Formal verification methods rely on mathematical proofs for verification. These methods may roughly be divided into two main categories: deductive reasoning and model checking. Verification based on deductive reasoning, such as theorem proving, aims at proving the correctness of a system using axioms and proof rules. Deductive reasoning is usually performed by an expert in logical reasoning, whose task is to

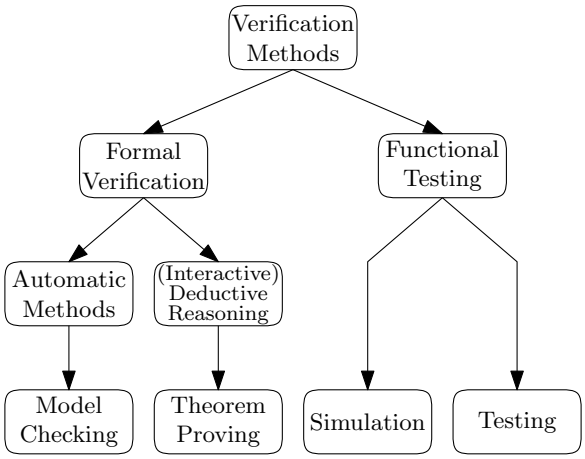


Figure 2.1: Verification methods

construct proofs manually, using interactive software tools that would aid the progress of the proof. However, one single proof may take days or even months to complete. For this reason, deductive reasoning has not been as widely used as other verification methods. Another disadvantage of deductive reasoning is that the automated reasoning process cannot always be guaranteed to terminate. This limits the properties that could be verified automatically. Nevertheless, despite the limitation on what could automatically be checked and the huge amount of resources that might be used to find a proof, a strong aspect of deductive reasoning is its capability to reason over systems with an infinite state, which is usually a limitation in model checking approaches.

Model checking, which is explored in the following section, limits the systems it can verify to those with a finite state-space. However, this limitation makes model checking a fully automated verification technique that is guaranteed to terminate with a yes/no answer. Because of its full automation, and despite its limitation to finite state systems, model checking is usually preferred to deductive reasoning. When needed, systems with an infinite space may sometimes be verified by restricting unbounded data structures to specific sets, or applying some abstraction and induction techniques.

Nevertheless, deductive reasoning remains the preferred technique for the verification of sensitive systems, such as security protocols. Moreover, there will always be examples of properties of infinite systems that cannot be verified via model checking, but by deductive reasoning. Interestingly, some techniques (e.g. Rajan et al., 1995) have even embedded deductive reasoners with model checkers. The model checker is then used to automatically verify the finite state parts of the system in question.

2.1.2 Model Checking

2.1.2.1 The Model Checking Process

The model checking problem is generally defined as follows: *Given a system S and a formula ϕ, does S satisfy ϕ?* As a result, the model checking process, illustrated in Figure 2.2, is divided into three steps: *modelling* the system, *specifying* the properties, and *verifying* that the system model satisfies the properties specification. In what follows, we discuss the details and the challenges of each of these three stages:

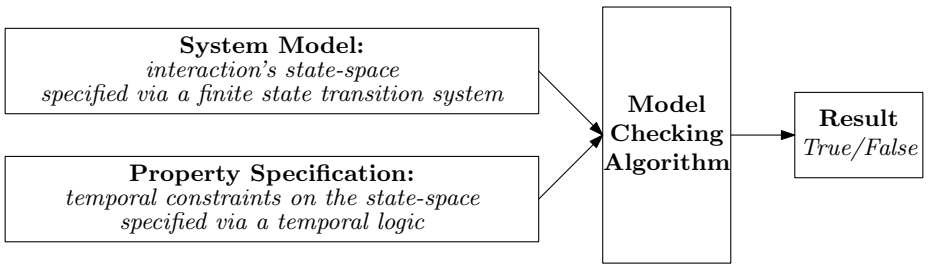


Figure 2.2: The model checking process

1. **Modelling:** The system to be verified must first be modelled in the language of the model checker. It is a language, often a process calculus, that presents the system as a finite state transition system. A finite state transition system[1] is usually presented via a state transition graph (or simply a state graph), and is a type of a non-deterministic finite state machine. In this thesis, the language that specifies the system model is the LCC process calculus, which is introduced later on in Section 2.2.

   The challenge of modelling systems depends not only on deciding what aspects of the system are to be verified, but also on the complexity of the system, the expressiveness of the description language, and the difficulty of mapping between the two. The abstraction of the system should make sure that essential factors are preserved while irrelevant features, which could unnecessarily complicate the verification process, are omitted.

2. **Specification:** After modelling the system, one should specify the properties to be verified in the system. Usually, these are properties of correctness, safety,

[1]Note that state transition systems, in general, need not be finite; however, model checking usually focuses on finite state systems only.

liveness, etc. Traditionally, these properties have been specified via some temporal (or modal) logic. Temporal properties distinguish themselves from other logics by introducing temporal features to the properties describing the behaviour of a system. Safety and liveness properties can then be specified by using temporal operators of the form *it will never be true*, *it will eventually be true*, etc. The model checker of this thesis uses the μ-calculus temporal logic for property specification. The μ-calculus is introduced in Section 2.3.

The specification stage is tricky and has its challenges. Formal verification is usually preferred to functional testing when there is a need to formally prove the correctness of the system and rule out any possible errors. However, how do we know if the property specification list is complete? That is, how do we know that the specification describes *all* the properties that the system should satisfy? This remains an open issue.

3. **Verification:** After modelling the system and specifying the properties that should be satisfied by that system, both the system model S and the property specification ϕ are fed to the model checker for verification. The model checker is essentially an algorithm that decides whether the system S satisfies the property ϕ, usually expressed as $S \models \phi$. Some model checkers also provide a counter example, or a trace, when the property is not satisfied.

   The model checking algorithm of this thesis constitutes the basis of our research. Our model checking approach and algorithm is presented in Chapter 3. However, the remainder of this section provides an overview of model checking techniques, their complications, and the different approaches for dealing with such complications.

   Note that the verification stage has also got its share of challenges. Allegedly, model checking is a fully automatic verification mechanism. In practise, however, the model checker may return false results due to incorrect modelling or even incorrect property specification. An error trace is usually helpful in fixing these common problems. Also, in theory, a model checker should always terminate with a yes/no answer. In practise, however, it is common for a model checker to fail in returning a yes/no result. This usually happens when the size of the model is too large to be handled by the available memory resources, known in model checking as the state-space explosion problem. Section 2.1.2.3 addresses this problem.

2.1.2.2 Model Checking Techniques: Local versus Global

There are two main techniques by which model checking may be performed (Müller-Olm et al., 1999):

Global Model Checking: Given a finite system model S and a formula ϕ, determine the set of states in S that satisfy ϕ.

Local Model Checking: Given a finite system model S, a formula ϕ, and a state s_i in S, determine whether s_i satisfies ϕ.

In global model checking, the property is verified against each state of the system. The system is said to satisfy the property if its initial states are all elements of the set of states satisfying the property. In local model checking, the model checker attempts to verify whether the initial state(s) of the system, usually indicated as s_0, satisfies property ϕ. To do this, the model checker looks at the immediate neighbouring states of s_0, as required by the formula, and so on. However, instead of traversing the entire state-space, the model checker terminates when a (dis-)satisfaction of ϕ is established. For example, if the property states that no *pay_money* action should ever occur in the system, and the first transition s_0 performs towards it neighbouring state is *pay_money*, then the model checker terminates with a negative result after inspecting only the first state.

2.1.2.3 The State-Space Explosion Problem

The state-space explosion problem is defined as the problem of running out of resources when trying to verify a property against a massive state-space. Most model checkers have different techniques implemented for dealing with the state-space explosion problem.

Some approaches that deal with this problem focus on building only part of the state-space. For example, sometimes extra branches leading to the same state are created in a state-space by simply rearranging the order in which actions are executed. Partial order reduction techniques aim at reducing the state-space by reducing those wasteful branches. Abstraction techniques are used to replace a system by a simpler one in which many details, irrelevant to the property to be verified, are dropped out.

Symbolic approaches are another way of dealing with the state-space explosion problem. The idea is to use implicit, rather than explicit, representation of the states and transitions, such as the use of binary decision diagrams (BDD). Temporal properties are then verified against a BDD representation rather than the huge state-space.

2.2 Lightweight Coordination Calculus (LCC)

As illustrated in Section 2.1.2.1, a state transition system is typically used by model checkers for describing the system model. However, there are different types of state transition systems. The most common are Kripke structures (Kripke, 1963) and Labelled transition systems (Milner, 1989). A Kripke structure is a tuple (S, R, L), where S is the set of states, R is a transition relation $R \subseteq S \times S$ that defines the transitions between states, and L is a labelling function $L : S \rightarrow 2^{AP}$, where AP is a set of atomic propositions that represent basic local properties of system states. When model checking Kripke structures, the sets S and AP are usually finite. On the other hand, a labelled transition system (LTS) is a tuple $(S, A, \rightarrow)$, where S is a set of states, A is a set of actions (or labels), and $\rightarrow$ is a transition relation $\rightarrow: S \times A \times S$ that defines the transitions between states. Usually, a transition $(s, a, s') \in \rightarrow$ is presented as $s \xrightarrow{a} s'$. When model checking labelled transition systems, the sets S and A should be finite.

Note that while Kripke systems use labels to associate states with sets of atomic propositions, labelled transition systems use labels to associate transitions with actions. The combination of both, which makes use of labels attached to both states and transitions, have been described in the literature as labelled kripke structures (LKS) (Chaki et al., 2005), doubly labelled transition systems (DLTS) (Nicola and Vaandrager, 1995), Kripke modal transition systems (Kripke MKS) (Huth et al., 2001), etc.

In this thesis, the system model is represented as a labelled transition system. Labelled transition systems are usually specified via a process calculus. Our model checker's specification language is also a process calculus: the lightweight coordination calculus (LCC). While we explain our choice for using LCC in Chapter 3, this section provides an introduction to the lightweight coordination calculus (LCC). LCC is introduced in Section 2.2.1, its syntax and semantics are presented in Section 2.2.2, the process of executing LCC interactions is illustrated in Section 2.2.3, the goal of achieving coordination via LCC is discussed in Section 2.2.4, and finally, a comparison between LCC and other process calculi is provided by Section 2.2.5.

2.2.1 What is LCC?

LCC is a language, based on logic programming (Robertson, 2004c), that provides means of achieving coordination in distributed systems. It approaches the coordination problem through the enforcement of a strict notion of social norms (Robertson, 2004a). These control what actions different agents can perform, when they can per-

form such actions, under what conditions may these actions be carried out, etc. Of course, these rules are associated with roles rather than physical agents; and agents can play more than one role in more than one interaction. This provides an abstraction for the interaction model from the individual agents that might engage in such an interaction. Nevertheless, agents' autonomy is preserved in a sense that it is up to the agents to decide: (1) whether or not to join an interaction, and (2) in what direction to drive these non-deterministic interaction models. The first case is dealt with by the agent's decision making process, which is outside the scope of the LCC specification. The second is achieved in LCC by making use of constraints whose satisfaction relies on the knowledge of the particular agent playing that role at that specific time.

LCC follows a process calculi approach for the specification of social norms. Each role is associated with a process. Each process definition is a specification of a labelled transition system, which indicates the actions the role can perform: what messages can be received, what messages can be sent, in what order may these messages be transmitted, and under what conditions may these actions occur.

For modelling multiagent systems, many methods have used the concepts of social norms (e.g. Greaves et al., 2000; Esteva et al., 2001, , etc.), several others have used transition graphs and process calculi (e.g. Esteva et al., 2002), logic programming (e.g. Clark and McCabe, 2003), etc. However, to our knowledge, LCC is the only process calculi that is used both in the specification of multiagent systems as well as in the execution of their interactions. For instance, the MAP process calculus has been used by Walton (2004) for the specification and verification of multiagent interactions. The ambient calculus has been used by Esteva et al. (2002) for the specification and verification of electronic institutions. The π-calculus has been used by Jiao and Shi (1999). The ψ-calculus was introduced by Kinny (2002) for agent specifications. Similarly, Hartonas (2003) introduces another process calculus for specifying agents and their workspace. However, unlike LCC, none of these process calculi have been used in the executable model.

Furthermore, LCC provides a lightweight method that has only two basic engineering requirements from agents: (1) to be able to extract the current state of the interaction and the agent's next permissible action(s), and (2) to have an appropriate constraint solver for dealing with LCC constraints. After presenting the LCC syntax and semantics in Section 2.2.2, Sections 2.2.3 and 2.2.4 discuss these two lightweight requirements in more detail.

2.2.2 Syntax and Semantics

The LCC interaction framework is the set of clauses specifying the expected message passing behaviour associated with the various roles of an interaction. Its syntax is presented by Figure 2.3.

$$
\begin{aligned}
Framework &:= \{Clause, \ldots\} \\
Clause &:= Agent :: ADef \\
Agent &:= a(Role, Id) \\
ADef &:= ADef\ then\ ADef \mid ADef\ or\ ADef \mid ADef\ par\ ADef \mid \\
&\quad\ Agent\ [\leftarrow C] \mid Message \mid null\ [\leftarrow C] \\
Message &:= M \Rightarrow Agent\ [\leftarrow C] \mid [C \leftarrow]\ M \Leftarrow Agent \\
C &:= Term \mid C \wedge C \mid C \vee C \\
Role &:= Term \\
M &:= Term
\end{aligned}
$$

where,
null represents an event which does not involve message passing (or a "do nothing" event),
$Term$ is a structured term in Prolog syntax,
Id is either a variable or a unique agent identifier, and
$[X]$ denotes the occurrence of either zero or one instance of X.

Figure 2.3: LCC syntax

A framework, or an interaction model, is composed of a set of clauses. A clause gives each agent role a definition that specifies its acceptable behaviour. LCC agents are defined by their roles and identifiers. When defining interaction models, it is common to specify the role while keeping the agent identifier as a variable. This allows the interaction model to be instantiated by different agents, each running its own instance of the interaction. The agent's behaviour may be defined in terms of three different classes of atomic actions. First, an agent can send or receive messages ($M{\Rightarrow}A$, $M{\Leftarrow}A$), which we sometimes refer to as message passing actions (MPA). Second, an agent can take on a different role. Third, an agent can do nothing, which is usually used when the agent needs to perform internal computations. Internal computations are sometimes referred to as constraints, or non message passing actions (N-MPA). Complex agent behaviour may then be built on top of these three classes of atomic actions using the sequential (*then*), choice (*or*), parallel composition (*par*), and conditional ($\leftarrow$) operators. The conditional operator is used for linking constraints to atomic actions.

LCC Example: We take the following scenario from Robertson (2004b) to illustrate the use of LCC as a specification language for achieving coordination in multiagent systems. The example illustrated in Figure 2.4 presents a scenario in which a workshop organiser, playing the LCC role *organiser*, asks another agent, playing the LCC role *headhunter*, for the appropriate people on that workshop topic. It then locates those people in the network by consulting with an agent playing the role *finder*. Finally, it engages in a discussion with these people in order to achieve a schedule that would suit everyone's time constraints.

When the interaction starts, the organiser needs to know in advance what the workshop subject A is, who the headhunter agent H is, and who the finder agent F is. The interaction starts when the organiser sends a message to the headhunter asking it to inform it of the experts in subject A. The organiser then receives a set $\mathcal{S}$ of the suitable people for subject A. It then takes the role of a locator to obtain the location of those people, followed by the role of a time coordinator to coordinate the workshop's time schedule with respect to the time constraints of these people.

In order to locate people, the organiser (playing the role $locator(F, \mathcal{S}, \mathcal{L})$) sends a message to the finder agent F asking it for the location of one of the people, after which a reply is received specifying the web location of the person in question. A recursion is performed over the entire list of people $\mathcal{S}$ until the web location of all those people have been obtained.

In order to coordinate the proposed schedule with the people involved in the workshop, the organiser (playing the role $time\_coordinator(\mathcal{S}, \mathcal{L}, A, [T|\mathcal{T}], T_w)$) tries to coordinate one proposed schedule after the other. For each proposed schedule T, it takes the role *time_proposer* to ask the people about their availability. This is done recursively over the entire list of people $\mathcal{S}$. If the final result is *ok*, then the *time_coordinator* role would have completed successfully. Otherwise, the *time_coordinator* role tries to repeat the whole process for another proposed schedule of the set $\mathcal{T}$.

The headhunter has a simple role: when a message is received asking about the appropriate people for a given subject, a reply should be sent back with a list of people's names, based on the headhunter's knowledge.

The finder agent also has a simple role: when a message is received asking for the location of the agent associated with some person, a reply is sent back with the URI of that person, based on the finder's knowledge. However, the finder's role is recursive. This is because, in a single run of the interaction, the finder may recursively be asked to find the location of other agents.

$$
\begin{array}{l}
a(organiser(A,H,F),O) :: \\
\quad ask(best\_people(A)) \Rightarrow a(headhunter,H)\ then \\
\quad inform(best\_people(A,\mathcal{S})) \Leftarrow a(headhunter,H)\ then \\
\quad a(locator(F,\mathcal{S},\mathcal{L}),O)\ then \\
\quad a(time\_coordinator(\mathcal{S},\mathcal{L},A,\mathcal{T},T_w),O) \leftarrow times(\mathcal{T}).
\end{array}
$$

$$
\begin{array}{l}
a(locator(F,\mathcal{S},\mathcal{L}),L) :: \\
\quad \left(\begin{array}{l}
ask(locate(X_p)) \Rightarrow a(finder,F) \leftarrow \mathcal{S} = [X_p|\mathcal{S}_r] \wedge \mathcal{L} = [X_l|\mathcal{L}_r]\ then \\
inform(located(X_p,X_l)) \Leftarrow a(finder,F)\ then \\
a(locator(F,\mathcal{S}_r,\mathcal{L}_r),L)\ \ )
\end{array} \right) \\
\quad or \\
\quad null \leftarrow \mathcal{S} = [] \wedge \mathcal{L} = [].
\end{array}
$$

$$
\begin{array}{l}
a(time\_coordinator(\mathcal{S},\mathcal{L},A,[T|\mathcal{T}_r],T_w),X) :: \\
\quad a(time\_proposer(\mathcal{S},\mathcal{L},A,T,R),X)\ then \\
\quad \left(\begin{array}{l}
null \leftarrow R = ok \wedge T_w = T \\
or \\
a(time\_coordinator(\mathcal{S},\mathcal{L},A,\mathcal{T}_r,T_w),X) \leftarrow R = not\_ok
\end{array} \right).
\end{array}
$$

$$
\begin{array}{l}
a(time\_proposer([P|\mathcal{S}_r],[L|\mathcal{L}_r],A,T,R),X) :: \\
\quad ask(available(T)) \Rightarrow a(person(P),L)\ then \\
\quad \left(\begin{array}{l}
\left(\begin{array}{l}
inform(available(T)) \Leftarrow a(person(P),L)\ then \\
a(time\_proposer(\mathcal{S}_r,\mathcal{L}_r,A,T,R),X)
\end{array} \right) \\
or \\
R = not\_ok \leftarrow inform(unavailable(T)) \Leftarrow a(person(P),L)
\end{array} \right).
\end{array}
$$

$$
\begin{array}{l}
a(headhunter,H) :: \\
\quad ask(best\_people(A)) \Leftarrow a(organiser(T,H,F),O)\ then \\
\quad inform(best\_people(A,\mathcal{S})) \Rightarrow a(organiser(T,H,F),O) \leftarrow choose\_persons(A,\mathcal{S}).
\end{array}
$$

$$
\begin{array}{l}
a(finder,P) :: \\
\quad ask(locate(X_p)) \Leftarrow a(Role,L)\ then \\
\quad inform(located(X_p,X_l)) \Rightarrow a(Role,L) \leftarrow likely\_location(X_p,X_l)\ then \\
\quad a(finder,P).
\end{array}
$$

$$
\begin{array}{l}
a(person(P),X) :: \\
\quad ask(available(T)) \Leftarrow Agent\ then \\
\quad \left(\begin{array}{l}
inform(available(T)) \Rightarrow Agent \leftarrow am\_available(T) \\
or \\
inform(unavailable(T)) \Rightarrow Agent \leftarrow am\_unavailable(T)
\end{array} \right).
\end{array}
$$

Figure 2.4: LCC scenario: a workshop scheduler

The role of the person contacted by our organiser to coordinate the time schedule with is also simple: if asked whether its schedule is free for a specific time/date T, the person replies with either available or unavailable messages.

2.2.3 LCC Interaction Execution

One of the two LCC engineering requirements is for agents to be able to extract the current protocol state, spot and perform their next permitted move(s), and save the new protocol state. This is easily achieved in LCC by applying a set of rewrite rules presented in Figure 2.5. The basic idea is that at the very beginning of the interaction, the agent makes a copy of the generic role definition by instantiating it with its own Id. When a new message is received, the agent looks up the appropriate role definition and performs its permitted actions. It then marks those actions as closed to help it spot its protocol state in the future. It then saves this modified role definition in the current protocol state, replacing the old one. All of this is achieved through one simple mechanism: applying the LCC rewrite rules.

The rewrite rules of Figure 2.5 are applied exhaustively to achieve a full expansion of a clause. An agent A with protocol definition $A :: B$ is expanded to $A :: E$ if B can be expanded to E. The term $A_1 \, or \, A_2$ can be expanded to E if either A_1 or A_2 can be expanded to E, given that the unexpanded part has not already been *closed*. For terms of the form $A_1 \, then \, A_2$, either A_1 is expanded, or A_2 is expanded if A_1 has already been *closed*. $A_1 \, par \, A_2$ is expanded to $E \, par \, A_2$ if A_1 expands to E, to $A_1 \, par \, E$ if A_2 expands to E, or to $E_1 \, par \, E_2$ if both A_1 and A_2 expand to E_1 and E_2, respectively. $M \Leftarrow A \leftarrow C$ is marked *closed* if the the constraint C can be satisfied and the message M is an element of the set of incoming messages M_i. $M \Rightarrow A \leftarrow C$ is marked *closed* if the constraint C is satisfied. In this case, the message M is added to the set of outgoing messages O. The term $null \leftarrow C$ is marked *closed* if the constraint C is satisfied. The term $a(R, I) \leftarrow C$ is expanded to $a(R, I) :: B$ if the constraint C is satisfied and the clause $a(R, I) :: B$ appears in the protocol $\mathcal{P}$.

2.2.4 Coordination in LCC

Robertson (2004a) describes the coordination mechanism in LCC as follows. An agent A engaged in interaction I receives the following tuple $(I, M, R, A, \mathcal{P})$, where M is the message, R is the role the agent should be playing in interaction I, and $\mathcal{P}$ is the protocol. The protocol consists of a set of LCC clauses C_F that defines the protocol frame-

$$
\begin{array}{ll}
A :: B \xrightarrow{M_i,M_o,\mathcal{P},O} A :: E & \text{if } B \xrightarrow{M_i,M_o,\mathcal{P},O} E \\
A_1 \text{ or } A_2 \xrightarrow{M_i,M_o,\mathcal{P},O} E & \text{if } \neg closed(A_2) \wedge A_1 \xrightarrow{M_i,M_o,\mathcal{P},O} E \\
A_1 \text{ or } A_2 \xrightarrow{M_i,M_o,\mathcal{P},O} E & \text{if } \neg closed(A_1) \wedge A_2 \xrightarrow{M_i,M_o,\mathcal{P},O} E \\
A_1 \text{ then } A_2 \xrightarrow{M_i,M_o,\mathcal{P},O} E \text{ then } A_2 & \text{if } A_1 \xrightarrow{M_i,M_o,\mathcal{P},O} E \\
A_1 \text{ then } A_2 \xrightarrow{M_i,M_o,\mathcal{P},O} A_1 \text{ then } E & \text{if } closed(A_1) \wedge A_2 \xrightarrow{M_i,M_o,\mathcal{P},O} E \\
A_1 \text{ par } A_2 \xrightarrow{M_i,M_o,\mathcal{P},O_1 \cup O_2} E_1 \text{ par } E_2 & \text{if } A_1 \xrightarrow{M_i,M_n,\mathcal{P},O_1} E_1 \wedge A_2 \xrightarrow{M_n,M_o,\mathcal{P},O_2} E_2 \\
C \leftarrow M \Leftarrow A \xrightarrow{M_i,M_i-\{M\Leftarrow A\},\mathcal{P},\emptyset} c(M \Leftarrow A) & \text{if } (M \Leftarrow A) \in M_i \wedge satisfied(C) \\
M \Rightarrow A \leftarrow C \xrightarrow{M_i,M_i,\mathcal{P},\{M\Rightarrow A\}} c(M \Rightarrow A) & \text{if } satisfied(C) \\
null \leftarrow C \xrightarrow{M_i,M_i,\mathcal{P},\emptyset} c(null) & \text{if } satisfied(C) \\
a(R,I) \leftarrow C \xrightarrow{M_i,M_i,\mathcal{P},\emptyset} a(R,I) :: B & \text{if } clause(\mathcal{P}, a(R,I) :: B) \wedge satisfied(C)
\end{array}
$$

where,
M_i represents the set of incoming messages,
M_o represents the modified set of incoming messages,
$\mathcal{P}$ represents both the generic and instantiated protocol, and
O represents the set of outgoing messages.
Also, $satisfied(C)$ is said to be true if the agent can satisfy C,
$clause(\mathcal{P}, X)$ is said to be true if the clause X appears in the dialogue framework of protocol $\mathcal{P}$, and
a protocol term is closed in the following cases:

$$
\begin{array}{l}
closed(c(X)) \\
closed(A \text{ or } B) \leftarrow closed(A) \vee closed(B) \\
closed(A \text{ then } B) \leftarrow closed(A) \wedge closed(B) \\
closed(A \text{ par } B) \leftarrow closed(A) \wedge closed(B) \\
closed(X :: D) \leftarrow closed(D)
\end{array}
$$

Figure 2.5: LCC rewrite rules

work, a set of clauses C_S that defines the current protocol state, and a set of clauses defining the common knowledge. The protocol framework is the original protocol which remains unchanged throughout an interaction. The protocol state consists of those clauses which are constantly modified to keep track of the current protocol state. Common knowledge in LCC is the knowledge needed to carry out a given interaction protocol. It is specific to the given interaction.

Upon receiving a tuple $(I, M, R, A, \mathcal{P})$, the agent checks whether there exists a copy of its own protocol state in C_S by checking for a clause matching its role and Id. If such a clause does exist, then it is retrieved, the rewrite rules described by Figure 2.5 are applied, and the new protocol state replaces the old one in C_S. However, if such a

clause does not exist, then the agent's original clause is retrieved from C_F and the new protocol state obtained by applying the rewrite rules is saved in C_S. Finally, messages that need to be sent to other agents will be sent along with the modified protocol $\mathcal{P}$. Note that in non-linear interactions, where several agents may send different messages simultaneously, each agent will have to save its own protocol state locally.

In conclusion, LCC addresses the challenge of achieving coordination in a modular way that has a very low impact on engineering agents by imposing only two engineering requirements: agents should have (1) a protocol expander, and (2) a constraint solver. The protocol expanding mechanism has already been introduced above. The constraint solver is what is needed to allow agents to solve LCC constraints. Note that LCC is neutral to both the protocol expansion mechanism and the choice of mechanism for communicating between agents. Although it is a logic programming language, two forms of deployment mechanisms have been implemented: Java and Prolog based mechanisms. LCC may also be considered neutral to the choice of constraint solver; however, we note that if different constraint solvers were addressing the same constraint in different manners, then this could raise coordination problems.

2.2.5 LCC and Process Calculi: a Comparison

Since our main goal is model checking multiagent systems, we focus on the process calculi nature of LCC. Process calculus is a calculus used for defining concurrent systems. The concept of process calculus emerged with Robin Milner's work on CCS, the Calculus for Communicating Systems (Milner, 1989). During this time, Hoare also started working on CSP, Communicating Sequential Processes (Hoare, 1985). Both CCS and CSP laid down the foundations on which process calculi that followed is based on. In what follows, we present a comparison between LCC, CCS, CSP, and the π-calculus(Milner, 1999), the most common process calculus, followed by a general overview of some features provided by other process calculi in comparison with LCC. Table 2.1 draws a comparison between the notations of the main four process calculi.

2.2.5.0.1 Definition/Recursion Operators. The operator used for defining processes (or agents in LCC) is common to all four process calculi of Table 2.1. It is defined as '::' in LCC, '$\stackrel{\text{def}}{=}$' in CCS and π-calculus, and '=' in CSP. The left hand side is usually the process' name (or agent's name in the form of $a(Role, Id)$ in LCC), while the right hand side is a term constructed using the remaining operators. Note that if the process'

	LCC	CCS	CSP	π-Calculus
Definition / Recursion	$A :: ADef$	$P \stackrel{\text{def}}{=} PDef$	$P = PDef$	$P \stackrel{\text{def}}{=} PDef$
Prefix	A *then* B	$a.P$	$a \rightarrow P$	$a.P$
Sequential	A *then* B		$P; Q$	
Choice	A *or* B	$P + Q$	$(a \rightarrow P \mid b \rightarrow Q)$ $P \sqcap Q$ $P \Box Q$	$P + Q$
Concurrency / Parallelism	A *par* B	$P \mid Q$	$P \parallel Q$ $P \mid\mid\mid Q$	$P \mid Q$
Restriction		$P \setminus J$		$(vx)P$
Concealment / Abstraction			$P \setminus C$	
Relabelling		$P[f]$	$L : P$	
Replication				$!P$
Input Action	$M \Leftarrow A\ [\leftarrow C]$	$a(x)$	$a?x$	$a(x)$
Output Action	$M \Rightarrow A\ [\leftarrow C]$	$\bar{a}(x)$	$a!x$	$\bar{a}(x)$
Empty Action / Empty Process	*null* $[\leftarrow C]$	NIL	$STOP$	**0**

Table 2.1: LCC and other process calculi: a rough comparison

name on the left hand side reappears in the right hand side, then this is a specification of recursion.

2.2.5.0.2 Prefix Operators. The prefix operator is also common to all four calculi. It specifies that a certain action is succeeded by a given process. It is defined as '*then*' in LCC, '.' in CCS and π-calculus, and '$\rightarrow$' in CSP.

2.2.5.0.3 Actions: Input/Output/Empty. Before we proceed with the remaining operators, we first introduce the various notions of actions. In CSP, actions could either be: (1) general interaction events, such as the ticking of a clock, which is represented via an action label '*tick*', or (2) a communication event, such as input '$a?x$' and output '$a!x$' actions, where 'x' is the transmitted data and 'a' is the name of the channel the data is transmitted through.

In CCS and π-calculus, interaction events are treated as input and output actions. Hence, actions in CCS and π-calculus are either input '$a(x)$', output '$\bar{a}(x)$', or silent τ actions. If no data is transmitted, then 'x' is dropped and the input and output actions

are represented via the channel names 'a' and '$\bar{a}$', respectively. This is usually used to specify synchronisation through signalling. The τ action is used to synchronise the input and output actions, forcing them to occur simultaneously. τ is discussed in more detail in Section 2.2.5.0.6, when addressing the concurrency issue.

LCC actions could be input '$M \Leftarrow A$' or output '$M \Rightarrow A$' actions. In LCC, the message M is a structured Prolog term. It could either be used as signals for synchronisation or for transmitting complex data structures. However, the concept of τ does not exist in LCC. This is basically due to the fact that LCC is not only a specification language but an executable language as well. This implies that it is not enough to state that input and output actions should occur in parallel; the details of these actions should also be specified. Furthermore, multiagent systems use networks that are not 100% reliable for transmitting messages: messages may be lost, delayed, or delivered in a wrong order. Therefore, it is impossible to guarantee, and hence to specify, that a sent message will instantly be collected. Also note that instead of specifying the direction of flow of messages through the specification of the channel through which a message M is sent, LCC requires the specification of the agent A which the message is sent from/to. This requires the specification of the agent's role and Id. In distributed open system, such as multiagent systems, agents should be capable of sending/receiving messages to/from any other agent in the system: restrictions on who can send messages to whom should be dealt with at a higher level by the agents themselves. Therefore, each agent is assumed to have channels connecting it to all other agents in the system, which makes channels synonymous with agents' roles and Ids.

LCC also allows the specification of agents' internal actions. Internal actions, also referred to as non-message passing actions (N-MPA), are actions performed by the agent on its own without the need to communicate with other agents. This usually represents internal computations, specified by attaching constraints '$[\leftarrow C]$' to other input/output actions. If only internal actions are to be performed, then the constraints would be attached to either the empty action '*null*' or the action of taking on a different role '$a(Role, Id)$'. Note that '*null*' in LCC differs slightly from the 'NIL' of CCS, '$STOP$' of CSP, and '$\mathbf{0}$' of the π-calculus. While '*null*' is an empty action, which may be succeeded by other actions, 'NIL', '$STOP$', and '$\mathbf{0}$' are empty processes that can not be followed by any other action, hence symbolising the very end of a process.

2.2.5.0.4 Sequential Operators. As opposed to the prefix operator that only permits actions to be succeeded by processes, the sequential operator allows the spec-

ification of processes being succeeded by other processes. This operator is unique to LCC and CSP. Note that '*then*' in LCC is used both as a prefix and a sequential operator, while ';' is the sequential operator of CSP.

2.2.5.0.5 Choice Operators. In CSP, there are three different choice operators. The first is the deterministic choice operator '|', which is used for specifying the choice between actions as opposed to the choice between processes. This is basically due to the fact that the initial actions should be distinct. Hence, it is used as follows '$(a \rightarrow P \mid b \rightarrow Q)$', where $a \neq b$. However, if the choice is between two actions which are essentially the same ($a = b$), then the non-deterministic choice operator '$\sqcap$' is used on processes as follows '$P \sqcap Q$'. This implies that the environment cannot influence the choice of processes, hence the name *non-deterministic choice*. Finally, CSP also defines a more general choice operator '$\Box$' which is used to represent both deterministic '|' and non-deterministic '$\sqcap$' choice operators. On the other hand, the choice operators, '+' of CCS and the π-calculus and '*or*' of LCC, do not differentiate between external choices affected by the environment and internal ones. In LCC, choices may or may not be affected by the agents. We believe that explicitly specifying the entity behind making the choice, whether it is the environment, the agents, etc., is irrelevant since it does not affect the flow of interactions in multiagent systems.

2.2.5.0.6 Concurrency, Restriction, and Concealment Operators. Concurrency is achieved by having several processes acting in parallel. In CSP, two operators are used to express concurrency. The parallel operator '||' is used when actions common to both processes may only occur simultaneously. An example of this is having the actions of sending and receiving a message occur in parallel, as opposed to sequentially. The interleaving operator '|||' is used when processes are concurrent, but their common actions are not necessarily synchronised. In CSP, the actions common to two processes are labelled by the same action name. In CCS and π-calculus, these common actions are also labelled by the same action name, but they are described as actions and their co-actions, with the co-actions having an additional bar above their actions' name. When synchronisation is necessary in CCS and π-calculus, the occurrence of one action and its co-action may be labelled as the action τ. This implies that both actions are synchronised and their details are concealed from the outside world. Although CSP does not define a τ action, it does provide a concealment operator '\' for concealing actions from the outside world. The CCS and π-calculus' concurrency operator '|' is

used to represent the interleaving operator '|||' of CSP. If certain actions and co-actions should be synchronised without being concealed, then the restriction operator '\' is used. Therefore, the parallel operator of CCS in conjunction with its restriction operator '\' becomes similar in functionality to parallel operator '||' of CSP, with respect to the restricted actions.

The LCC parallel operator '*par*' is equivalent to that of CCS. However, there is no equivalence in LCC to the τ action of CCS and no explicit definition of synchronisation through a restriction operator such as the '\' of CCS. When synchronisation is necessary in LCC, it may be explicitly specified in the process' definition, possibly using signals and/or acknowledgements. Using such methods, we believe LCC provides a more realistic approach for dealing with synchronisation in distributed open systems. This is because of two reasons. First, we assume that there is no shared information between agents other than by message passing, making message passing the only method for synchronisation. Second, communication in distributed open systems does not make use of dedicated channels, as is the case of traditional process calculi systems, but between agents and through less reliable networks. When communication is performed through a network instead of a dedicated channel, there is no way of controlling many aspects of communication, such as the delivery of messages, the order of messages received, etc. Therefore, synchronising a message output action with its message input co-action is not realistic.

2.2.5.0.7 Relabelling Operators. Relabelling is used by both CCS and CSP. In CCS, it is specified as '$P[f]$', where f is the function that performs the relabelling. In CSP , it is specified as '$L : P$', where L is the set of labels used for relabelling. Relabelling actions is useful for constructing a system from similar processes operating concurrently. However, the issue of relabelling is irrelevant to LCC, where messages are directly sent from/to agents with specific roles and Ids, as opposed to common channels. Therefore, even if there exists more than one agent playing the same role in an interaction, then the messages sent by/to these agents can always be distinguished through the agents' unique Ids. This eliminates the need of relabelling in LCC.

2.2.5.0.8 Replication Operator. While recursion is defined as having infinite sequential occurrences of the same process, replication is defined as having infinite concurrent occurrences of the same process. In π-calculus, replication is specified by '$!P = P \mid P \mid P \mid ...$'. In LCC, the replication operator does not exist basically due to the

restricted use of the concurrent operator on which replication is based.

Multiagent systems are viewed as a collection of agents working in parallel. The use of replication seems appealing for several scenarios. For example, in an auction system, it would be convenient to be able to specify that there is exactly one auctioneer and an indefinite number of bidders acting concurrently. In LCC, this cannot be made explicit; such a specification is implicit in the auctioneer's definition, for example, which is responsible for sending invites to bidders. At the beginning of the interaction, one copy of each generic agent role definition is sufficient. As other agents start joining the interaction (such as bidders in an auction system), they instantiate the appropriate generic agent role definition by binding it to their unique Id and add this new instantiated process to the pool of concurrent process.

Since this thesis is concerned with the verification of multiagent system interactions, it is important to note the effect of replication on the verification process. Ideally it would be interesting to prove that certain properties hold, for instance, for an infinite number of bidders. In practise, this depends on the verifier's capability of dealing with infinite processes. For model checking, the technique used for verification in this thesis, the system model that needs to be verified should be a finite system. This implies that only a finite set of parallel processes may be used, regardless of whether these processes use sequential recursion resulting in an infinite behaviour. As a result, our verification process uses the concurrent operator only at the final stage of specification, after all processes (or agent definitions) have already been specified. It is used as a way of constructing an instance of the interaction model by specifying which agents/roles will be playing in parallel.

2.2.5.0.9 Additional CSP Operators. Table 2.1 omits several of the least common features of CSP. Unlike CCS, CSP was initially designed with the issue of addressing failure in mind. Some of these features deal with interrupts and catastrophes, and offer recovery methods, such as restarting and making use of checkpoints. LCC does not allow the specification of possible reaction to failure. The decision of what to do next is left entirely for the individual agents. Nevertheless, the coordination mechanism of LCC provides a copy of the interaction's current state-space along with the generic one. This helps agents to pinpoint the exact point of failure and decide their next move based on that. Mechanisms addressing failure could then be built on a higher level, making use of the available current and generic state-spaces. For example, Osman (2003) explores the possibility of allowing agents to backtrack in LCC to other more successful paths

of the state-space when necessary; however, it does not tackle the problem of agents not always being able to backtrack after certain internal actions have already been executed and committed to. Hassan et al. (2005) provides a constraint failure mechanism that allows agents to restrict rather than simply instantiate LCC constraints, resulting in interactions which are less likely to break. McGinnis and Robertson (2004) provide a mechanism for dynamically adapting LCC interactions to increase the chance of successful termination.

Some of the other features of CSP are those of using assignment, conditionals, and loops. The LCC syntax and its use of constraints allows the specification of such features in a straightforward manner.

2.2.5.0.10 π-Calculus and Dynamic Processes. The π-calculus in Table 2.1 looks extremely similar to CCS. The π-calculus, developed by Milner, Parrow, and Walker, is in fact based on Milner's CCS. The main difference the π-calculus introduces is the concept of dynamic process. In CCS, data is communicated between processes via channels. In the π-calculus, ports (or channels) may also be communicated between processes. This implies that the system can now evolve as its configuration changes. This is similar to using variables in LCC for specifying the agent to communicate with. Since messages are sent to agents with specific roles and Ids, then the role and Id may be specified as a variable whose value is computed and possibly exchanged at run time.

2.2.5.0.11 Features of Other Process Calculi. Several flavours of CCS, CSP, and π-calculus have emerged. New languages, originally based on these three languages, have also been introduced. Each of these brought forward new concepts not dealt with by its predecessors. Samples of these new concepts are the use of timeouts, prioritising actions, examining stochastic behaviour, considering an agent's ambience, etc. In LCC, these features are not explicitly defined. When necessary, a quick fix would be to incorporate many of these features by specifying them in terms of LCC constraints. Timeouts, for instance, may be inserted as constraints wherever necessary in an interaction's state-space. Priorities could be specified as constraints affecting the agent's choice of actions based on the agents' preference. As for the notion of ambience, it is worth noting that the various benchmark scenarios of this thesis were successfully specified in LCC with no need for ambient specification. For instance, rather than having to invest in new notation, LCC allows the specification of environmental changes for a role within the role's definition by making use of the role's parameters. Nevertheless,

a version of LCC incorporating the notion of ambient is currently being studied by the EU funded OpenKnowledge project (Joseph et al., 2006).

2.3 Modal and Temporal Logic

As illustrated in Section 2.1.2.1, modal and temporal logics are typically used by model checkers to specify the properties to be verified in a given system model. This section provides a general introduction to modal and temporal logic, focusing on the μ-calculus, our model checker's specification language.

Logic is used to determine the validity of arguments and propositions, aiming at proving which are true and which are false. But how do traditional logics, such as propositional logic, deal with propositions of the form "it is *possible* that the U.S. will attack Iran over its nuclear program". Modal logic is a logic that extends propositional logic by allowing the specification of the *necessity* and *possibility* modalities; hence, allowing inference over such arguments. In addition to the logical operators of propositional logic $\neg$, $\wedge$, $\vee$, $\rightarrow$, and $\Rightarrow$, the two modal operators $\Box$ and $\Diamond$ are introduced for representing *necessity* and *possibility*, respectively. The semantics of modal logic are then defined in terms of *possible worlds*. $\Box P$ is true iff P is true in all possible worlds. Similarly, $\Diamond P$ is true iff P is true in some possible world [2].

Modal logic has helped define various ideas in terms of modalities resulting in different classes of modal logic, where each class has its own interpretation of the $\Box$ and $\Diamond$ operators. For instance, deontic logic is the logic of obligations, permissions, prohibitions, etc. It deals with statements such as '*one must do this*' and '*one may do that*'. These are defined in terms of the $\Box$ and $\Diamond$ modal operators, requiring a slightly different interpretation of these operators with a new set of axioms. Another example is temporal logic, which deals with the temporal information about propositions. Temporal logic deals with statements of the form '*it will always be true that*" and '*it will eventually be true that*'. Again, this temporal information may be viewed as a different interpretation of the modal operators. Similarly, epistemic logic (logic of knowledge), doxastic logic (logic of belief), and many others have also been defined in terms of the $\Box$ and $\Diamond$ modal operators.

In this thesis, we make use of both temporal and deontic logic. We give an overview of temporal logic in this section, followed by an overview of deontic logic in Sec-

[2] Some studies, such as Blackburn and van Benthem (2006), have been interested in the relationship between modal logic and first-order logic.

Syntax:

$\Phi \quad ::= \quad \texttt{tt} \mid \texttt{ff} \mid \Phi_1 \wedge \Phi_2 \mid \Phi_1 \vee \Phi_2 \mid [K]\Phi \mid \langle K\rangle\Phi$

Semantics:

$E \models \texttt{tt}$

$E \not\models \texttt{ff}$

$E \models \Phi_1 \wedge \Phi_2 \quad \text{iff} \quad E \models \Phi_1 \text{ and } E \models \Phi_2$

$E \models \Phi_1 \vee \Phi_2 \quad \text{iff} \quad E \models \Phi_1 \text{ or } E \models \Phi_2$

$E \models [K]\Phi \quad \text{iff} \quad \forall F \in \{E' \,:\, E \xrightarrow{a} F \text{ and } a \in K\}.\ F \models \Phi$

$E \models \langle K\rangle\Phi \quad \text{iff} \quad \exists F \in \{E' \,:\, E \xrightarrow{a} F \text{ and } a \in K\}.\ F \models \Phi$

Figure 2.6: Hennessy-Milner Logic: syntax and semantics

tion 2.4. The temporal logic used by our model checker is the μ-calculus, an extension of the modal logic: Hennessy-Milner Logic (HML). Therefore, we provide an overview of HML in Section 2.3.1, followed by a general introduction to temporal logics in Section 2.3.2, and a specification of the μ-calculus in Section 2.3.3.

2.3.1 Hennessy-Milner Logic

The Hennessy-Milner Logic (HML) (Hennessy and Milner, 1980) is a logic for describing the capabilities/actions of processes. Its syntax and semantics are presented in Figure 2.6. The syntax implies that HML formulae may be constructed using the boolean connectives $\wedge$ and $\vee$, the modal operators *box* and *diamond*, and the truth constants `tt` and `ff`. In HML, the *box* and *diamond* operators are defined in the context of an explicitly specified action K: $[K]\Phi$ and $\langle K\rangle\Phi$. Note that the Hennessy-Milner Logic was later on extended to the modal logic M[3], by letting K represent a set of actions instead of a single action. In the modal logic M, the set $-K$ abbreviates $A - K$, where A is the universal set of actions. Similarly, the set $-$ abbreviates $A - \emptyset$.

HML formulae are used to describe properties of processes. A process E either satisfies a modal formula Φ ($E \models \Phi$) or not ($E \not\models \Phi$). Therefore, the semantics of HML formulae are better understood in the context of their satisfaction.

The syntax implies that the property `tt` is always satisfied, whereas `ff` is never satisfied. The property $\Phi_1 \wedge \Phi_2$ is satisfied at a process E, if both properties Φ_1 and Φ_2 are satisfied at E. Similarly, $\Phi_1 \vee \Phi_2$ is satisfied at a process E, if either property Φ_1

[3]The syntax and semantics of HML and the modal logic M are taken from Stirling (2001).

or Φ_2 is satisfied at E. $[K]\Phi$ is satisfied at E if for every action $a \in K$ that E can take to F, Φ will be satisfied at F. In other words, the *box* operator $[K]\Phi$ implies that it is necessary that any action $a \in K$ will lead to the satisfaction of property Φ. Similarly, $\langle K\rangle\Phi$ is satisfied at E if there exists at least one action $a \in K$ that E can take to F and Φ is satisfied at F. In other words, the *diamond* operator $\langle K\rangle\Phi$ implies that it is possible for an action $a \in K$ to occur, leading to the satisfaction of property Φ.

Note that HML is a modal logic. It expresses the capability of performing actions in the immediate next step. However, there is no way of describing temporal properties, such as the capability of the clock to continuously tick forever. Temporal notations and temporal logics are introduced in the following section.

2.3.2 Temporal Logics: an Overview

Temporal logic is a logic that is concerned with representing and reasoning about temporal information. In temporal logic, time is viewed as a sequence of states, representing the "flow of time". The logic allows reasoning about temporal properties of propositions by studying which states these propositions are valid at.

Modern temporal logic is based on the Tense Logic of Arthur Prior (Prior, 1955), which is based on modal logic. In addition to the logical operators, Prior introduced four modal operators, namely **P**, **H**, **F**, and **G**, for representing what has been true in the past, what has always been true in the past, what will be true in the future, and what will always be true in the future.

Several extensions were made to the Tense Logic. Amongst these were the introduction of the **S** and **U** temporal operators for representing 'p is valid since q' and 'p is valid until q', respectively. Later on, the *next* temporal operator **X** was introduced to specify that a proposition holds in the immediate time step, i.e. in the next state.

All of the above operators model temporal properties in a deterministic world. Temporal logics that focus on deterministic worlds are called linear temporal logics. To account for non-determinism, branching into the future is made possible by introducing two branching operators, **A** and **E**, that act as quantifiers over the different paths/branches. **A** stands for *for all paths*, while **E** stands for *there exists a path*.

Table 2.2 presents the various temporal operators used, the languages that use them, and a simple description of each. Note that the most popular and widely used temporal logics are the linear time logic LTL and the branching time logics CTL, CTL*, and μ-calculus (Kozen, 1983; Edmund M. Clarke et al., 1999, p. 27-32). Table 2.2 only

focuses on the original Tense logic along with LTL and CTL. CTL* is excluded because it is an extension of CTL that uses the same operators, but extends the syntax by allowing the operators to be freely mixed[4]. The μ-calculus is also excluded by Table 2.2 simply because it uses a completely different syntax. The next section introduces the μ-calculus, our model checker's branching temporal logic.

Operator	Name	Used by	Description
Pα	Past	Tense	It has, at some time, been the case that α was true.
Hα	-	Tense	It has always been the case that α was true.
Fα	Future / Eventually	Tense, LTL, CTL	It will, at some time, be the case that α is true.
Gα	Globally	Tense, LTL, CTL	It will always be the case that α is true.
α**U**β	Until	LTL, CTL	It will always be the case that α is true until β is true. And β should become true at some point.
W	Weak Until	CTL	It will always be the case that α is true until β is true. But β need never be true.
α**R**β	Release	LTL,	If α becomes true, then β need not be true anymore.
Nα /**X**α	Next	LTL, CTL	In the next state, it will be the case that α is true.
Aα	All/Always	CTL	In all paths, it will be the case that α is true.
Eα	Exists	CTL	There exists a path where it will be the case that α is true.

Table 2.2: Temporal operators

2.3.3 The Modal μ-Calculus

In Section 2.3.1, we saw that modal logic allows us to specify properties of immediate capabilities and necessities. However, it was unable to express enduring traits of processes, such as those representing properties of the form 'it is always the case that'. The temporal logics discussed in Section 2.3.2, introduced new operators to describe long term, or even infinite, features of processes. The main two operators that express these are the **G** and **F** operators, representing 'it will always be the case' and 'it will eventually be the case', respectively. The minimal set of operators depends on the specific logic. In CTL, for example, all operators may be defined in terms of terms of **AG** and **EG**.

[4]Unlike CTL*, CTL applies strict rules on the mixing of CTL operators. For example, operators should occur in pairs having one branch operator (such as **A** or **E**) followed by a state operator (such as **F**, **G**, etc.).

The μ-calculus, however, takes a different approach in addressing temporal properties. It may be viewed as an extension to the Hennessy-Milner Logic. It basically uses the Hennessy-Milner Logic for representing the immediate capabilities and necessities. It then builds on top of this using a form of recursion to achieve a representation of long term and infinite features. The recursion of the μ-calculus is achieved by making use of fixpoint operators. Despite its simple syntax, the result is a rich logic with powerful features. However, there is one major drawback for using the modal μ-calculus. This is best described by Bradfield and Stirling (2001, p. 9), which provides moral support for the newcomers to this field:

> Fixpoint logics are notorious for being incomprehensible. Indeed, it has been known for several reasonably expert people to spend an hour or two trying to work out whether a one line formula means what it was intended to mean. Furthermore, a full understanding of the mathematical intricacies of the modal mu-calculus requires familiarity with mathematics that, although not difficult, is not covered in most undergraduate mathematics or computer science programmes; this can be a source of irritation for the practitioner new to the area. Fortunately, one can get a very good intuitive understanding, which is even provably equivalent to the formal semantics, without knowing the full details. Moreover, this intuition is all but essential for understanding the deep theory as well.

Despite this drawback, the modal μ-calculus has been the focus of interest of many researchers in the field of verification. This is for two main reasons. First, the logic is so simple yet so rich that it subsumes most temporal languages used today, such as LTL, CTL, ACTL, PDL, PDL-Δ, and CTL* (Mateescu and Sighireanu, 2003; Leucker et al., 2003). Second, efficient model checking algorithms are already available for this logic (Edmund M. Clarke et al., 1999, p. 97).

Section 2.3.3.1 introduces the syntax and semantics of the μ-calculus. This is followed by Section 2.3.3.2, which aims at providing a basic understanding of the calculus. Finally, Section 2.3.3.3 introduces a subset of the calculus, the alternation-free μ-calculus, which is the subset used by our model checker.

2.3.3.1 Syntax and Semantics

Figure 2.7 presents the syntax and semantics of the modal μ-calculus. The syntax extends that of the Hennessy-Milner logic (Figure 2.6) by introducing propositional variables (Z) along with the least and greatest fixed point operators (μZ and νZ, respectively).

Syntax:

$\Phi ::= \quad \mathtt{tt} \quad | \quad \mathtt{ff} \quad | \quad Z \quad | \quad \Phi_1 \wedge \Phi_2 \quad | \quad \Phi_1 \vee \Phi_2 \quad | \quad [K]\Phi \quad | \quad \langle K \rangle \Phi \quad | \quad \nu Z.\Phi \quad | \quad \mu Z.\Phi$

Semantics:

$E \models_{\mathsf{V}} \mathtt{tt}$

$E \not\models_{\mathsf{V}} \mathtt{ff}$

$E \models_{\mathsf{V}} Z$ iff $E \in V(Z)$

$E \models_{\mathsf{V}} \Phi_1 \wedge \Phi_2$ iff $E \models_{\mathsf{V}} \Phi_1$ and $E \models_{\mathsf{V}} \Phi_2$

$E \models_{\mathsf{V}} \Phi_1 \vee \Phi_2$ iff $E \models_{\mathsf{V}} \Phi_1$ or $E \models_{\mathsf{V}} \Phi_2$

$E \models_{\mathsf{V}} [K]\Phi$ iff $\forall F \in \{E' \ : \ E \xrightarrow{a} F \text{ and } a \in K\}.\ F \models_{\mathsf{V}} \Phi$

$E \models_{\mathsf{V}} \langle K \rangle \Phi$ iff $\exists F \in \{E' \ : \ E \xrightarrow{a} F \text{ and } a \in K\}.\ F \models_{\mathsf{V}} \Phi$

$E \models_{\mathsf{V}} \nu Z.\Phi$ iff $E \in \bigcup \{\mathsf{E} \subseteq \mathsf{P} \ : \ \mathsf{E} \subseteq \{E \in \mathsf{P} : E \models_{\mathsf{V}[\mathsf{E}/Z]} \Phi\}\}$

$E \models_{\mathsf{V}} \mu Z.\Phi$ iff $E \in \bigcap \{\mathsf{E} \subseteq \mathsf{P} \ : \ \{E \in \mathsf{P} : E \models_{\mathsf{V}[\mathsf{E}/Z]} \Phi\} \subseteq \mathsf{E}\}$

Figure 2.7: Modal μ-calculus: syntax and semantics

Most of the satisfaction rules are similar to those of HML of Figure 2.6. However, a valuation function V is introduced to assign to each variable Z a subset of processes $\mathsf{V}(Z)$ that satisfy this variable (or property). Therefore, we say state E satisfies Z ($E \models_V Z$), if and only if E is an element of $\mathsf{V}(Z)$. Note that the satisfaction of a property Φ at a given state E is now defined as $E \models_{\mathsf{V}} \Phi$, as opposed to $E \models \Phi$ in HML, since satisfaction is now relative to the valuation V of the propositional variables that appear in the property Φ.

The modal μ-calculus' main extension to HML is the addition of the least and greatest fixed point operators. A state E is said to satisfy property $\mu Z.\Phi$ if E is an element of the least fixed point solution of Φ, which is the intersection of all prefixed points. Similarly, E is said to satisfy the property $\nu Z.\Phi$ if E is an element of the greatest fixed point solution of Φ, which is the union of all post-fixed points. Note that $\mathsf{V}[\mathsf{E}/Z]$ is the valuation that maps E to Z, but otherwise agrees with V. The following section aims at providing an intuitive understanding of the least and greatest fixed point operators of the modal μ-calculus.

2.3.3.2 An Intuitive Understanding

2.3.3.2.1 Fixed Points In summary, a fixed point of a function is a point whose value remains unchanged (i.e. fixed) after applying the function to it. For example, the function $f(x) = x^2$ has two fixed points: $x = 0$ and $x = 1$, where $f(0) = 0$ and

$f(1) = 1$. In what follows, we provide some basic definitions of the various types of fixed points.

Consider the function f on a lattice $(\mathsf{S}, \subseteq)$. A fixed point of the function f is a value x, such that $x = f(x)$. A pre-fixed point of the function f is a value $x \in S$, such that $f(x) \subseteq x$. Similarly, a post-fixed point of the function f is a value $x \in S$, such that $x \subseteq f(x)$. Note that a fixed point of f is both a pre-fixed and post-fixed point of f. The least fixed point of the function f is a fixed point that is less than or equal to all other fixed points, according to some partial order. It is defined as $\bigcap\{x \subseteq S : f(x) \subseteq x\}$. Similarly, the greatest fixed point of the function f is a fixed point that is greater than or equal to all other fixed points, according to some partial order. It is defined as $\bigcup\{x \subseteq S : x \subseteq f(x)\}$.

2.3.3.2.2 Extending HML with Fixpoint Operators Let f be defined as $f[\Phi, Z](\mathsf{E}) = \{E \in \mathsf{P} : E \models_{\mathsf{V}[\mathsf{E}/Z]} \Phi\}$; i.e. f is a function that takes a set of states E as its input and outputs the set of states $E \in \mathsf{P}$ that satisfy the property Φ with respect to the valuation V. Stirling (2001, p. 94–97) presents a proof of the monotonicity of f followed by a sample proof of the Knaster-Tarski theorem that guarantees the existence of a unique least and greatest fixed points of f. The Knaster-Tarski theorem is an important theorem to the μ-calculus because it guarantees the existence of unique least and greatest fixed point solutions for monotonic functions on complete lattices $(\mathsf{P}, \subseteq)$.

The modal μ-calculus extends the Hennessy-Milner Logic with recursion. Recursion forms the essence of the modal μ-calculus. Equations of the form $Z \stackrel{\text{def}}{=} \Phi$, where the variable Z reappears in the right hand side of the equation, are used to specify temporal properties of processes. Such equations have unique greatest and least fixed point solutions, the solutions of the monotonic function $f[\Phi, Z]$. The functions that compute these least/greatest fixed point solutions are called the least/greatest fixpoint operators and are defined as $\mu Z.\Phi$ and $\nu Z.\Phi$, respectively.

2.3.3.2.3 Exercise As an exercise, this section provides an example that makes use of pre and post fixed points. Readers with the necessary mathematical background on this subject may find this section helpful. Other readers may directly skip to the following section (Section 2.3.3.2.4).

Let us define the CTL formula $EF\Phi$ using the fixed point operators. $EF\Phi$ specifies that there exists a path in which Φ is eventually satisfied. It is easy to define the

appropriate recursive modal equation: $Z \stackrel{\text{def}}{=} \Phi \vee \langle - \rangle Z$, where Φ does not contain Z, i.e. Z is unbound (or free) in Φ. This equation implies that either Φ is satisfied at the current state or something can happen which satisfies Z, and so on. The tricky question is this: which fixed point solution of this equation best describes $EF\Phi$? To find the right solution, we compute both the pre and post fixed points accordingly:

$$
\begin{aligned}
\text{Pre-fixed Points} \qquad & f[\Phi \vee \langle - \rangle Z, Z](\mathsf{E}) \subseteq \mathsf{E} \\
= \quad & \{E \in \mathsf{P} \ : E \models_{\mathsf{V}[\mathsf{E}/Z]} \Phi \vee \langle - \rangle Z\} \subseteq \mathsf{E} \\
= \quad & \{E \in \mathsf{P} \ : E \models_{\mathsf{V}} \Phi \ \vee \ E \models_{\mathsf{V}[\mathsf{E}/Z]} \langle - \rangle Z\} \subseteq \mathsf{E} \\
= \quad & \{E \in \mathsf{P} \ : E \models_{\mathsf{V}} \Phi \ \vee \ \exists a \in K.\ \exists F \in \mathsf{E}.\ E \xrightarrow{a} F\} \subseteq \mathsf{E} \\
= \quad & \text{if } E \in \mathsf{P} \text{ and } (E \models_{\mathsf{V}} \Phi \text{ or } \exists a \in K.\ \exists F \in \mathsf{E}.\ E \xrightarrow{a} F) \text{ then } E \in \mathsf{E}
\end{aligned}
$$

$$
\begin{aligned}
\text{Post-fixed Points} \qquad & \mathsf{E} \subseteq f[\Phi \vee \langle - \rangle Z, Z](\mathsf{E}) \\
= \quad & \mathsf{E} \subseteq \{E \in \mathsf{P} \ : E \models_{\mathsf{V}[\mathsf{E}/Z]} \Phi \vee \langle - \rangle Z\} \\
= \quad & \mathsf{E} \subseteq \{E \in \mathsf{P} \ : E \models_{\mathsf{V}} \Phi \ \vee \ E \models_{\mathsf{V}[\mathsf{E}/Z]} \langle - \rangle Z\} \\
= \quad & \mathsf{E} \subseteq \{E \in \mathsf{P} \ : E \models_{\mathsf{V}} \Phi \ \vee \ \exists a \in K.\ \exists F \in \mathsf{E}.\ E \xrightarrow{a} F\} \\
= \quad & \text{if } E \in \mathsf{E} \text{ then } E \models_{\mathsf{V}} \Phi \text{ or } \exists a \in K.\ \exists F \in \mathsf{E}.\ E \xrightarrow{a} F
\end{aligned}
$$

The pre-fixed point solution implies that a set of processes $\mathsf{E} \subseteq \mathsf{P}$ is a set that contains all states E satisfying Φ. It then builds on top of that by adding all states E which do not necessarily satisfy Φ, but can perform some action a to some state F that does satisfy Φ. And so on. Therefore, the set E possesses the property of performing a finite number of actions until Φ is eventually satisfied. In CTL, this would be defined as $EF\Phi$ (Stirling, 2001, p. 98). The post-fixed point solution implies that a state which is an element of E could either satisfy Φ or it could perform some action a to another state in E. While the pre-fixed point solution always contains states that satisfy Φ (if $E \models_{\mathsf{V}}$ then $E \in \mathsf{E}$) and then builds on top of that by adding the states that can reach those that satisfy Φ, the post-fixed point does not. This keeps the possibility of performing actions infinitely often without ever reaching a state satisfying Φ open. In CTL, this would be defined as $EF\Phi \vee EG\langle - \rangle \mathtt{tt}$ (Stirling, 2001, p. 98).

Note that the direction of the implication constitutes the main difference between pre and post fixed point solutions. Therefore, Bradfield and Stirling (2001) suggest that instead of using recursive equations of the form $Z \stackrel{\text{def}}{=} \Phi \vee \langle - \rangle Z$, one should think of recursion in the form of either $X \Leftarrow \Phi \vee \langle - \rangle X$ or $Y \Rightarrow \Phi \vee \langle - \rangle Y$. The first defines a pre-fixed point, stating that 'if a state satisfies $\Phi \vee \langle - \rangle X$, then it satisfies any solution X' of the equation'. On the other hand, the second defines a post-fixed point, stating that 'if Y' satisfies any solution of the equation, then it surely satisfies $\Phi \vee \langle - \rangle Y$'.

Interestingly enough, the Knaster-Tarski theorem guarantees the existence of unique

pre and post-fixed points. Furthermore, the greatest post-fixed point and the least pre-fixed point of a monotonic function are essentially the greatest and least fixed points. Bradfield and Stirling (2001) argue that one may continue to use the logic presented above by selecting between pre and post fixed point solutions, and yet use the notation of the least and greatest fixed point operators μ and ν, respectively.

Going back to the main question raised by this example, one concludes that the property defined as 'there exists a path in which the property Φ eventually holds' is expressed as $\mu Z.\Phi \vee \langle - \rangle Z$ in the modal μ-calculus. Whereas the formula $\nu Z.\Phi \vee \langle - \rangle Z$ represents the property defined as 'there exists a path where either Φ is eventually satisfied or actions may be performed infinitely often'.

2.3.3.2.4 Basic Intuition Bradfield and Stirling (2001, Section 3.2) provide us with yet another basic intuitive understanding to the complex modal μ-calculus. In conclusion, they argue that with a little refinement one may understand fixpoint operators and the μ-calculus accordingly: "ν means looping, and μ means finite looping". This notion of looping versus finite looping is the main difference between the two formulae discussed above: $\nu Z.\Phi \vee \langle - \rangle Z$ and $\mu Z.\Phi \vee \langle - \rangle Z$. As we saw in the previous example, $\mu Z.\Phi \vee \langle - \rangle Z$ specifies that recursion could occur, but it should be finite because Φ will eventually be true. However, $\nu Z.\Phi \vee \langle - \rangle Z$ specifies that recursion could be infinite with Φ never being true.

2.3.3.2.5 A Note on Negation The monotonicity of a function is one of the basic requirements for guaranteeing a unique least/greatest fixed point solution. As a result, to preserve monotonicity, negation is usually avoided in the μ-calculus (it may only be permitted under strict conditions, as explained by Stirling (2001, p. 101)). Nevertheless, the absence of negation is not a major issue, since the complements of formulae (marked with "^{c}") are already defined in the logic:

$$\begin{array}{llllll}
\mathtt{tt}^c & = & \mathtt{ff} & \mathtt{ff}^c & = & \mathtt{tt} \qquad\qquad Z^c = Z \\
(\Phi \wedge \Psi)^c & = & \Phi^c \vee \Psi^c & (\Phi \vee \Psi)^c & = & \Phi^c \wedge \Psi^c \\
([K]\Phi)^c & = & \langle K \rangle \Phi^c & (\langle K \rangle \Phi)^c & = & [K]\Phi^c \\
(\nu Z.\Phi)^c & = & \mu Z.\Phi^c & (\mu Z.\Phi)^c & = & \nu Z.\Phi^c
\end{array}$$

2.3.3.3 The Alternation Free μ-calculus

Alternation depth in the μ-calculus represents the number of nested alternating least and greatest fixed point operators. A widely used subset of the modal μ-calculus is the alternation-free μ-calculus, μMA, which permits dependence of fixed points as long as they are of the same kind. It was first introduced by Emerson and Lei (1986), and may be summarised as follows (Stirling, 2001, p. 126):

$$\Phi \in \mu\text{MA} \quad \text{iff} \quad \text{if } \mu Y.\ \Psi_1 \in Sub(\Phi) \text{ and } \nu Z.\ \Psi_2 \in Sub(\Phi) \text{ then } Y \text{ is not free in } \Psi_2 \text{ and } Z \text{ is not free in } \Psi_1$$

In order to better understand the alternation-free μ-calculus, Figure 2.8 provides a graphical representation of the following two logical statements:

$$\nu Z_1.\Phi_1(Z_1, \nu Z_2.\Phi_2(Z_1, Z_2, \mu Y_1.\Psi_1(Y_1, \mu Y_2.\Psi_2(Y_1, Y_2)))) \tag{2.1}$$

$$\mu Y.\Phi(Y, \nu Z.\Psi(Y, Z)) \tag{2.2}$$

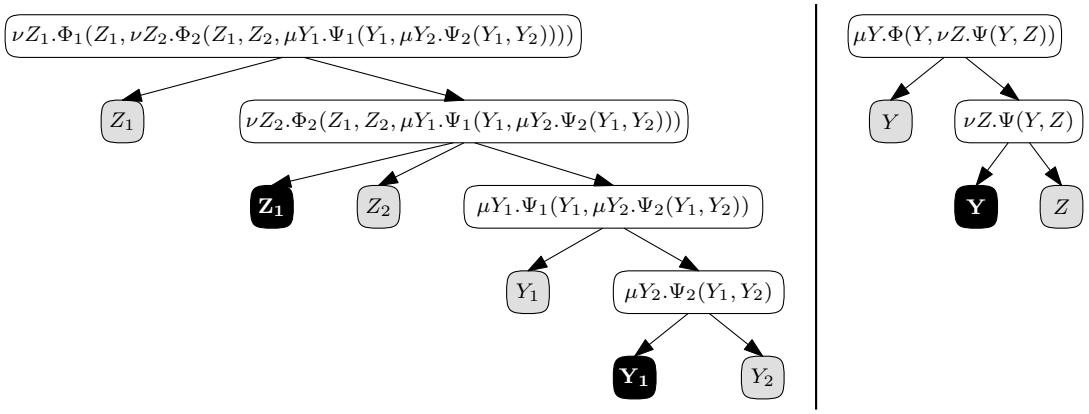


Figure 2.8: Alternation versus alternation-free μ-calculus formulae

Figure 2.8 helps illustrate the occurrence of variables within sub-formulae. Grey boxes represent variables that are bound within their sub-formula, while black boxes represent variables that are free within their sub-formula. Note that for logical statement 2.2, Y is a free variable in sub-formula $\nu Z.\Psi$ and a bound variable in sub-formula $\mu Y.\Phi$. This contradicts the definition of an alternation-free formula. As for logical statement 2.1, Figure 2.8 shows that Z_1 is a free variable in sub-formula $\nu Z_2.\Phi_2$; however, this does not contradict the definition of alternation-free, since Z_1 is not a bound variable in any sub-formula of the form $\mu X.\phi$. Similarly, Y_1 is also a free variable in sub-formula $\mu Y_2.\Psi_2$, but it also does not exist as a bound variable in any sub-formula

of the form $\nu X.\phi$. Therefore, logical statement 2.1 fulfils the alternation-free requirements.

In other words, we can summarise that a formula Φ is considered to be alternation-free if 'no sub-formula Ψ of a formula Φ contains both a free variable X bound by a μX in Φ as well as a free variable Y bound by a νY in Φ' (Bollig et al., 2002).

But what about the expressive power of μMA? Bradfield (1998) has proven that the expressive power of the μ-calculus does indeed grow with the alternation depth of its formulae. In other words, the complexity of these formulae is measured by their alternation depth, and the complexity of model checking such formulae is exponential in the alternation depth (Edmund M. Clarke et al., 1999, p. 108). Nevertheless, it has also been proven that most common modal and temporal logics, such as CTL, PDL, ACTL, may be expressed in the alternation-free μ-calculus with alternation depth 1 (Mateescu, 2003), and a maximum of alternation depth 2 is required to express CTL* and LTL formulae (Leucker et al., 2003). As a result, the user need not bother with notoriously complex μ-calculus formulae with an alternation depth greater than 2. The expressive power of the alternation free μ-calculus and its existing efficient model checking algorithms has turned the μMA into a popular specification language in the field of software and hardware verification.

2.4 Deontic Logic and Policy Languages

Similar to traditional model checkers, the model checker proposed by this thesis uses a process calculus (LCC) for specifying system models and a temporal logic (μ-calculus) for the property specification. These languages were introduced earlier in Sections 2.2 and 2.3, respectively. However, Chapter 3 introduces yet a third input to our proposed model checker: the agent's constraints. These constraints are specified via a deontic based policy language (DPL), introduced in Section 3.4.2. This section aims at providing the background behind the DPL language, starting with a modest introduction to deontic logic (Section 2.4.1) followed by a brief overview of policy languages (Section 2.4.2).

2.4.1 Deontic Logic

Deontic logic is the logic of duties. It deals with concepts like permissions, prohibitions, obligations, etc. It may be viewed as one way of defining social norms by

specifying who can do what. The Stanford Encyclopedia of Philosophy[5] states that deontic logic is the logic that deals with notions of what is:

permissible (permitted)	must
impermissible (forbidden, prohibited)	supererogatory (beyond the call of duty)
obligatory (duty, required)	indifferent / significant
gratuitous (non-obligatory)	the least one can do
optional	better than / best / good / bad
ought	claim / liberty / power / immunity

In practise, however, research on deontic logic has focused on only five of these notions: permissions, prohibitions/forbiddance, obligations, gratuitousness, and indifference. This section follows this common line of research by focusing on these five deontic operators.

2.4.1.1 The Deontic Operators

Deontic logic may be viewed as a special type of modal logic. An act a that is obligatory can be defined in terms of the modal logic necessity operator $\Box a$. What is permissible, or possible, can be defined in terms of the modal logic possibility operator $\Diamond a$, which is equivalent to $\neg\Box\neg a$. What is forbidden is viewed as that act whose negation is obligatory , i.e. it may be defined in terms of $\Box\neg a$. What is gratuitous is that which is not obligatory, i.e. it is defined as $\neg\Box a$. Finally, what is indifferent is both possible and gratuitous, i.e. it is equivalent to $\neg\Box\neg a \wedge \neg\Box a$.

In what follows, we use the operators O, $\mathcal{P}$, $\mathcal{F}$, $\mathcal{G}$, and $\mathcal{I}$ to represent obligations, permissions, forbiddance, gratuitousness, and indifference, respectively. It is common to pick one of the first four operators to be the basic operator, and define all remaining operators in terms of it. In this chapter, we choose the obligation operator O to be our basic operator. The remaining four operators are then defined as follows:

$$\begin{aligned} \mathcal{P}a &\equiv \neg O\neg a \\ \mathcal{F}a &\equiv O\neg a \\ \mathcal{G}a &\equiv \neg Oa \\ \mathcal{I}a &\equiv \neg O\neg a \wedge \neg Oa \end{aligned}$$

[5]The Stanford Encyclopedia of Philosophy's entry on deontic logic may be found at: http://plato.stanford.edu/entries/logic-deontic/. Interesting remarks on the challenges of defining deontic logic may be found at: http://plato.stanford.edu/entries/logic-deontic/challenges.html.

To achieve a better understanding of the deontic operators we refer to set theory. If actions were divided into sets according to what is permissible $\mathcal{P}$, obligatory $\mathcal{O}$, forbidden $\mathcal{F}$, gratuitous $\mathcal{G}$, and indifferent $\mathcal{I}$, then the result would best be presented by Figure 2.9.

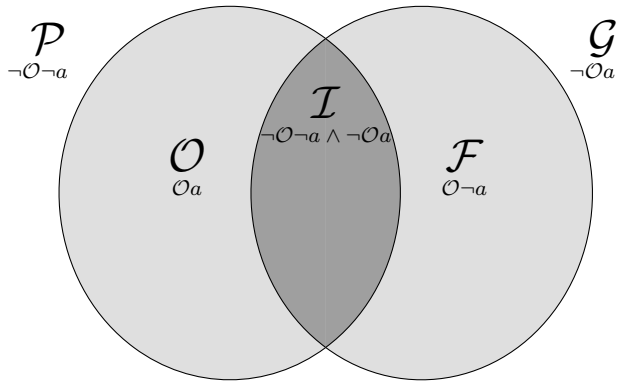


Figure 2.9: Deontic sets

There are two main sets $\mathcal{P}$ and $\mathcal{G}$ describing what is permissible and what is gratuitous, respectively. The set of obligatory actions $\mathcal{O}$ is equivalent to the complement of $\mathcal{G}$ in $\mathcal{P}$. Similarly, the set of forbidden actions $\mathcal{F}$ is equivalent to the complement of $\mathcal{P}$ in $\mathcal{G}$. Finally, the set of indifferent actions $\mathcal{I}$ is essentially the intersection of sets $\mathcal{P}$ and $\mathcal{G}$.

2.4.1.2 Axioms and Rules

There are various systems defining deontic logic: Mally (1926)'s deontic logic, von Wright (1951, 1971)'s dyadic deontic logic, Hilpinen (1971)'s standard deontic logic, etc. Each of these specifies its own set of axioms and rules. We do not intend to provide a complete survey of deontic logic systems; however, to provide an insight of these systems, we introduce the semantics of the Standard Deontic Logic (SDL), the "benchmark system of deontic logic"[6]. The axioms and rules of SDL are presented in Figure 2.10.

The first axiom (**TAUT**) is common to all logics. The second (**K**) is a well known property of modal logics in general. In the context of deontic logic, it specifies that if there is an obligation to $p \rightarrow q$ and there is an obligation to p, then there is an obligation to q. The third axiom is called (**D**) for deontic[7]; it states that if there is

[6] As described by the Stanford Encyclopedia of Philosophy.

[7] In fact, the standard deontic logic SDL is sometimes simply referred to as system **KD** or even system **D**, in reference to its axioms.

Axiom1 :	All tautologous wffs of the language	(**TAUT**)
Axiom2 :	$O(p \rightarrow q) \rightarrow (Op \rightarrow Oq)$	(**K**)
Axiom3 :	$Op \rightarrow \neg O \neg p$	(**D**)
Rule1 :	$\frac{p,\ p \rightarrow q}{q}$	(**MP**)
Rule2 :	$\frac{p}{Op}$	(**O – NEC**)

Figure 2.10: SDL: the system's axioms and rules

an obligation to p then p must be permitted, i.e. there is no obligation for $\neg p$. The inference rules of this logic are the propositional calculus modus ponens rule (**MP**) and the obligation necessitation rule (**O-NEC**), which states that if p holds then the obligation to p also holds.

Several theorems and rules may be derived from the system presented in Figure 2.10. For instance, the following rule $\frac{p \rightarrow q}{Op \rightarrow Oq}$, named (**O-RM**), may be derived as follows:

$$\frac{\dfrac{p \rightarrow q}{O(p \rightarrow q)}\ (\mathbf{O-NEC})}{Op \rightarrow Oq}\ (\mathbf{K})$$

Applying the same proof to the tautology $p \rightarrow p \vee q$ results in the corollary $Op \rightarrow O(p \vee q)$. However, Ross (1941) identifies a paradox resulting from this corollary. He points out that this corollary results in statements such as "If I ought to mail a letter, then I ought to mail or burn it", which is clearly unacceptable.

The paradox presented above is only one of the numerous paradoxes of deontic logic. Deontic logic systems are famous for the paradoxes they raise. However, we do not dwell on these, since this thesis does not make use of a deontic logic system, but focuses on the general concepts and ideas of deontic logic. In other words, our definition of deontic logic follows the basic, and possibly over simplified, definition that focuses solely on the categorisation of obligations, permissions, and prohibitions. Section 3.4.2, which defines our deontic based policy language, revisits this issue discussing why such paradoxes are irrelevant to our work.

2.4.2 Policy Languages

Policy languages have been used widely in hardware systems and networks for security reasons, trust negotiation, access control, authorisation, etc. Sloman (1994) defines policies to be "one aspect of information which influences the behaviour of objects within the system". Damianou et al. (2002) categorises policies into two types. The first is the *obligation policies* for managing actions. These are usually event triggered condition-action rules. The basic concept is that specific events trigger specific actions, and the actions may only be executed if a predefined set of conditions is satisfied. The second type is the *authorisation policies*, which are usually used for access control. Deontic concepts that are based on permissions and prohibitions are usually used here. Since it is not practical to define policies relating to individual agents, policies are defined in the context of roles and organisational groups. For this reason, policy languages may be viewed as yet another method for the specification of social norms.

Policy languages have been defined for dealing with a variety of issues. For example, the ASL policy language (Jajodia et al., 1997) is strictly a security policy. Ponder (Dulay et al., 2002) addresses both security and management issues. The Rei policy language (Kagal et al., 2003) is a general purpose policy language that supports security issues, management, conversations, etc. Policy languages, like any other programming language, may be logic-based (e.g. ASL (Jajodia et al., 1997), Rei's Prolog implementation (Kagal et al., 2003), RDL (Hayton et al., 1998)), object-oriented (e.g. Ponder (Dulay et al., 2002), RuleML(Boley, 2003)), based on markup languages (e.g. Rei's RDF implementation(Kagal et al., 2003), XACML(Oasis, 2003), TPL (Herzberg et al., 2000)), etc.

The literature contains a huge collection of policy languages with different colours and flavors. Nevertheless, all policies are essentially (in their simplified form) a tuple of the form $(S, O, \langle Sign \rangle A)$ which permits or prohibits — depending on the sign of A — a subject S from executing an action A on an object O. Obligations are usually event-triggered rules. Additionally, conditions may be attached to these rules. Available policy languages are basically more specialised versions of the above, each with its own conflict resolution mechanism.

In Section 3.4.2, we propose a deontic based policy language for modelling agent constraints. However, to help the reader acquire a taste of existing policy languages, we introduce two of the literature's languages: Ponder and Rei.

2.4.2.1 Ponder

Ponder is a policy language used for access control as well as defining other actions agents may perform. The basic unit in a system is an object which could be an agent, a resource, or a person. Objects are then grouped into domains. This provides a hierarchical structuring of objects. Policies are applied to objects as well as domains. There are four different types of policies whose rules are presented by Figure 2.11. Note that the meaning of each rule is described under the "Description" column.

Policy Type	Abstract Form	Enforced by	Description
Authorisation +	$S \rightarrow T.A \ ? \ C$	Target	Subject S is authorised to invoke action A on target T if condition C holds.
Authorisation −	$S \nrightarrow T.A \ ? \ C$	Target	Subject S is not authorised to invoke action A on target T if condition C holds.
Refrain	$S \mapsto T.A \ ? \ C$	Subject	Subject S must refrain from invoking action A on target T if condition C holds.
Delegation +	$G \ [S \rightarrow T.A] \rightarrow H \ ? \ C$	Policy System	Given an existing authorisation policy $S \rightarrow T.A$, then the set of grantors $G \subseteq S$ are authorised to create a new authorisation policy $H \rightarrow T.A$ if the condition C holds, delegating their authorisation to a set of grantees H.
Delegation −	$G \ [S \rightarrow T.A] \nrightarrow H \ ? \ C$		Given an existing authorisation policy $S \rightarrow T.A$, then the set of grantors $G \subseteq S$ are not authorised to create a new authorisation policy $H \rightarrow T.A$ if the condition C holds.
Obligation	$E, \ S \rightarrow M \ ? \ C$	Subject	On event E, subject S is obliged to execute method M if condition C holds.

Figure 2.11: Ponder's basic policy types

Ponder creates different policies for different domains. For example, the negative authorisation policy is used by targets to protect themselves against unauthorised subjects, while refrain policies are enforced by subjects who want to avoid certain targets,

such as untrusted resources.

Authorisation and refrain rules are usually used to specify access control policies. Obligations may also be specified through event triggered rules. These are especially helpful for managing the system and controlling the system behaviour. Delegation policies are used to specify who can introduce specific policies to the system.

Conflicts between policies may arise. One solution is to explicitly define precedence rules. Another is to rely on more general rules, such as giving a negative authorisation a precedence over positive authorisations. Meta policies may also be used to test for conflicts and define application specific precedence rules. Please refer to Dulay et al. (2002) for further information on Ponder and its meta policies.

2.4.2.2 Rei

Rei is based on deontic concepts. It includes constructs for rights, prohibitions, obligations and dispensations. It allows the specification of different kinds of policies: security, privacy, management, conversations, etc. Figure 2.12 presents the structure of the Rei policy language, excluding meta information (meta policies).

Policies are specified via a set of *has* constructs that assign policy objects to entities (or subjects). There are four kinds of policy objects: rights, prohibitions, obligations, and dispensations. All policy objects are composed of an action and a set of conditions that need to be satisfied for this action to take place. Conditions may be compound, making use of the *and*, *or*, and *not* logical operators. Actions are usually defined through an identifier, a target object, a set of pre-conditions to be satisfied, and a set of post-conditions/effects that should hold as a result of the action being performed. However, a different type of actions also exist in Rei. These are known as speech acts, which are common in the field of multiagent systems. They are composed of a sender, a receiver, as well as a right or another action. They are used to allow a sender to delegate its right to someone else, request a right, revoke someone's right, or even cancel earlier requests. As normal actions, speech acts also require the sender to have the right to send a given speech act.

Again, conflicts between policies may arise. Similar to Ponder, Rei uses meta policies to resolve conflicting policies by either giving priorities for one rule over the other (e.g. $overrides(A, B)$), or by specifying precedence rules. Precedence rules define whether positive or negative modalities holds. In a positive modality, precedence is given to rights and obligations over prohibitions and dispensations. The opposite is true in a negative modality.

$$
\begin{array}{rcl}
Policy & := & \{has(Subject, PolicyObject), \ldots\} \\
PolicyObject & := & right(Action, Conditions) \quad | \\
 & & prohibition(Action, Conditions) \quad | \\
 & & obligation(Action, Conditions) \quad | \\
 & & dispensation(Action, Conditions) \\
Action & := & SpeechAct \quad | \\
 & & ActionDef \quad | \\
 & & seq(Action, Action) \quad | \\
 & & nond(Action, Action) \quad | \\
 & & repetition(Action) \quad | \\
 & & once(Action) \\
SpeechAct & := & delegate(Sender, Receiver, right(Action, Conditions)) \quad | \\
 & & request(Sender, Receiver, right(Action, Conditions)) \quad | \\
 & & request(Sender, Receiver, Action) \quad | \\
 & & revoke(Sender, Receiver, right(Action, Conditions)) \quad | \\
 & & cancel(Sender, Receiver, right(Action, Conditions)) \quad | \\
 & & cancel(Sender, Receiver, Action) \\
ActionDef & := & action(ActionName, TargetObjects, PreConditions, Effects) \\
Conditions & := & Term \quad | \\
 & & not(Conditions) \quad | \\
 & & Conditions, Conditions \quad | \\
 & & Conditions; Conditions \\
PreConditions & := & Conditions \\
Effects & := & Conditions \\
Sender & := & Subject \\
Receiver & := & Subject
\end{array}
$$

where,
Term is a structured term in Prolog syntax,
ActionName is a unique action identifier, and
Subject and *TargetObjects* are either variables or unique object identifiers.

Figure 2.12: Rei's policy structure

Chapter 3

Multiagent System Verification: the MCID Model Checker

This thesis is concerned with the problem of obtaining predictable, reliable interactions between group of agents in open environments. The most popular approaches to this in practise have been to model interaction protocols and to model the deontic constraints imposed by individual agents. Both of these approaches are important, but their combination creates the practical problem of ensuring that interaction protocols are meshed with agents that possess compatible deontic constraints. The main question this thesis addresses is: *"Given an interaction model and an agent with given deontic constraints wishing to participate in that model, could that combination work?"* This is essentially an issue of property checking dynamically at run time. To check whether the agent may successfully engage in such a protocol, the agent is allowed to specify its requirements from the interaction and to prove that these requirements are not violated by anything that can be currently inferred from the interaction model. This chapter shows how model checking can be applied to this problem.

The chapter opens with a discussion of modelling multiagent systems (MAS) in Section 3.1. This is followed by a motivating scenario in Section 3.2. The design and implementation plans of the verifier are presented in Section 3.3, followed by the verifier's specification languages in Section 3.4, the technical details of the verification process in Section 3.5, and a conclusion in Section 3.7.

3.1 MAS Modelling: an Interaction Based Approach

One of the most fundamental issues of multiagent system engineering is to enable predictable, reliable interactions. The challenge is in doing so without requiring a deep standardisation of the way in which the agents are engineered, yet preserving as much as possible their autonomy in individual reasoning. It is not plausible that agents, built independently and with no agreement on forms of interaction, could be predictably reliable. Researchers have searched for ways of standardising some aspects of coordination in the hope that this small amount of standardisation would provide predictability sufficient for important tasks. Two contrasting approaches to this problem have emerged.

Specification of global interaction models: The first approach focuses on engineering the system in such a way that it is capable of providing reliable interactions without having to deal with the complication of engineering the various agents that might engage in such interactions. It does so through the specification of explicit models of interactions, describing how the interaction may be carried out. These are detached from individual agents and are typically accessed when an agent anticipates it wants to initiate or join such an interaction. Organisational approaches (Horling and Lesser, 2005), electronic institutions (Esteva et al., 2001), and distributed dialogues (Robertson, 2004b) are all examples of such approaches.

Specification of agents' local constraints: The second approach focuses on engineering intelligent autonomous agents that are capable of shaping and controlling interactions through the explicit specification of their constraints. These constraints are imposed by individual agents on the actions they will or will not allow. They are typically accessed when a specific interaction is anticipated with the individual agent. They are built locally for an individual agent through BDI models (Mascardi et al., 2005), deontic constraints (Meyer and Wieringa, 1993), policy languages (Damianou et al., 2002), etc.

Just as in human societies, we believe both approaches are important; therefore, we propose a system model that meshes them together. In human societies, for instance, there exists a set of social norms described via organisational rules, contracts and deals, judiciary laws, etc. As humans, we also have a choice in deciding who to interact with and how to interact with them. For example, it is up to the person to decide whether they will have an AOL internet connection or not. However, if they do, then this implies that they agree to AOL's terms and conditions. This shows that autonomous

individuals with private constraints will still need to abide to global 'social' constraints when interacting with others.

This thesis focuses on open and distributed systems, whose entities are autonomous agents. In such systems, various agents may join or leave the system at any time. Interactions become the backbone that holds the system together. Agents group themselves into different, and possibly multiple, interactions. Figure 3.1 provides an example of a collection of agents that are grouped into three different interactions (or scenarios).

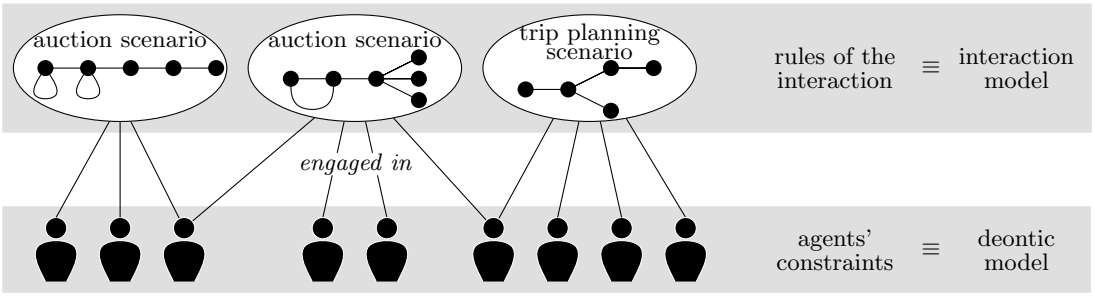


Figure 3.1: Proposed MAS model: a 2-layered architectural model

Due to the dynamic nature of the system, we believe interaction groups should be created dynamically and automatically by the agents. It is the agents' responsibility to group themselves into various scenarios. There should be no higher layer for coordination, control, synchronisation, etc. As a result, multiagent systems are modelled through two layers only: the interaction and agents layers.

The interaction model specifies the rules and constraints on the interaction, which indicates how exactly the interaction may be carried out. It is specified via a generic process or state-machine language, and it is independent of the agents that might engage in the interaction. The agents' model specifies the rules and constraints on the agents. These are the agents' permissions, prohibitions, and obligations; therefore, we call this model the deontic model. Note that for one scenario there is one global interaction model and several local deontic models.

We propose the 2-layered architectural model approach while taking into consideration the issue of verifying such systems, which is the main goal of this thesis. Section 3.3 presents the advantages of using such a model for multiagent system verification. However, we first provide a motivating example in Section 3.2, which is used for illustration purposes in the remainder of this chapter.

3.2 Motivating Example

The example used throughout this chapter is a slightly modified version of the travel agent use case of He et al. (2004). Figure 3.2 presents an overview of the interaction. While the customer agent interacts with the travel agent, the latter contacts airline Web services and queries a hotel directory to provide the customer agent with a list of available flights and hotel offers. Also, the travel agent communicates with a credit card Web service to process the customer's payment, if a deal is made.

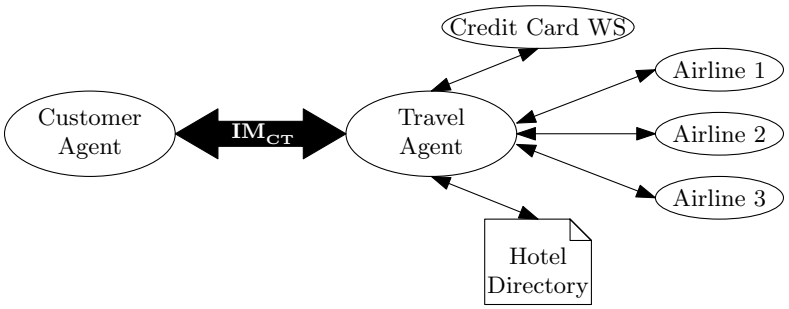


Figure 3.2: Travel agency scenario: an overview (He et al., 2004)

Figure 3.3 defines the interaction model corresponding to the communication between the customer and the travel agent ($\mathbf{IM_{CT}}$ of Figure 3.2). The interaction starts when a customer agent C provides the travel agent T with its vacation's start date SD, end date ED, and its destination D. The travel agent forwards this information to the airline Web services As for retrieving a list of possible flights FL, which is forwarded to the customer agent. After the customer selects a flight Fx, the travel agent searches the hotel directory HD and sends a detailed list of hotel options HL back to the customer. The customer selects a hotel option Hx, the travel agent computes the total amount TA to be paid, and the customer sends back its payment details PD. The travel agent verifies the payment details with the credit card Web service CD, which provides a reason R for payment failure, if any. If the payment is authorised, the travel agent pays the appropriate airline A and hotel H, and confirms the booking by sending the receipts $Furl$ and $Hurl$ back to the customer. Otherwise, the customer is informed of the failure and it might either choose to quit the interaction or retry the payment. Note that for the purpose of keeping the example simple and clear, we assume no errors in retrieving the list of flights and hotels. We also assume the customer will always select flight and hotel options from the lists provided by the travel agent. However, we do model the possible failure in the customer's payment activity.

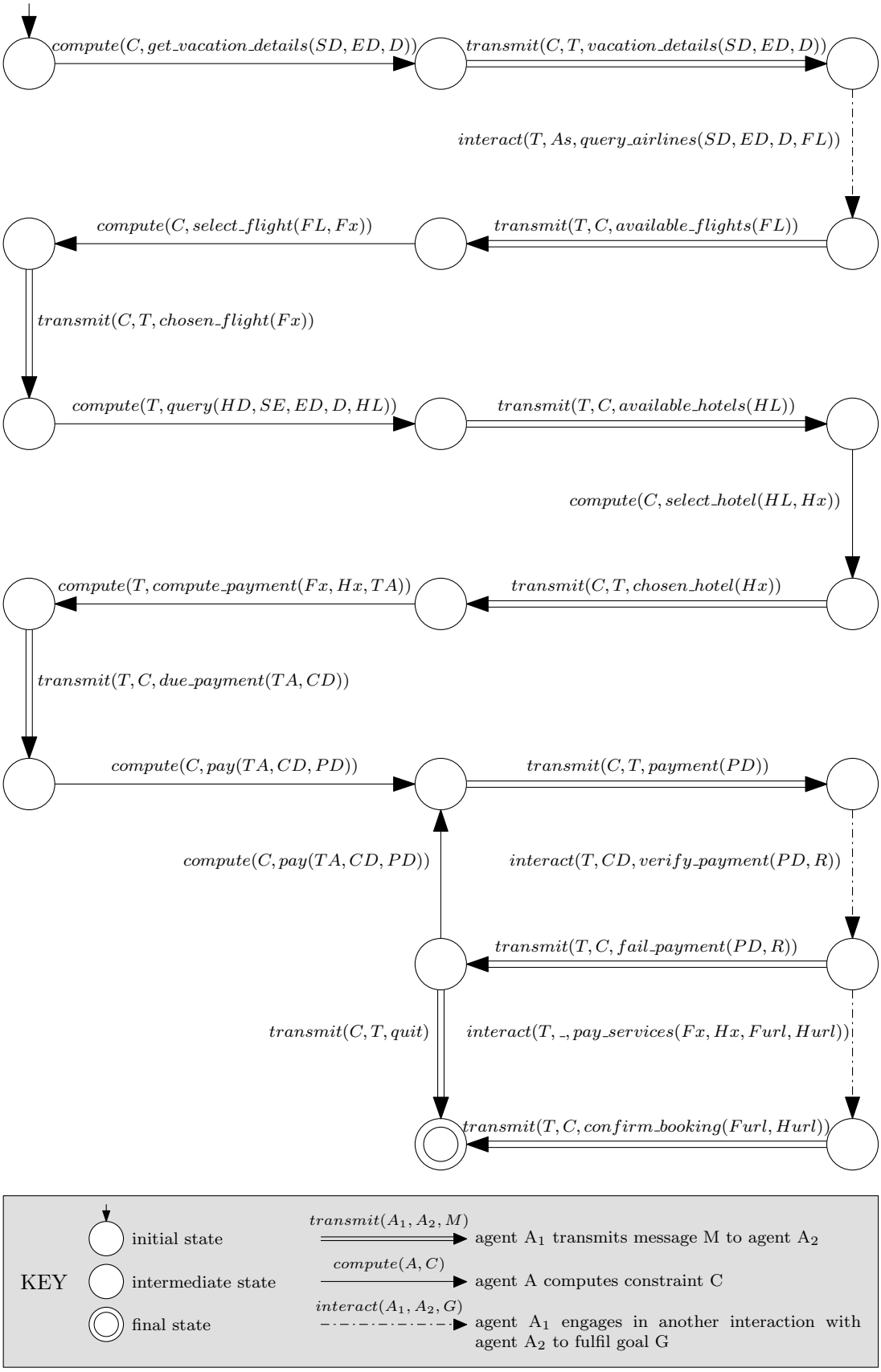


Figure 3.3: Travel agency scenario: interaction's state graph of $\mathbf{IM_{CT}}$ of Figure 3.2

In addition to the interaction rules of Figure 3.3, the agents involved in this interaction may also lay down their own set of restrictions: their deontic constraints. Figure 3.4 provides a sample of such rules. For instance, while the travel agent should be capable of accessing the hotel directory, the hotel directory might place a constraint permitting only members of the Student and Youth Travel Association (SYTA) to access it (deontic rule $\mathcal{D}1$ of Figure 3.4). The customer should also be a customer of the selected credit card Web service, otherwise it will not be capable of making any payments (deontic rule $\mathcal{D}2$ of Figure 3.4). For transmitting payment details, the customer might not be capable of using any encryption other than the *OpenPGP* encryption (deontic rule $\mathcal{D}3$ of Figure 3.4). The travel agent might require the credit card Web service to authenticate itself with a *X*.509 certificate (deontic rule $\mathcal{D}4$ of Figure 3.4). Note that these rules address various issues, such as access control (deontic rule $\mathcal{D}1$), authentications and trust issues (deontic rule $\mathcal{D}4$), security issues (deontic rule $\mathcal{D}3$), etc.

$\mathcal{D}1$: The hotel directory allows the travel agent to query it only if it is a member of SYTA.

$\mathcal{D}2$: The credit card web service requires the customer of the travel agency scenario to be its own customer for it to be capable of making a payment.

$\mathcal{D}3$: The customer can not perform any encryption other than *OpenPGP*.

$\mathcal{D}4$: The travel agent requires that the credit card web service to be capable of authenticating itself with a *X*.509 certificate.

Figure 3.4: Travel agency scenario: some deontic constraints

Now assume that the customer agent is ignorant of the available services in the system. Therefore, it approaches a broker agent for finding suitable agents for this scenario. With the broker being responsible for finding suitable agents, it should also be capable of verifying at interaction time that the protocol it has instantiated with these agents is likely to work. This implies that it should verify that the agents it has selected are compatible with each other as well as compatible with the given interaction model. For example, trying to ally the customer agent with a travel agent that does not have access to a hotel directory will result in a scenario failure, regardless of whether the interaction protocol itself is error free or not. As a result, the verifier should handle both interaction and deontic constraints, and should be capable of operating automatically at run time. The broker can then use such a verifier to verify an instance of the interaction protocol, i.e. the interaction protocol instantiated for the selected group of agents.

3.3 Multiagent System Verification

After presenting an overview of multiagent system models in Section 3.1 followed by a motivating example in Section 3.2, we now present an overview of our proposed model for verifying multiagent systems. We first discuss the reasons behind our decision to verify multiagent system models designed according to the 2-layered architectural MAS model of Section 3.1. This is followed by a presentation of the design and implementation plans for building such a verifier.

3.3.1 Verification of Interaction and Deontic Models

Verifying multiagent systems has traditionally focused on verifying systems built via one of the two approaches presented in Section 3.1. Verifying interaction models (e.g. Walton, 2004; Wen and Mizoguchi, 1999) is important, yet not sufficient on its own when one is interested in verifying dynamic properties depending on the agents engaged in the interaction. It is essential to make sure the interaction model is reliable; however, predicting the success of interactions is impossible in such cases, since this is usually highly affected by the autonomous agents executing the interaction. On the other hand, verifying solely the agents' constraints, such as their BDI model (e.g. Wooldridge et al., 2002; Bordini et al., 2003b, etc.), is a highly complex task. The global interaction is usually built from the implicit interaction constraints within agent specifications. If the constraints are not explicit enough to specify and direct the interaction (and, in distributed open systems, usually they are not), then this may result in verifying possibly an infinite number of permissible interactions. Furthermore, these verification techniques assume that the agents' BDI model may be accessible to the verifier. This might be feasible in closed systems were all agents have been engineered in the same way and allow their BDI models to be made public. However, this is very hard to achieve in distributed open systems, where agents could not and should not permit access to their BDI models. It is not likely that an agent would make its goals, intentions, and plans public. More importantly, it would not be easy for agents to extract such information from their specification. Usually this information is not specified declaratively, but rather it is implicit in the procedural implementation of the agent. Finally, we believe verification should not be about predetermining how agents will act in response to each action. Instead, verification should simply focus on whether conflicts might arise from a given interaction model with a given set of collaborating agents. For this, we distinguish between an agent's BDI layer and its deontic one. While the

BDI layer is inaccessible, the deontic one specifies the constraints that the agents might want to make public. For instance, while a bidder agent in an auction system should not reveal its bidding strategy, it would be important to make some of its constraints public, such as its inability to pay via *PayPal*. Figure 3.5 illustrates the three layers affecting an interaction. Note that our proposed 2-layered architectural model neglects the inaccessible BDI layer.

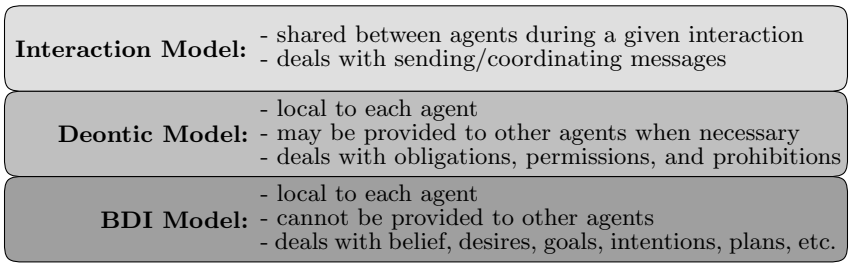


Figure 3.5: The 3-architectural layers affecting interactions in MAS

Verifying the combination of global interaction models along with the agents' local constraints in distributed open systems is more efficient than solely verifying agents' constraints, since we now have an explicit interaction model that limits the number of possible interactions. Moreover, the results of such a verification method are more predictable and reliable, since they take into consideration the current agents engaged in the given interaction and how they might affect and direct this interaction. Of course, this requires verification to be performed by the agents at run time when the conditions for verification are met and the related deontic models are obtained.

3.3.2 The Verifier's Design and Implementation Plans

As illustrated by the travel agency scenario of Section 3.2, the broker agent will need to verify the interaction protocol against the selected agents' deontic rules, so that a team of collaborating agents is reached. In general, in distributed open systems consisting of autonomous agents, it is necessary for agents to be capable of automatically verifying at run time dynamic interaction and deontic models of multiagent systems. Figure 3.6 illustrates the plan for implementing such a verifier.

We choose model checking from amongst other verification techniques, such as those presented by Section 2.1.1, because it provides a fully automatic verification process which could be carried out by the agents at interaction time. The result is the MCID verifier for model checking interaction and deontic models.

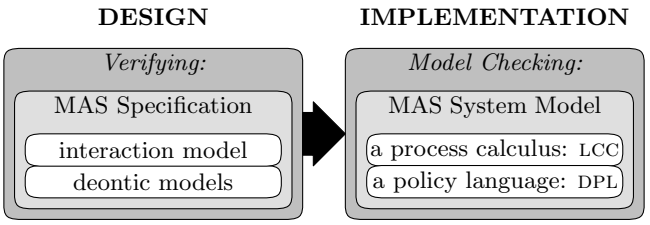


Figure 3.6: Verifier's design and implementation plans

Our system model is a bundle of interaction and deontic rules. For specifying interaction protocols, which deal with coordinating messages between agents, we choose a process calculus. Process calculus is a calculus for representing concurrent and distributed processes. It accounts for the non-deterministic and non-terminating nature of these processes. Its success in efficiently describing the rules for coordination makes it especially appealing for specifying interaction protocols of multiagent systems. Policy languages, on the other hand, have been widely used in hardware systems and networks for expressing deontic rules: the rules of obligations, permissions and prohibitions. Policy languages address issues such as security, trust specification, authorisations, etc. This makes them good candidates for specifying agents' deontic rules.

Section 2.1.2 has already introduced the model checking mechanism in detail. In summary, the model checking problem may be defined as follows: *"Given a finite transition system S and a temporal formula ϕ, does S satisfy ϕ?"* Both the system model and the property specification are fed to the model checker for verification. The model checker is essentially an algorithm that decides whether a model S satisfies a formula ϕ. Figure 2.2 provided an overview of a typical model checker's input. This now needs to be modified, since our system model is composed of both an interaction and a deontic one. Figure 3.7 presents the modified version illustrating MCID's design. We note that one of the main contributions of this thesis lies in the addition of this new deontic model to the model checker's input.

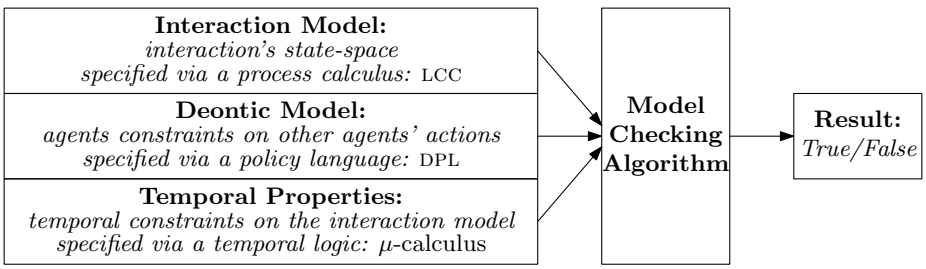


Figure 3.7: MCID's design

After laying down the foundations for our proposed model checker, Section 3.4 presents the specification languages selected for specifying the interaction model, the deontic model, and the temporal properties fed to the model checker. It is followed by a presentation of the model checking algorithm and formal semantics in Section 3.5.

3.4 Specification Languages

This section introduces the specification languages used by the MCID model checker. The first is the LCC process calculus for specifying interaction models, the second is the DPL policy language for specifying deontic constraints, and the third is the μ-calculus for specifying interactions' temporal properties.

3.4.1 Lightweight Coordination Calculus (LCC)

3.4.1.1 Why Choose LCC?

What is required is a language for specifying interaction models of multiagent systems. The literature provides a variety of solutions that deal with specifying and regulating interactions. These are mainly driven by the concept of following social norms (e.g. Shoham and Tennenholtz, 1995), such as having contracts and commitments (e.g. Dignum et al., 2002), organisational approaches (Horling and Lesser, 2005), electronic institutions (Esteva et al., 2001), distributed dialogues (Robertson, 2004b), etc. Having agents playing *roles* is a central issue in many of these approaches. The idea is that actions, tasks, and duties, along with constraints, are assigned to roles instead of agents. This provides an abstraction level of actions and specifications independent of the particular agents engaged in the given interaction. Agents may then play different roles and engage in different scenarios. For example, one agent might play the role of a bidder in some auction system, and the role of a seller in another. LCC is also based on the concept of agents playing roles and sharing a dialogue framework for achieving distributed coordination. It defines the interaction protocol without having to specify details of agents involved in this interaction, which is a requirement specified in Section 3.1. Nevertheless, the agents' autonomy in LCC is preserved with the heavy use of constraints. So while an LCC interaction protocol is independent of and deployed by possibly any agent, the constraints in LCC provide means for the agents to influence and direct the interaction as they please.

Since the interaction model is intended to be verified via a model checker (for reasons specified in Section 3.3), then the natural way to specify such a model is via a process calculus (also for reasons specified in Section 3.3). Process calculi are usually used for modelling concurrent systems, providing a way for abstracting the interaction and synchronisation details from the low-level technical details of the system. For this reason, process calculi have not been traditionally used in building the executable models of these systems. However, since LCC was created for the purpose of specifying solely interaction models, it was decided that the same language is to be used in the executable model as well. This makes LCC an ideal candidate for our model checker's description language. Having the executable model fed directly to the model checker avoids the complexity of modelling the system in another language and the possibility of introducing errors in doing so. Eliminating this step also contributes to the remarkably small size of the model checker. Furthermore, some properties to be verified may require the agents to automatically invoke the verifier at run time when the conditions for verification are met. The LCC protocol, capturing the actual system to be verified, is directly fed to the model checker. Given the nature of LCC's clause expansion mechanism (Section 2.2.3), agents are capable of automatically extracting the current state of the interaction and directly feeding it to the model checker when necessary.

3.4.1.2 Syntax and Semantics

We refer the reader to Section 2.2.2, where the syntax and semantics of LCC have been thoroughly illustrated. Figure 2.3 provided a summary of this. As an example, the following section presents a specification of the travel agency scenario in LCC.

3.4.1.3 Example: the travel agency scenario

To illustrate the specification of multiagent systems via LCC, let us consider a section of the travel agency scenario. Figure 3.8 models the interaction between the customer and the travel agent described earlier in Figure 3.3. The first two clauses specify the interaction rules of the two roles played by the customer agent: *customer* and *paying_customer*. The interaction starts when the agent retrieves its vacation details: the vacation's start date SD, end date ED, and destination D. It then sends these details to the travel agent, receives a list of available flights FL, selects an appropriate flight Fx, sends its selected flight to the travel agent, receives a list of available hotel options HL, selects a hotel Hx, sends its selected hotel to the travel agent, receives

$a(customer(T), C)$ ::
$vacation\_details(SD, ED, D) \Rightarrow a(travel\_agent(\_, \_, \_), T)$
$\leftarrow get\_vacation\_details(SD, ED, D)$ then
$available\_flights(FL) \Leftarrow a(travel\_agent(\_, \_, \_), T)$ then
$chosen\_flight(Fx) \Rightarrow a(travel\_agent(\_, \_, \_), T) \leftarrow select\_flight(FL, Fx)$ then
$available\_hotels(HL) \Leftarrow a(travel\_agent(\_, \_, \_), T)$ then
$chosen\_hotel(Hx) \Rightarrow a(travel\_agent(\_, \_, \_), T) \leftarrow select\_hotel(HL, Hx)$ then
$due\_payment(TA, CD) \Leftarrow a(travel\_agent(\_, \_, \_), T)$ then
$a(paying\_customer(TA, CD, T), C)$.

$a(paying\_customer(TA, CD, T), C)$ ::
$payment(PD) \Rightarrow a(verify\_payment(\_, \_, \_, \_), T) \leftarrow pay(TA, CD, PD)$ then
($confirm\_booking(Furl, Hurl) \Leftarrow a(verify\_payment(\_, \_, \_, \_), T)$
or ($fail\_payment(PD, R) \Leftarrow a(verify\_payment(\_, \_, \_, \_), T)$ then
($a(paying\_customer(TA, CD, T), C) \leftarrow repay(TA, R, CD)$
or $quit \Rightarrow a(verify\_payment(\_, \_, \_, \_), T) \leftarrow \neg repay(TA, R, CD)$))).

$a(travel\_agent([As], HD, CD), T)$ ::
$vacation\_details(SD, ED, D) \Leftarrow a(customer(\_), C)$ then
$a(query\_airlines([As], SD, ED, D, FL), T)$ then
$available\_flights(FL) \Rightarrow a(customer(\_), C)$ then
$chosen\_flight(Fx) \Leftarrow a(customer(\_), C)$ then
$null \leftarrow query(HD, SD, ED, D, HL)$ then
$available\_hotels(HL) \Rightarrow a(customer(\_), C)$ then
$chosen\_hotel(Hx) \Leftarrow a(customer(\_), C)$ then
$due\_payment(TA, CD) \Rightarrow a(customer(\_), C) \leftarrow compute\_payment(Fx, Hx, TA)$ then
$a(verify\_payment(CD, Fx, Hx, TA), T)$.

$a(verify\_payment(CD, Fx, Hx, TA), T)$ ::
$payment(PD) \Leftarrow a(paying\_customer(\_, \_, \_), C)$ then
$a(verify\_payment2(PD, CD, R), T)$ then
(($a(pay\_services(Fx, Hx, Furl, Hurl), T) \leftarrow R = success$ then
$confirm\_booking(Furl, Hurl) \Rightarrow a(paying\_customer(\_, \_, \_), C)$)
or ($fail\_payment(PD, R) \Rightarrow a(paying\_customer(\_, \_, \_), C) \leftarrow \neg R = success$ then
($a(verify\_payment(CD, Fx, Hx, TA), T)$
or $quit \Leftarrow a(paying\_customer(\_, \_, \_), C)$)).

Figure 3.8: Travel agency scenario: LCC interaction model of Figure 3.3

the bill of amount TA to be paid via credit card CD, and finally takes a different role *paying_customer* for paying its bill. The role *paying_customer* is responsible for retrieving the payment details, e.g. credit card number, expiry date, etc. Then it either receives a message confirming its bookings ($confirm\_booking$), or a message informing it of the reason R for the payment's failure. In the latter case, the agent might either decide to retry its payment ($a(paying\_customer(T, CD), C)$) or send a *quit* message to the travel agent to conclude the interaction.

Similarly, the last two clauses specify the travel agent's rules governing its interaction with the customer. To keep the example simple and short, Figure 3.8 omits role definitions dealing with interactions between the travel agent and the other agents in the system, e.g. $query\_airlines$, $get\_airline\_replies$, and $pay\_services$. Appendix A.1.1 presents a complete and more realistic version of the travel agency scenario.

3.4.2 Deontic-Based Policy Language (DPL)

Section 3.4.1 presented the LCC language used for specifying interaction models. In this section we propose a deontic-based policy language (DPL) for specifying the agents' deontic constraints.

3.4.2.1 Designing DPL

Agents usually have constraints on the actions they can or cannot perform in a given interaction. What is needed then is to link the deontic operators to the context of an interaction's state-space. Note that instead of choosing one of the available deontic based policy languages, we decide to design and define a new language that could directly be mapped to our interaction models. A comparison between DPL and other policy languages is provided in Section 3.8.

From Section 2.4.1, we know that deontic logic is based on the following five main operators: $\mathcal{P}$ for *permission*, $\mathcal{O}$ for *obligation*, $\mathcal{F}$ for *forbiddance* or *prohibition*, $\mathcal{G}$ for *gratuitousness* or what is *non-obligatory*, $\mathcal{I}$ for *indifference* or what is *optional*. Usually, one of the operators is taken as a basic operator and the remaining four are defined in its term. In what follows, we define the operators in terms of the permission operator $\mathcal{P}$ as follows: $\mathcal{P}a$ (action a is *permitted*), $\mathcal{O}a \equiv \neg\mathcal{P}\neg a$ (action a is *obligatory*), $\mathcal{F}a \equiv \neg\mathcal{P}a$ (action a is *forbidden*), $\mathcal{G}a \equiv \mathcal{P}\neg a$ (action a is *gratuitous*), and $\mathcal{I}a \equiv (\mathcal{P}a) \wedge (\mathcal{P}\neg a)$ (action a is *indifferent*).

We interpret these operators in the context of LCC interaction models. For example,

instead of talking about permissions in general, we state the actions a particular agent is permitted to do with respect to a particular interaction. The state-space, specified via the LCC process calculus, is a representation of the various worlds that may be realised. Each path in the state-space represents a possible world that may unfold. We define the relation of the deontic operators to the interaction model's state-space as follows[1]:

- **Permission:** action *a* is permitted if there exists at least one world/path where *a* is realisable
- **Obligation:** action *a* is obligatory if *a* is realisable in all worlds/paths
- **Forbiddance:** action *a* is forbidden if *a* is not realisable in all worlds/paths
- **Gratuitousness:** action *a* is gratuitous if there exists a world/path in which *a* is not realisable
- **Indifference:** action *a* is indifferent if there exists a world/path where *a* is realisable and there also exists a world/path where *a* is not realisable

Figure 3.9 provides a graphical representation of the relation between the deontic sets and the occurrence of the deontically constrained actions in an interaction's state-space. Note that if an action is realisable in a path, then the action may occur at some node in this path. However, if an action is not realisable in a path, then the action should not occur at any node in this path.

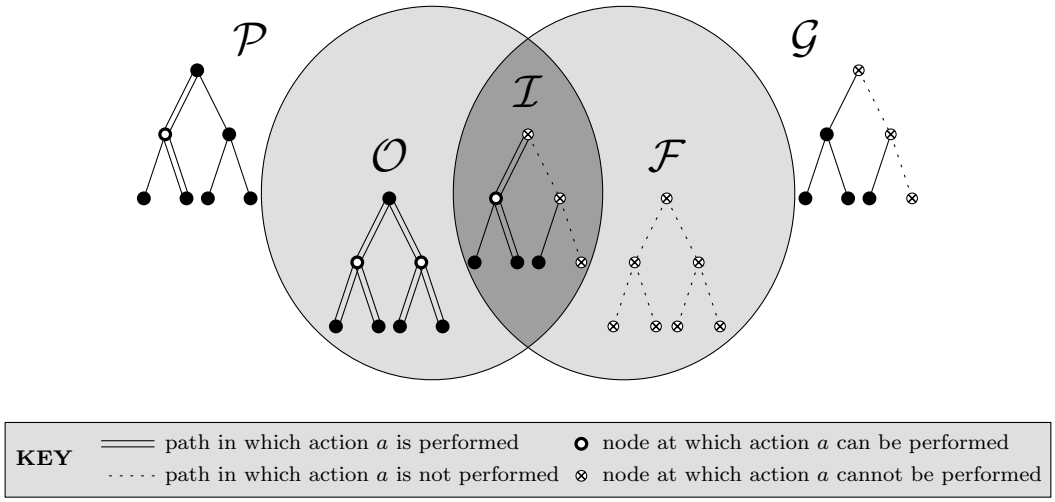


Figure 3.9: Relating deontic sets to the occurrence of actions in a state-space

[1]This definition stems from the definition of the deontic operators in Section 2.4.1.1.

3.4.2.2 Syntax and Semantics

We introduce a deontic-based policy language for specifying the five deontic operators $\mathcal{P}$, $\mathcal{O}$, $\mathcal{F}$, $\mathcal{G}$, and $\mathcal{I}$. Figure 3.10 presents the language's syntax. The deontic rule is specified by one of the following predicates: $must(Agent, Action, Sign)$ or $can(Agent, Action, Sign)$. '$Sign$' could take one of the following two values: '+' and '−'. As in LCC, the '*Agent*' is specified by a role and a unique identifier: $a(Role, Id)$. The '*Action*' could either be a message passing action (MPA) or a non-message passing action (N-MPA). An MPA is also specified in the LCC format: $Message \Leftarrow Agent$ and $Message \Rightarrow Agent$. An N-MPA represents some internal action or a computation the agent performs. Just like '*Role*' and '$Message$', an N-MPA is also a structured Prolog term.

$$
\begin{array}{rcl}
DeonticRule & := & must(Agent, Action, Sign)\,[\leftarrow Condition] \mid \\
 & & can(Agent, Action, Sign)\,[\leftarrow Condition] \\
Agent & := & a(Role, Id) \\
Sign & := & + \mid - \\
Action & := & MPA \mid N\!-\!MPA \\
MPA & := & Message \Leftarrow Agent \mid \\
 & & Message \Rightarrow Agent \\
Condition & := & Condition \wedge Condition \mid \\
 & & Condition \vee Condition \mid \\
 & & Temporal \mid \\
 & & Term \\
Role, N\!-\!MPA, Message & := & Term
\end{array}
$$

Figure 3.10: DPL syntax

Negative permissions, which represent both obligations '$\neg\mathcal{P}\neg a$' and forbiddance '$\neg\mathcal{P}a$', are specified via the '*must*' predicate of our policy language. For example, if an agent is obliged to perform action 'a', then we say: $must(agent, a, +)$. However, if the agent is forbidden to perform 'a', then we say: $must(agent, a, -)$.

On the other hand, positive permissions, which are either normal permissions '$\mathcal{P}a$' or gratuitousness '$\mathcal{P}\neg a$', are specified via the '*can*' predicate of our policy language. For example, if an agent is permitted to perform the action 'a' in a given interaction model, then we say: $can(agent, a, +)$. However, if the agent is gratuitous towards performing 'a', then we say: $can(agent, a, -)$. As for indifference, it is defined as the

conjunction of gratuitousness and permissions, i.e. a conjunction of negative and positive '*can*' predicates. We note that the semantics of these DPL predicates are better understood in the context of the occurrence of actions in interaction models, as illustrated by Section 3.5.3.1.

Additionally, conditions may be attached to these rules. The square brackets around '[← *Condition*]' imply that zero or one occurrence of this term is permitted. Conditions are usually composed of Prolog terms, and may possibly contain temporal properties. If a temporal property is used, then its syntax should follow the μ-calculus syntax of Figure 2.7. The addition of temporal properties to the conditions increases the richness of the properties that may be verified. For instance, one agent may decide that it can make a payment only if the interaction guarantees that a receipt will be sent.

3.4.2.3 Example: the travel agency scenario

In this section we specify the deontic constraints of Figure 3.4 in DPL. The results are presented in Table 3.1. The first constraint, $\mathcal{D}1$, states that the travel agent can query the hotel directory ($query(HD, \_, \_, \_, \_)$) only if it is a member of the student youth travel association '*syta*'. Querying the hotel directory should be an option in this interaction, since the customer is interested in booking a hotel. Therefore, the '*can*' predicate is used.

Similarly, constraint $\mathcal{D}2$ states that the customer can make a payment ($pay(\_, CD, \_)$), and for it to be able to pay, it should be a customer of the appropriate credit card service. However, it should not be obligatory for the customer to pay in every single path that may unfold. Therefore, the '*can*' predicate is used again.

Constraint $\mathcal{D}3$ states that the customer should not send any payment messages ($payment(\_) \Rightarrow a(\_, \_)$) if it performs its encryption in anything other than OpenPGP. A negative '*must*' rule is therefore used.

#	**DPL specification**
$\mathcal{D}1$	$can(a(travel\_agent(\_, \_, \_), T), query(HD, \_, \_, \_, \_), +) \leftarrow member(T, syta).$
$\mathcal{D}2$	$can(a(customer(\_), C), pay(\_, CD, \_), +) \leftarrow customer(C, CD).$
$\mathcal{D}3$	$must(a(customer(\_), C), payment(\_) \Rightarrow a(\_, \_), -) \leftarrow encrypt(X) \wedge X \neq openPGP.$
$\mathcal{D}4$	$must(a(credit\_card, CD), \_, +) \leftarrow authenticate(x.509).$

Table 3.1: Travel agency scenario: DPL specification of deontic rules of Figure 3.4

Constraint $\mathcal{D}4$ states that for the credit card web service to engage in this interaction (in other words, for it to be able to perform any action, which is specified via the underscore variable '_'), it should be capable of authenticating itself via the X.509 certificate. The '*must*' predicate is used to ensure this.

3.4.3 μ-Calculus

Sections 3.4.1 and 3.4.2 have presented the languages used for specifying the interaction and deontic models, respectively. The interaction model lays down the rules of the interaction. The deontic constraints are constraints on what the agents can or cannot do. However, we still need a language for specifying the properties expected from a given interaction model. These will be used to verify whether the selected interaction model fits certain requirements, usually describing liveness and safety properties. In model checking, these properties need to be expressed in a temporal logic. For this, we choose the μ-calculus, which has been introduced earlier in Section 2.3.3. This section discusses the reasons behind our selection and provides examples of temporal properties that could be verified for our travel agency scenario.

3.4.3.1 Why Choose the μ-Calculus?

What is needed is a branching time temporal logic for specifying properties of LCC's state-space. The literature provides us with several languages to select from; however, we choose the μ-calculus for one main reason: its simplex syntax. In μ-calculus, the branching operators are expressed in terms of the traditional modal operators. The *necessary* modal operator $\Box$ expresses the `A` temporal operator, which stands for '*for every path in the state-space*'. The *possible* modal operator $\Diamond$ expresses the `E` temporal operator, which stands for '*there exists a path in the state-space*'. All remaining temporal operators referring to time in a given path, such as *next* `N`, *finally/eventually* `F`, *globally* `G`, *until* `U`, and *release* `R`[2], are expressed in terms of recursion in the μ-calculus. Recursion is specified through the use of fixed point operators.

As illustrated by Figure 2.7, the μ-calculus is a logic with a concise syntax and expressive and powerful semantics. This is essentially the result of adding simple recursion operators (the fixpoint operators) to the basic $\Box$ and $\Diamond$ modalities. Our model checker is a logic-based one written in XSB tabled Prolog. This implies that the core of the model checker, which is responsible for proving the satisfaction of properties in a

[2]Section 2.3.2 presents an overview of the temporal operators `A`, `E`, `N`, `F`, `G`, `U`, and `R`.

given system model, is achieved by performing a direct translation from the μ-calculus semantics of Figure 2.7 to the model checker's satisfaction rules of Figure 3.16. The result is is a neat and compact model checker with a simple and straightforward algorithm. Section 3.5.2 discusses the model checker's algorithm in more detail.

To avoid undesirable looping behaviour, the use of XSB Prolog restricts the model checker to the alternation-free μ-calculus (Ramakrishna et al., 1997). That is because formulae with alternation result in loops through negation, which are not easily handled by XSB. In the alternation-free μ-calculus, nesting of minimal and maximal fixed-point operators is prohibited, as illustrated by Section 2.3.3.3. However, despite this limitation, the alternation-free μ-calculus remains expressive enough to subsume many common temporal logics, such as CTL and ACTL (Leucker et al., 2003). Moreover, the complexity of model checking is known to grow exponentially with the alternation depth (Leucker et al., 2003). For this reason, we believe the use of alternation-free μ-calculus to be a good trade-off.

3.4.3.2 Syntax and Semantics

The syntax and semantics of the μ-calculus have been discussed in detail in Section 2.3.3. Figure 2.7 provided a summary of this. However, it is important to note that a slight modification has been introduced to the original language. The set of actions 'A' does not represent message input and output actions only, but non-message passing actions (N-MPA) as well, which are essentially agents' internal computations. These are specified by '$\#(C)$', where 'C' is a constraint to be satisfied by the agent. This is important for verifying interaction models written in LCC, which considers such constraints as actions to be performed by the agents.

The following section provides an example of a possible temporal property that may be verified for the travel agency scenario.

3.4.3.3 Example: the travel agency scenario

It is important to verify whether interaction models are reliable or not. For example, in the travel agency scenario, one important property to verify is whether the customer will always receive a receipt after it makes its payment. This is expressed by Property 3.1.

$$\begin{aligned} &\nu X.\ [-]X \wedge [out(payment(\_), a(customer(\_), \_))] \\ &\quad (\ \mu Y.(\langle - \rangle \mathtt{tt} \wedge [-]Y) \vee \langle in(confirm\_booking(\_, \_), a(customer(\_), \_)) \rangle \mathtt{tt}) \end{aligned} \tag{3.1}$$

Property 3.1 states that every time a payment is sent by the customer ($[out(payment(\_), a(customer(\_), \_))]$) the following sub-formula should hold: $\mu Y.(\langle - \rangle \mathtt{tt} \wedge [-]Y) \vee \langle in(confirm\_booking(\_, \_), a(customer(\_), \_)) \rangle \mathtt{tt}$. This sub-formula states that either a booking confirmation is received ($\langle in(confirm\_booking(\_, \_), a(customer(\_), \_)) \rangle \mathtt{tt}$), or something can happen ($\langle - \rangle \mathtt{tt}$) and the sub-formula should hold again for everything that happens ($[-]Y$). The use of the least fixpoint operator (μY) guarantees termination, which implies that the booking confirmation will eventually be received. Furthermore, Property 3.1 is always satisfied in the state-space because for every action that can happen in the interaction the property is said to be satisfied again at the next state ($[-]X$), and this may continue infinitely often (νX).

While Property 3.1 fails for the interaction model of Figure 3.8, Property 3.2 succeeds. Property 3.2 states that if the customer agent sends its payment, then it will either receive a receipt or a reason for payment failure.

$$\begin{aligned} &\nu X.\ [-]X \wedge [out(payment(\_), a(customer(\_), \_))] \\ &\quad (\ \mu Y.(\langle - \rangle \mathtt{tt} \wedge [-]Y) \vee \langle [in(confirm\_booking(\_, \_), a(customer(\_), \_)), \\ &\qquad\qquad in(fail\_payment(\_, \_), a(customer(\_), \_))] \rangle \mathtt{tt}) \end{aligned} \tag{3.2}$$

Note that the travel agency scenario presented in this chapter has been simplified for the purpose of maintaining clarity in our explanation. As a result, the success and failure of the above properties might have been obvious to the reader. However, realistic scenarios are usually a bit more complex. For instance, Appendix A.1.1 provides us with a more realistic scenario with a richer choice of actions for the agents to select from. Taking a look at that scenario affirms that the satisfaction of properties, such as Property 3.1 or 3.2, might not really be very obvious to the agents as well as humans.

3.5 The MCID Model Checker

Sections 3.1 and 3.3 have introduced our system model and our choice of verifier: the MCID model checker. Section 3.4 introduced the specification languages used by the MCID model checker. In this section, we present the technical details and the formal semantics of the MCID system.

3.5.1 MCID's Technique: Local Model Checking

Examples of the interaction protocol, deontic constraints, and temporal properties fed to the model checker for verifying the travel agency scenario have been presented in

Sections 3.4.1.3, 3.4.2.3, and 3.4.3.3. Figure 3.11 illustrates this by providing a fraction of the real input. Note that the LCC interaction model is essentially a specification of a finite state-graph.

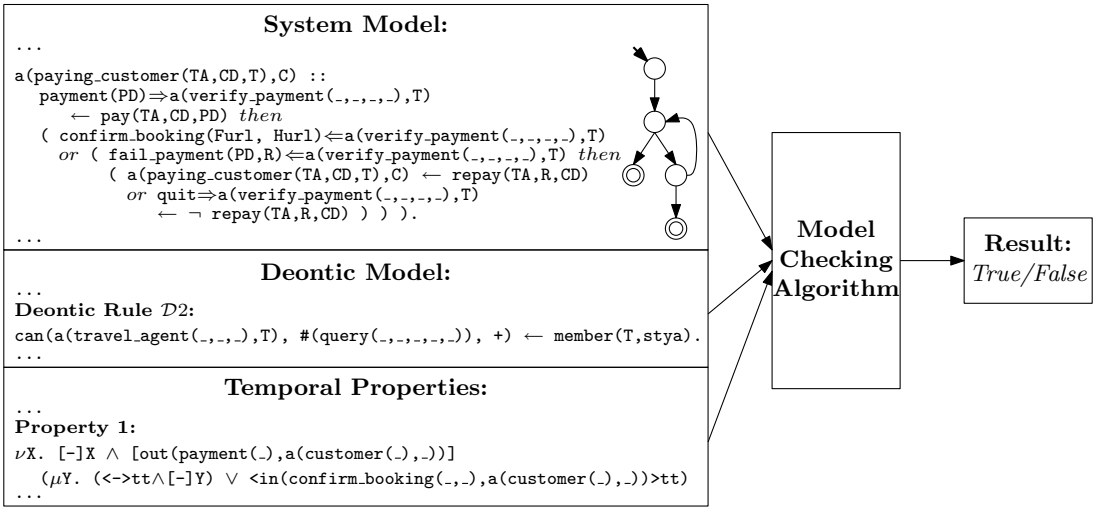


Figure 3.11: Travel agency scenario: MCID's sample input

Two model checking techniques exist: global and local model checking. In global model checking, the entire state-space is generated before satisfaction is verified. Only after all the states have been enumerated, the model checker can start searching for those states satisfying a given property. On the other hand, local model checking techniques tend to verify whether a single state, the initial state s_0, satisfies the property in question. The state-space is then constructed and traversed at run time as needed. The system model of Figure 3.11 provides a sample state-graph defining the actions of the paying customer agent $a(paying\_customer(\_,\_,\_),C)$. Figure 3.12 presents a clearer illustration of this state-graph. Note that this is only a section of the travel agency scenario's state-graph, and that the entire state-graph of the system is required for verification. The ultimate goal of the model checker is to verify whether a temporal property is satisfied in a given interaction. The state-graph is unfolded into a state-space and traversed one step at a time until a solution is reached. While the state-graph has to be finite, the resulting state-space may be infinite, as depicted by the right hand graph of Figure 3.12.

Local model checking, along with LCC's logic-based clause expansion mechanism for constructing and traversing a state-space, and the μ-calculus use of recursion for specifying temporal properties, suggest the use of a logic based model checker. Model checking approaches based on tableaux systems, such as that of Stirling and Walker

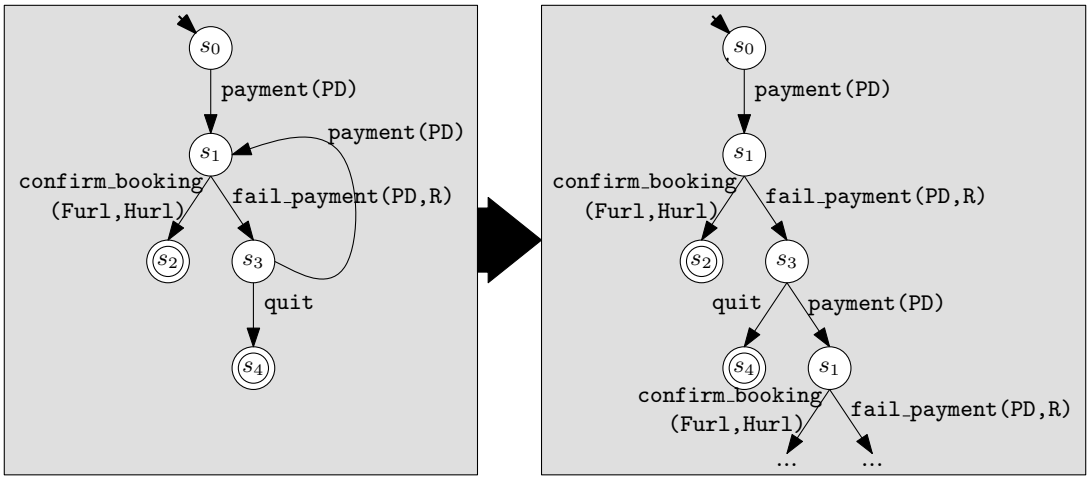


Figure 3.12: Constructing and traversing a state-space

(1991), provide one solution. However, termination in these verification techniques is a crucial issue. We refer to the XMC model checker for inspiration. The XMC system (Ramakrishnan et al., 2000) is a model checker built on top of XSB tabled Prolog (Sagonas et al., 1994). The concept of using tables for caching results in XSB ensures termination, avoids redundant sub-computations, and computes the well-founded model for normal logic programs (Ramakrishna et al., 1997). We rationally reconstruct the XMC model checker to suit our requirements. Sections 3.5.2 and 3.5.3 present MCID's algorithm, framework, and formal semantics. Section 3.5.4 discusses the differences between the XMC and MCID systems.

3.5.2 MCID's Algorithm

The model checker's algorithm is defined by Figure 3.13. Verification starts when the set of temporal properties P, the set of deontic constraints D, and the interaction's state-space — defined via the tuple $\langle N, T \rangle$, where N is the set of nodes/states and T is a relation defining the transitions between these nodes/states — is available. The set of states to be verified, S, is initially set to the empty set $\emptyset$; note that the $\overset{\text{is}}{=}$ operator assigns the variable on the left-hand side the value of the set on the right-hand side. Each deontic constraint d is then translated into a temporal constraint t and appended to the set of temporal properties P. The initial state s_0 is retrieved from I and appended to the set S. Now if the set P is empty, then this implies that verification has completed successfully. Otherwise, a property ϕ is retrieved from P to be verified against the initial state s_0. If ϕ is satisfied at state s_0 ($s_0 \models \phi$), then the property is said to be

satisfied in the interaction I, and S is set to $\emptyset$ in order to repeat the verification process all over again for the remaining properties of P. However, if ϕ fails to be satisfied at s_0, then a transition is made to the next state(s). This new state(s) is added to S so that the property ϕ will be verified against it in the next verification cycle. Only if the property has been checked and failed against all possible states, then the property is said to be unsatisfied in the interaction.

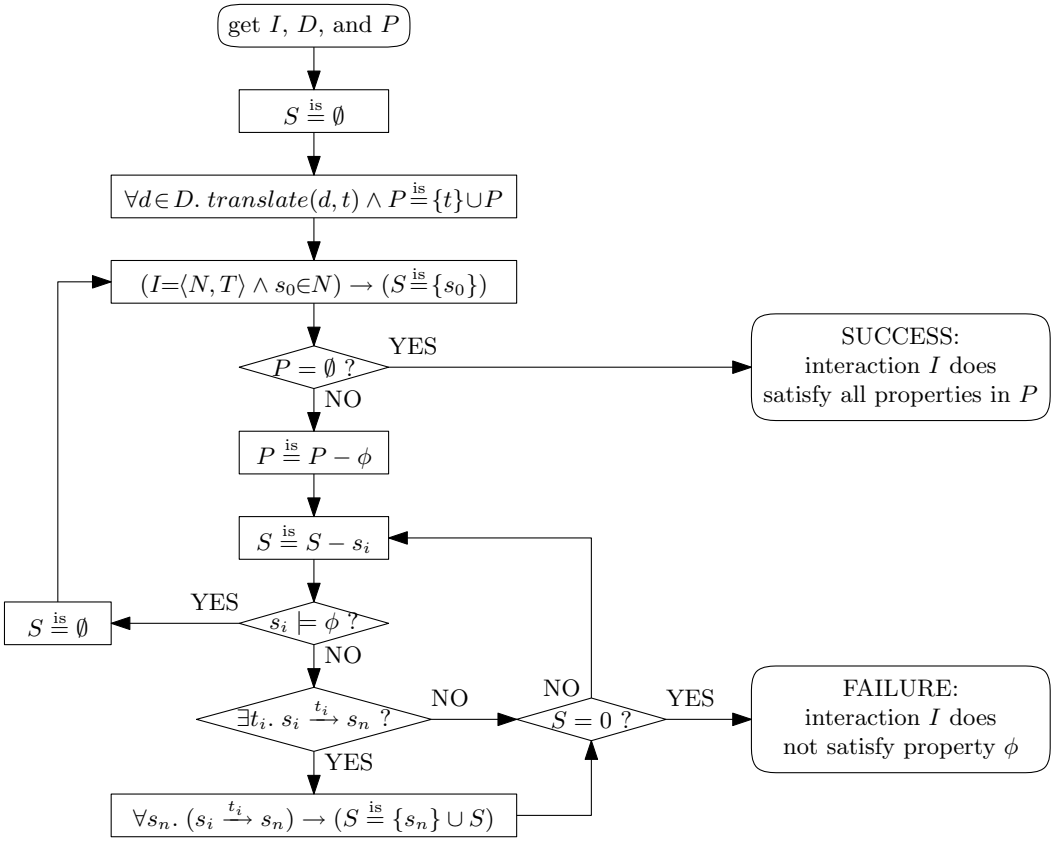


Figure 3.13: MCID's algorithm

We note that the current MCID system, which has been designed for run time verification of multiagent systems, does not return a counter example if a property fails to be satisfied. Counter examples returned by model checkers are usually useful for designers in order to pinpoint the errors in their design, so that these errors could be fixed. However, the MCID system was designed to be invoked by agents at run time in order to help agents select appropriate interaction models. The agent is usually interested in whether a given interaction will be chosen or not. A counter example is therefore not essential.

3.5.3 MCID's Framework and Formal Semantics

As the previous section illustrates, the model checker attempts to satisfy a property at the initial state of the state-space. If the property is not satisfied, then a transition to the next state(s) is made, at which the property is checked again, and so on. If we take a look at the algorithm of Figure 3.13, we notice that there are three main operations to be performed by the MCID system. The first is translating deontic constraints into temporal ones. This has to be performed only once at the beginning of the verification process. Its details are discussed in Section 3.5.3.1. The other two operations are proving *satisfaction* at a given state and performing a *transition* from one state to the next. These are discussed in Sections 3.5.3.2 and 3.5.3.3, respectively. Figure 3.14 presents MCID's framework and its three main components.

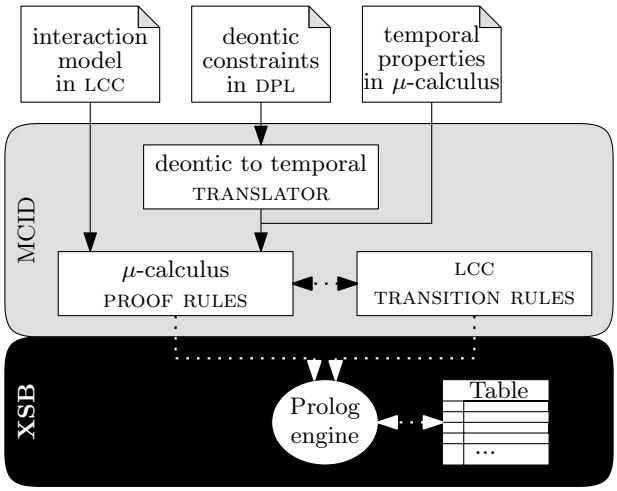


Figure 3.14: MCID's framework

Note that the interaction's state-graph should always be a finite graph. However, these graphs may be unfolded into an infinite state-space. To deal with non-termination, as in the XMC model checker, we build our system on top of XSB tabled Prolog. The predicate responsible for verifying the satisfaction of a given property at a given state (the '$satisfies$' predicate of Figure 3.16), is defined as a tabled predicate. This means that results of calls to this predicate are cached in a table in the XSB system. Every time a call is made to this predicate, the table is searched for results. Only if this is a new call — a call with a new combination of a state and a temporal property — will it be resolved against the predicate definition. Caching results implies that each system state will be visited only once when evaluating a temporal property. This implies that the model checker will always terminate with an answer in a finite state-graph. Note

that the XSB system is dealt with as a black box. We do not need to know the details of the XSB system, but we do need a basic understanding of the concept of tables in XSB.

3.5.3.1 Deontic to Temporal Translator

Deontic constraints are specified via the DPL language of Figure 3.10. However, to verify whether these constraints will be broken or not, the verifier needs to study the occurrence of the actions constrained deontically in the given state-space. Since we are using model checking as our method of verification, then the occurrence of these actions needs to be specified in a temporal logic. Figure 3.9 has already provided a mapping between deontic sets and the occurrence of the deontically constrained action in an interaction's state-space. This mapping needs to be associated with the general temporal operators `A`, `G`, `E`, and `F`, defined in Section 2.3.2. The final result is presented in Figure 3.15.

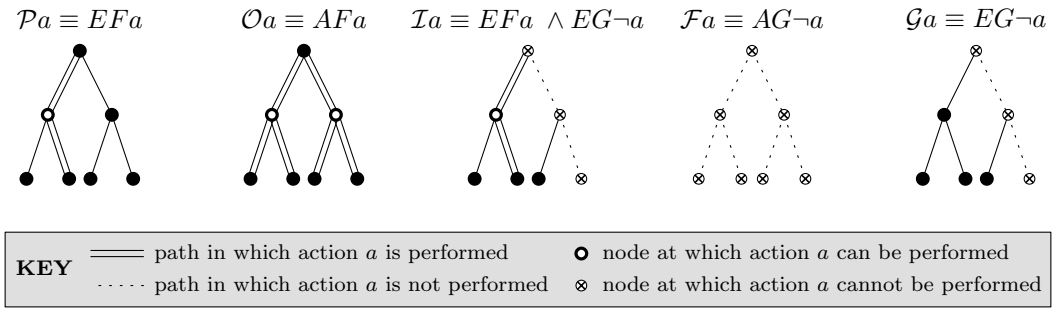


Figure 3.15: Mapping deontic operators into temporal ones

Mapping DPL predicates into MCID's temporal language, the μ-calculus, is a straightforward task. It is based on translating DPL predicates into general deontic logic representation (as presented by the language's semantics in Section 3.4.2.2), mapping the deontic logic representation into a temporal one (as presented by Figure 3.15), and specifying the temporal logic in the μ-calculus (following the μ-calculus syntax and semantics presented in Section 2.7). Table 3.2 provides the result of this mapping.

Permission in the μ-calculus is specified as follows: $\mu X.\langle Action'\rangle \mathtt{tt} \vee \langle - \rangle X)$, which states that either the action can occur ($\langle Action'\rangle \mathtt{tt}$) or something can happen after which the property should be satisfied again ($\langle - \rangle X$). Finally, termination is guaranteed by the least fixpoint operator (μX). The guaranteed termination implies that the action $Action'$ will eventually occur in some path.

Gratuitousness in the μ-calculus is specified as follows: $\nu X.\langle -Action'\rangle X \vee [-]\mathtt{ff}$, which states that either the action in question cannot happen in at least one of the next

#	**DPL Predicate**	$\mu-$**Calculus Equivalence**
1.	$can(Agent, Action, +) \leftarrow Condition.$	$(satisfied(Condition)$ $\wedge\ \mu X.\langle Action'\rangle \mathtt{tt} \vee \langle - \rangle X) \vee$ $(\neg satisfied(Condition)$ $\wedge\ \nu X.[Action']\mathtt{ff} \wedge [-]X)$
2.	$can(Agent, Action, -) \leftarrow Condition.$	$(satisfied(Condition)$ $\wedge\ \nu X.\langle -Action'\rangle X \vee [-]\mathtt{ff}) \vee$ $(\neg satisfied(Condition)$ $\wedge\ \mu X.[-Action']X \wedge \langle - \rangle \mathtt{tt})$
3.	$must(Agent, Action, +) \leftarrow Condition.$	$(satisfied(Condition)$ $\wedge\ \mu X.[-Action']X \wedge \langle - \rangle \mathtt{tt}) \vee$ $(\neg satisfied(Condition)$ $\wedge\ \nu X.\langle -Action'\rangle X \vee [-]\mathtt{ff})$
4.	$must(Agent, Action, -) \leftarrow Condition.$	$(satisfied(Condition)$ $\wedge\ \nu X.[Action']\mathtt{ff} \wedge [-]X) \vee$ $(\neg satisfied(Condition)$ $\wedge\ \mu X.\langle Action'\rangle \mathtt{tt} \vee \langle - \rangle X)$

Table 3.2: Mapping DPL predicates into μ-calculus formulae

steps, after which the property should be satisfied again ($\langle -Action' \rangle X$), or nothing can happen anymore ($[-]\mathtt{ff}$). This property is satisfied infinitely often (νX). In summary, it states that there exists a path in which the action $Action'$ never happens.

Obligation in the μ-calculus is specified as follows: $\mu X.[-Action']X \wedge \langle - \rangle \mathtt{tt}$, which states that something can happen ($\langle - \rangle \mathtt{tt}$), and for all actions that occur that are different from $Action'$, the same property should be satisfied again ($[-Action']X$). Finally, μX guarantees termination, implying that the action $Action'$ will eventually occur in all paths.

Forbiddance in the μ-calculus is specified as follows: $\nu X.[Action']\mathtt{ff} \wedge [-]X$, which states that action $Action'$ is not permitted ($[Action']\mathtt{ff}$), and for every other action that occurs, the same property will be satisfied again ($[-]X$) infinitely often (νX).

Furthermore, as illustrated earlier by Section 3.4.2.2 and presented in our translation in Table 3.2, if the condition of the deontic rule is satisfied, then '*can*' predicates with a positive sign are treated as permissions, '*can*' predicates with a negative sign as gratuitousness, '*must*' predicates with a positive sign as obligations, and '*must*' predicates with a negative sign as forbiddance.

But what if the condition of a DPL deontic rule is not satisfied? In this case, different

interpretations may be accepted. For example, if the condition of a permission rule is not satisfied, do we consider this enough proof for forbidding the action, or do we require additional rules that explicitly forbid the action before doing so? In our current implementation, as illustrated by Table 3.2, every time a condition is not satisfied the negation of the μ-calculus specification should be satisfied. For example, permissible actions whose conditions have not been satisfied are treated as forbidden actions, and vice versa. Similarly, obligatory whose conditions have not been satisfied are treated as gratuitous actions, and vice versa.

3.5.3.2 μ-Calculus Proof Rules

The '*proof rules*' component of Figure 3.14 takes in an interaction model and a set of temporal properties. It then proves the satisfaction of these properties in the given interaction model. The properties are specified via the μ-calculus temporal language; therefore, their satisfaction is proved following the μ-calculus proof rules of Figure 3.16. The rules are based on the semantics of the μ-calculus language, which were presented earlier in Figure 2.7. They imply that a state E always satisfies `tt` (*true*), and never `ff` ($false$). E satisfies $\phi_1 \vee \phi_2$ if it satisfies either ϕ_1 or ϕ_2, and it satisfies $\phi_1 \wedge \phi_2$ if it satisfies both ϕ_1 and ϕ_2. $\langle A \rangle \phi$ is satisfied if state E can make at least one transition A to state F, such that F satisfies ϕ. $[A]\phi$ is satisfied if for all transitions A that state E can take to F, then F satisfies ϕ. Prolog, by nature, computes the least fixed point solution. Hence, $\mu Z.\phi$ is satisfied if state E satisfies the property ϕ. The greatest fixed point, however, is the dual of the least fixed point. Therefore, the greatest fixed point formula is satisfied if the least fixed point of the negated formula fails to be satisfied. The translation of these rules into XSB Prolog is then a straightforward mechanism. The result is a compact and efficient model checker.

$$
\begin{array}{lcl}
satisfies(E, \texttt{tt}) & \leftarrow & true \\
satisfies(E, \phi_1 \vee \phi_2) & \leftarrow & satisfies(E, \phi_1) \vee satisfies(E, \phi_2) \\
satisfies(E, \phi_1 \wedge \phi_2) & \leftarrow & satisfies(E, \phi_1) \wedge satisfies(E, \phi_2) \\
satisfies(E, \langle A \rangle \phi) & \leftarrow & \exists F.\ (trans(E, A, F) \wedge satisfies(F, \phi)) \\
satisfies(E, [A]\phi) & \leftarrow & \forall F.\ (trans(E, A, F) \rightarrow satisfies(F, \phi)) \\
satisfies(E, \mu Z.\phi) & \leftarrow & satisfies(E, \phi) \\
satisfies(E, \nu Z.\phi) & \leftarrow & dual(\phi, \phi') \wedge \neg satisfies(E, \phi')
\end{array}
$$

Figure 3.16: μ-calculus proof rules

3.5.3.3 LCC Transitions Rules

Proving satisfaction may sometime require transitions to be made from one state to the next (note the occurrence of $trans(E, A, F)$ in Figure 3.16). In such cases, and since the the state-space is specified via the LCC language, the '*transition rules*' component of Figure 3.14 performs these transitions based on the LCC transition rules of Figure 3.17. These rules are based on LCC's expansion mechanism, which was presented earlier in Figure 2.5. They imply that $E :: D$ can perform a transition A to F if D can perform a transition A to F. $E_1\ or\ E_2$ can perform a transition A to F if either E_1 or E_2 can perform a transition A to F. $E_1\ then\ E_2$ can perform a transition A to E_2 if E_1 can perform a transition A to the empty process *nil*. Otherwise, it can perform a transition A to $F\ then\ E_2$ if E_1 can perform a transition A to F. Note that while *null* is LCC's empty action, *nil* is used to describe an empty process. An empty action may be succeeded by other actions; however, an empty process marks the end of the process, implying that no further actions may be executed. $E_1\ par\ E_2$ can perform a transition A to either $F\ par\ E_2$ if E_1 can perform a transition A to F, or to $E_1\ par\ F$ if E_2 can perform a transition A to F. $M \Leftarrow P$ can perform a transition $in(M)$ to the empty process *nil* by retrieving the incoming message M. $M \Rightarrow P$ can perform a transition $out(M)$ to *nil* by sending the message M. Finally, $E \leftarrow C$ may perform some internal computation $\#(X)$ if X is a term in the conjunction of terms C ($XinC$) and the constraint C is satisfied ($sat(C)$).

$$
\begin{array}{lcl}
trans(E :: D, A, F) & \leftarrow & trans(D, A, F) \\
trans(E_1\ or\ E_2, A, F) & \leftarrow & trans(E_1, A, F) \vee trans(E_2, A, F) \\
trans(E_1\ then\ E_2, A, E_2) & \leftarrow & trans(E_1, A, nil) \\
trans(E_1\ then\ E_2, A, F\ then\ E_2) & \leftarrow & trans(E_1, A, F) \wedge F \neq nil \\
trans(E_1\ par\ E_2, A, F\ par\ E_2) & \leftarrow & trans(E_1, A, F) \\
trans(E_1\ par\ E_2, A, E_1\ par\ F) & \leftarrow & trans(E_2, A, F) \\
trans(M \Leftarrow P, in(M), null) & \leftarrow & true \\
trans(M \Rightarrow P, out(M), null) & \leftarrow & true \\
trans(E \leftarrow C, \#(X), E) & \leftarrow & X\ in\ C \wedge sat(C)
\end{array}
$$

Figure 3.17: LCC transition rules

3.5.4 MCID versus XMC

MCID was originally based on the XMC model checker. XMC was first selected due to its appealing logic-based nature (see Section 3.5.1). It was then stripped down to its basic operations, which were later modified to suit our needs, resulting in the MCID system.

Currently, MCID, like XMC, is built on top of the XSB tabled Prolog system, which addresses the critical issue of termination. Unlike XMC, the transition rules of MCID are based on the LCC transition rules. However, similar to XMC, the core of the MCID system is based on the μ-calculus proof rules. It is worth noting that MCID, unlike XMC, does not use the traditional notion of channels. Process calculi usually make use of channels, through which messages are passed, to connect processes together. Instead of using channels, LCC directly specifies the agent to which the message is sent to. MCID implements this by introducing a list of messages M_i. The sending and receiving agent details are attached to the transmitted messages which are saved in M_i. When an agent is expecting a message, it searches for it in the list of incoming messages M_i. To keep things simple and clear, we ignore these details in Figure 3.16.

However, the major difference between the MCID and XMC systems is the introduction of a deontic layer to the MCID system model. This adds a whole new component to the verifier for dealing with the deontic constraints (Section 3.5.3.1).

3.6 Results

To verify the travel agency scenario presented in this chapter, the MCID model checker is fed the LCC interaction model of Figure 3.8, the DPL deontic constraints of Table 3.1, and temporal properties 3.1 and 3.2. However, as illustrated in Section 3.5.3.1, all deontic constraints are first translated into μ-calculus properties before verification starts.

We note that verification is expected to be carried out by the agents, such as a broker agent, at run time. In which case, the exact number of agents playing each role would be known. In our case, the system is verified for one customer agent, one travel agent, two airline web services, one hotel directory service, and one credit card web service. The CPU time and memory usage, from the moment the model checker is invoked until the results are returned to the user, are presented by Table 3.3. We note that all the results presented in this thesis have been obtained when running the model checker on an Intel Core 2, 2GHz machine.

It is not only our proposed model checker's algorithm that distinguishes our verifi-

Property #	CPU time (in *sec*)	Memory usage (in MB)
Deontic Rule $\mathcal{D}1$	0.036	2.342
Deontic Rule $\mathcal{D}1$	0.048	2.342
Deontic Rule $\mathcal{D}1$	0.004	0.137
Deontic Rule $\mathcal{D}1$	0.044	1.999
Temporal Property 3.1	0.064	2.518
Temporal Property 3.2	0.064	2.772

Table 3.3: Travel agency scenario: verification results

cation mechanism from others, but the entire system model and property specification. The combination of interaction and deontic models, one of the main novel contributions of this thesis, introduces a major distinction between the system models verified in this thesis and those verified by other model checkers. For this reason, we find it hard to compare our mechanism to others by simply checking numerical results. Nevertheless, we believe these numerical results are sufficient to prove that interaction time verification could be indeed possible. As illustrated by Table 3.3, on average, the CPU time consumed is a fraction of a second and the memory usage is less than 3MB. We believe that having agents wait a fraction of a second before deciding whether or not to join a given interaction to be a realistic and reasonable waiting time in general. Furthermore, most machines nowadays come with a decent memory storage; hence, a few megabytes is usually not a problem.

Therefore, we believe that the results of Table 3.3 highlight the efficiency of the MCID verifier. This efficiency is the result of many factors. First, verifying an explicitly defined interaction model considerably limits the interaction's state-space. Second, using a logic based model checker contributes to the efficiency in computing complex data structures. Third, the model checker makes use of the underlying XSB system for searching the state-space. Tabling mechanisms in XSB, which caches results in tables, ensures termination and avoids redundant sub-computations. In conclusion, the result is a lightweight and efficient model checker that promotes interaction time verification.

3.7 Conclusion

This chapter has presented a verification technique for predicting and preventing failure — due to errors within the interaction model, conflicts between the interaction

protocol and the agents' rules and requirements, as well as clash of interests between the agents — as early as possible in multiagent system scenarios. This is made possible by allowing agents to invoke the model checker at run time for verifying instances of the interaction protocol.

Having the LCC specification language as the executable language as well supports our dynamic model checking requirement. This is made possible due to the clause expansion mechanism of LCC (Figure 2.5), which allows agents to automatically retrieve the current protocol state and feed it to the model checker. Furthermore, the combination of using the μ-calculus as the model checker's temporal logic, along with the logic-based nature of the verifier, results in an extremely compact size model checker. The entire model checker is built in less than 200 lines of Prolog code[3]. This makes MCID ideal to be used by agents at run time. The system, however, is built on XSB tabled Prolog. The actual burden of searching the state-space is, therefore, thrown on the underlying XSB system. Last, but not least, the logic-based nature of MCID provides efficiency in computing constraints and complex data structures. This is crucial for our system model, which makes heavy use of constraints and structured terms.

As for the DPL language presented in this chapter, the language has been kept relatively simple. To some extent, DPL may be viewed as similar to common policy languages such as Rei (Kagal et al., 2003), ASL (Jajodia et al., 1997), Ponder (Dulay et al., 2002), etc. For instance, DPL allows the specification of conditions for specifying when agents can/cannot perform a given action. It also allows the specification of obligations, through the use of positive '*must*' predicates. However, the expressiveness of the language is increased by allowing more complex conditions. Conditions can make use of the underlying μ-calculus temporal logic and the agent's constraint language, which is Prolog in our case. For example, using temporal constraints, one may easily specify that action *A* can be performed only if action *B* has already occurred. Another interesting aspect of DPL is that it has been specified for the purpose of verifying deontic constraints. Therefore, unlike other policy languages, the semantics of DPL predicates are better understood with respect to the temporal occurrence of permitted, prohibited, or obligatory actions in the interaction's state-space. As a result, a direct mapping between DPL predicates and the μ-calculus temporal logic exists. This is essential for keeping the model checker's algorithm basic, efficient, and lightweight.

[3]The model checker's code is available in Appendix B.

3.8 Future Work

It is worth noting that the MCID system may be used in three different ways:

- ✦ To find a suitable set of **collaborating agents** for a given interaction model. This is useful for addressing collaboration and coalition formation problems.
- ✦ To prove the **satisfaction** of certain properties in a given system. For example, verifying safety and liveness properties in addition to more interesting ones, such as properties of trustworthiness.
- ✦ To aid agents in their **decision process** by verifying which actions will result in the (dis)satisfaction of certain properties. This could be helpful in dialogue games, negotiation protocols, argumentation, etc.

These three problems are intertwined. This thesis focuses on the first two. For instance, the temporal properties of Section 3.4.3.3 address the second problem, while deontic constraints are used to address the first. The third problem is closely connected to the second as well, yet it addresses a completely different issue from that of finding collaborating agents. We therefore leave the third issue — using the model checker for aiding the agents' decision process — for future work. Considering probabilistic and stochastic analysis might be useful in tackling these issues. However, we note that the introduction of such analysis would require significant modifications to the current model checker.

As an application, we have chosen to focus in this thesis on the issue of trust. This choice is due to the significance of this problem in distributed open systems, especially that despite the extensive research in the field of trust, no verification mechanism has yet addressed this issue. The next chapter illustrates how the MCID model checker may be used to verify trust in multiagent systems by verifying the agents' trust constraints along with properties of trustworthiness of the interaction model.

Chapter 4

Verification of Trust in Multiagent Systems

Chapter 3 has introduced an efficient model checking system which has the potential of addressing a variety of problems, such as proving the correctness of interaction models, finding suitable collaborating agents, aiding the agent's decision process in dialogue games and others, etc. However, due to the criticality of the trust issue in distributed open systems and the urgent need for practical trust solutions, we choose the trust domain to be our model checker's application domain. In this chapter, we show how the MCID system may be used by agents at run time to verify whether a certain interaction is trustworthy and whether the other agents it will be interacting with are to be trusted in performing their assigned roles.

This chapter opens with an introduction to the issue of trust in Section 4.1. This is followed by an overview of available trust models and mechanisms for multiagent systems in Section 4.2. Our proposed trust model is presented in Section 4.3, followed by a motivating example in Section 4.4. Our proposed trust policy language for the specification of trust is introduced in Section 4.5. We then revisit our verification mechanism in Section 4.6, before drawing our conclusions in Section 4.8.

4.1 The Issue of Trust

Trust remains a fundamental challenge for the success of distributed open systems. Despite much research, the notion of trust remains vague and, as yet, there is no consensus on what exactly trust is. This is because trust may be addressed at different levels. At the low system level, trust is associated with network security, such as:

- ✦ authentication, for determining the identity of the user entity,
- ✦ access permissions, for deciding who has access to what,
- ✦ content integrity, for determining whether the content has been modified,
- ✦ content privacy, for ensuring only authorised entities can access the content, etc.

At higher levels, the focus is on trusting entities — these could be human users, software agents, services, directories, etc. — to:

- ✦ perform actions as requested,
- ✦ provide correct information,
- ✦ not to misuse information,
- ✦ execute protocols correctly, etc.

As a result, what is considered to be a trust issue for someone may mean something completely different for another. For some people, trust may be synonymous with network security, popularity, reputations, etc. We prefer to inherit a more general definition of trust. We define the trust issue as the problem of who to interact with, when to interact with them, and how to interact with them (Ramchurn et al., 2004). We call this the *who/when/how* questions of trust.

The complexity of the issue of trust, as we view it, lies neither in the broad spectrum of these trust issues nor in the diversity of methods addressing them. The problem we try to tackle lies in the inability of being able to decide at run-time which issues should currently be addressed and how to address them. We argue that different environments and different scenarios require different strategies for dealing with trust. For example, under critical circumstances, it may be preferred to distrust new entrants to the system, i.e. agents with no ratings. However, under less critical circumstances, it may be favoured, and possibly necessary, to trust any agent with no explicit negative reputation. Therefore, we leave it to the agent to decide which strategy is most suitable for a given scenario. Such a decision is usually made for a specific interaction model with a predefined set of collaborating agents. As a result, we propose a contextualised trust model that is built on top of existing trust mechanisms, allowing the specification of more dynamic mechanisms.

Available research has mainly focused on analysing, refining, and developing strategies that address various trust issues at different system levels. We believe all these

strategies are important. However, it is the agent's decision process that decides which strategy is appropriate for a given scenario. The agent's reasoning layer that selects this best strategy is outside the scope of this thesis. Our work concentrates on providing this layer with the formal methods for specifying and verifying these strategies, which in effect does aid its decision process. We show how the specification and verification methods of Chapter 3 may be applied to this problem.

Before presenting our contextualised trust model and its specification and verification mechanisms, the following section provides an overview of the available trust models and mechanisms of the literature.

4.2 Trust in Multiagent Systems: an Overview

Much research has been undertaken in the field of trust in multiagent systems aiming at answering the *who/when/how* questions of trust. Ramchurn et al. (2004) divides this work into two categories. The first deals with trust at the individual level and the second at the system level. Figure 4.1 presents the different trust models and mechanisms at both levels. In what follows, we give a brief overview of these models and mechanisms.

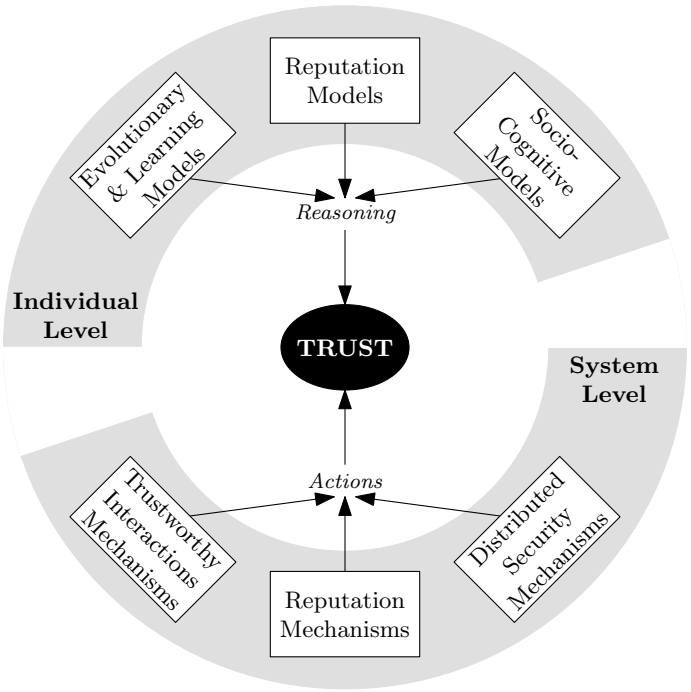


Figure 4.1: Trust models and mechanisms (Ramchurn et al., 2004)

- **Individual Level Trust:** Individual level trust focuses on providing agents with models that would allow them to compute their trust in others. As a result, the burden of computation of such models lies on the agent.

  - **Evolutionary & Learning Models.** These models rely on the fact that agents can interact with each other over a number of encounters, leading them to learn about each others behaviour with time and improve their expectations accordingly. Similar to game theory, agents may analyse their opponents' previous moves and adapt their strategies in retrospect. This could result in evolving strategies promoting emerging trust, such as the Tit for Tat strategy[1].

  - **Reputation Models.** In open environments, it is hard for agents to have interacted sufficiently with the majority of agents in the system. Therefore, instead of relying on personal previous experience, agents may rely on other agents' experiences and their impressions of each other. In order to achieve this, each agent should be capable of constructing a social network to be used for retrieving reputations. For aggregating these ratings, agents should pick strategies that would address questions of the form: What if agents lie when rating others in order to preserve a certain coalition? What if the new entrant with no ratings is actually an old one with a bad reputation who has re-entered the system with a new identity? Should all new entrants be penalised? etc.

  - **Socio-Cognitive Models.** Evolutionary and learning models as well as reputation models are based on agents relying on past experiences and their outcome for computing their trust in each other. However, relying on past experience need not be the only solution. For example, agents may analyse and compute their trust in others based on their belief in the other's capability, willingness, persistence, and motivation. For example, knowing that agent X is capable of selling wine, one may deduce that X is most probably also capable of selling beer. Another example states that one may believe that agent Y will carry out a deal simply because it knows that completing the deal successfully will provide Y with an unmissable reward.

[1] The Tit for Tat strategy is based on punishment and reward. If the agent defects, it will be punished with another defection. If it cooperates, it will be rewarded with another cooperative action. Over a period of encounters, this will promote emerging trust and both agents will be expected to be cooperating.

- **System Level Trust:** System level trust aims at providing the system with the appropriate mechanisms for preserving a certain level of trustworthiness by driving agents to perform correctly. As a result, the burden of computation of such mechanisms lies on the system, as opposed to the agent.

  - **Trustworthy Interaction Mechanisms.** Protocols may be used to guide or dictate the individual steps of an interaction. These protocols could aim at encouraging agents to be truthful. For example, while an English auction encourages truth telling on the auctioneer, a Vickrey auction encourages truth telling on the bidders[2]. Of course, these mechanisms assume agents are abiding to the protocol, which is not always the case. For addressing the lying agents problem and other collusion issues (such as having bidders cheat by collaborating and sharing information to beat the auctioneer), additional security mechanisms may be enforced, possibly making use of cryptographic techniques and other technologies.

  - **Reputation Mechanisms.** Sometimes agents do not bother rate each other if they do not directly benefit from that. Modelling the reputation mechanism at the system level can help achieve an 'incentive compatible system', for example, by introducing side payments. The reputation model at the individual level has also raised several issues, many of which are best dealt with at the system level. For example, to discourage agents from cheating and re-entering the system with a new identity, the system can make it costly for agents to change identities.

  - **Distributed Security Mechanisms.** The system should use mechanisms for addressing issues in the domain of network security, such as identity proof, access permission, content integrity, content privacy, etc. Certificates may be used to prove an agent is reciprocative, reliable, and honest. Organisational approaches of multiagent systems provide means of specifying which agent can perform what actions based on the roles they play.

[2]In an English auction, all bids are made public, preventing the auctioneer from lying about who won the item and at what price. In a Vickrey auction, bids are sealed, providing the opportunity for the auctioneer to lie about who is the highest bidder and/or what is the highest bid. However, the winner of a Vickrey auction only pays the amount of the second highest bid. This implies that if the bidder bids a value lower than its true valuation, then the bidder will either risk losing the item altogether or win it for the same price as bidding its true valuation. This essentially strips the bidder from any incentive for lying about its true valuation.

4.3 A Contextualised Trust Model

After introducing the critical issue of trust and presenting an overview of the available trust mechanisms, this section proposes a contextualised trust model that is built on top of these existing models and mechanisms. The problem, as presented in Section 4.1, is that while one trust mechanism may be preferred for one scenario, it could be inappropriate or even useless for another. For example, buying a £12.00 music CD online requires a completely different approach for dealing with trust from buying a £600.00 Gibson electric guitar. In the first case, the buyer might be satisfied with checking the general ratings of the seller. In the latter case, even if the seller has very high ratings, the buyer might still need to inspect many details, answering questions such as: Are the ratings written by friends of the seller? Are the ratings based on similar transactions? Is the seller an authorised dealer? Does the system protect the buyer from fraud? etc. Clearly, different approaches and different mechanisms are needed for dealing with different scenarios. This, we believe, raises a need for a contextualised trust model, which allows the agent to decide whether the interaction it is about to join and the agents it will be collaborating with are sufficiently trustworthy. Each agent will then have its own set of trust rules that restrict when, how, and who to interact with. Of course, the system may still provide services such as issuing certificates, storing reputations, etc. However, these services would be distributed with no centralised control behind them. It is then up to the agent to decide when and how to use such services.

This view of a contextualised trust model is in line with the proposed multiagent system model of Section 3.1. We view a multiagent system as a collection of interaction models and a group of autonomous agents. These autonomous agents are responsible for grouping themselves into the various interactions. There should be no higher coordination layer, such as a centralised control centre, that would affect or control such groups. Each agent will have its own set of rules that it would consult when deciding whether the interaction it is going to join and the agents it will be collaborating with are trustworthy or not.

Earlier trust models and mechanisms have focused either on models that allow agents to compute their trust in others (i.e. individual level trust models) or on mechanisms that allow the system to enforce rules resulting in a trustworthy environment (i.e. system level trust mechanisms). Our approach is slightly different since it focuses solely on what the agent's trust requirements and constraints are. Nevertheless, these could impose constraints on both the interaction model and other agents in the system.

Figure 4.2 provides examples on the specification of trust mechanisms in our proposed contextualised model. The result is two classes of constraints: constraints on the agents and constraints on the interaction model. For example, if there is a requirement that some agent should be enforced to prove its identity (the *Distributed Security Mechanism* of Figure 4.2), then this constrains the interaction by expecting it to allow the agent to send its PGP *before* performing any crucial actions. This also imposes constraints on the agent by expecting it to be capable of performing PGP encryptions.

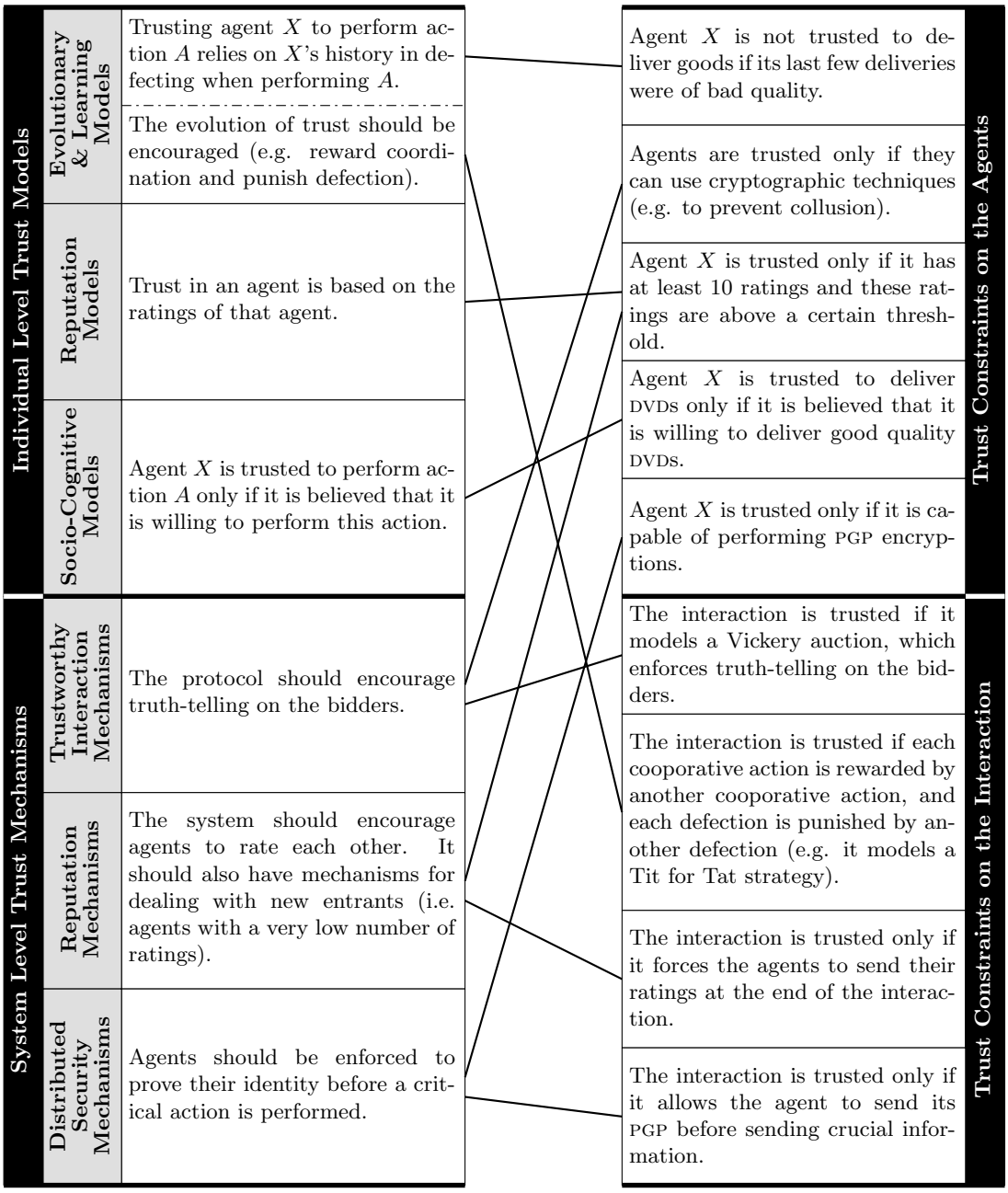
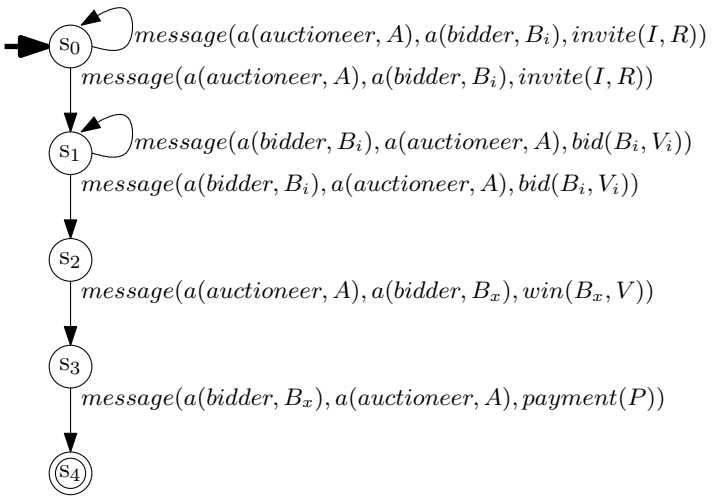


Figure 4.2: Specifying existing trust mechanisms in the contextualised trust model

4.4 Motivating Example

Consider the auction scenario of Figure 4.3. The interaction starts at state s_0 when the auctioneer A sends an invite to a set of bidders for bidding on an item I with a reserve price R. Only after the invites have been sent to all bidders will the interaction move to state s_1. The bidders will then send their sealed bids back to the auctioneer. When all bids are collected, the interaction moves to state s_2. The auctioneer informs the winner of the price V to be paid, moving the interaction to state s_3. Finally, the winning bidder sends its payment details P to the auctioneer, completing the interaction at state s_4.



where,
message(A, B, M) represents the transmission of message M from agent A to agent B, and
$a(R, I)$ represents an agent I playing the role R

Figure 4.3: Auction scenario: the interaction's state graph

Now let us assume that some agent is interested in engaging in the above auction scenario for either selling or buying some music CDs. Trust issues will automatically arise on two different levels. These may be summarised by the following two questions:

- ✦ Is the interaction protocol trustworthy?
- ✦ In such an interaction, which agents are to be trusted?

Answering the first question allows the agent to pick an appropriate interaction protocol for selling/buying its item. The appropriateness of an interaction model is

measured in terms of the satisfiability of certain properties. Traditional properties to check for are usually liveness and safety properties. The auctioneer may decide that the interaction protocol is trusted only if it is deadlock free (trust issue $\mathcal{T}1$ of Figure 4.4). A much more interesting set of properties is obtained when tackling domain specific issues. A challenging trust issue to verify is whether or not the interaction protocol encourages truth-telling on the bidders (trust issue $\mathcal{T}2$ of Figure 4.4).

$\mathcal{T}1$: The interaction model is trusted only if it is deadlock free.

$\mathcal{T}2$: The interaction model is trusted only if it encourages truth telling on the bidders, i.e. the bidders cannot be better off if they bid a value either lower or higher than their true valuation of the item.

$\mathcal{T}3$: If the agent knows from previous experience that DVDs from auctioneer X are not original, then X is not trusted in delivering good quality DVDs.

$\mathcal{T}4$: If the auctioneer agent X is not trusted in delivering good quality DVDs, then it is not trusted in delivering good quality CDs.

$\mathcal{T}5$: Agent X is trusted to take the role of an auctioneer only if it has high ratings. Furthermore, new auctioneers are not accepted, and more importance should be given to the latest ratings.

Figure 4.4: Auction scenario: some trust issues

Answering the second question allows the agent to select a trusted set of collaborating agents. For example, one bidder may trust auctioneer X to sell anything but DVDs, possibly because it knows from previous experience that X's DVDs are not original. It may also use socio-cognitive models to deduce that if the DVDs are not original, then most probably the CDs will not be as well (trust issues $\mathcal{T}3$ and $\mathcal{T}4$ of Figure 4.4). Another widely used trust mechanism is the use of ratings and reputations. The agents should be capable of collecting (or having access to) each others ratings. It is then up to each agent to aggregate these ratings as they see fit. For example, one bidder may decide not to trust new auctioneers with no selling history, and to give more importance to recent ratings (trust issue $\mathcal{T}5$ of Figure 4.4).

The trust issues of Figure 4.4 cover a wide sample of the trust mechanisms of the literature: from socio-cognitive models ($\mathcal{T}4$ of Figure 4.4), to evolutionary and learning models ($\mathcal{T}3$ of Figure 4.4), reputation mechanism ($\mathcal{T}5$ of Figure 4.4), and trustworthy interaction mechanisms ($\mathcal{T}2$ of Figure 4.4).

Note that our current research does not focus on how an agent selects a trust mechanism. Instead, we focus on how an agent may specify and verify its trust constraints.

For example, while we do not focus on how the agent obtains the ratings of others, we do require the specification of how will these ratings be aggregated ($\mathcal{T}5$ of Figure 4.4). After specifying their trust constraints, agents need to verify whether these constraints hold in a given system. We show how the MCID system of Chapter 3 may solve this.

In the remainder of this chapter, our running example will refer to the trust constraints of Figure 4.4, which are to be verified against the auction system of Figure 4.3. The auction system is specified through the LCC process calculus, as presented by Figure 4.5. The auctioneer A — knowing the item I, the reserve price R, and the set of bidders Bs — recursively sends an invite to all bidders of the set Bs. It then takes the role of *auctioneer*2 to collect bids, send the winner a message, collect payment, and deliver the item won. On the other side, each bidder agent receives an invite from the auctioneer and sends its bid based on its valuation. The winner receives a *win* message. It then sends back its payment details P.

$a(auctioneer(I, R, Bs), A)$ ::

$a(auctioneer(I, R, Bs, Bs), A)$.

$a(auctioneer(I, R, Bs, Bs_i), A)$ ::

$$\left(\begin{array}{l} invite(I, R) \Rightarrow a(bidder, B) \leftarrow Bs = [B|T] \ then \\ a(auctioneer(I, R, T, Bs_i), A) \end{array}\right)$$

or

$a(auctioneer2(I, Bs_i, [\,]), A) \leftarrow Bs = [\,]$.

$a(auctioneer2(I, Bs, Vs), A)$ ::

$$append([B, V], Vs, Vn) \leftarrow bid(B, V) \Leftarrow a(bidder, B)\ then$$
$$\left(\begin{array}{l} auctioneer2(I, Bs, Vn), A) \leftarrow not(all\_bid(Bs, Vn)) \\ or \\ \left(\begin{array}{l} win(B_1, V_2) \Rightarrow a(bidder, B_1) \\ \quad \leftarrow all\_bid(Bs, Vn)\ and\ highest(Vn, B_1, \_)\ and\ second\_highest(Vn, \_, V_2)\ then \\ deliver(I, B_1) \leftarrow payment(P) \Leftarrow a(bidder, B_1) \end{array}\right) \end{array}\right).$$

$a(bidder, B)$ ::

$invite(I, R) \Leftarrow a(auctioneer(\_, \_, \_, \_), A)\ then$

$bid(B, V) \Rightarrow a(auctioneer2(\_, \_, \_), A) \leftarrow valuation(I, V)\ then$

$win(B, Vw) \Leftarrow a(auctioneer2(\_, \_, \_), A)\ then$

$payment(P) \Rightarrow a(auctioneer2(\_, \_, \_), A) \leftarrow payment(Vw, P)$.

Figure 4.5: Auction scenario: the LCC interaction model

Note that we do not go through the details of LCC, since the language has already been introduced in Chapter 2 and sample LCC interaction models have been presented in Chapters 2 and 3. The trust constraints of Figure 4.4, however, need to be specified in some trust specification language. The following section introduces our proposed trust policy language.

4.5 Trust Policy Language (TPL)

Chapter 3 introduced the concept of the agents constraints, the deontic model, and suggested the addition of these constraints to the system model and the property specifications fed to the model checker. In this chapter, we focus on the agents' *trust* constraints. We therefore propose a domain specific policy language for the specification of trust rules, as opposed to general deontic rules. The syntax and semantics of our trust policy language (TPL) is presented in the following section, followed by some concrete examples.

4.5.1 Syntax and Semantics

Figure 4.6 presents TPL's syntax. It states that trust rules may either hold in general or under certain conditions: $TrustSpecs[\leftarrow Condition]$. The interaction's trustworthiness is modelled by $trust(interaction(IP), Sign)$, where IP identifies the LCC interaction model in question. $Sign$ could take the values '+' and '−' to model trust and distrust, respectively. The agent's trustworthiness is modelled by $trust(Agent, Sign)$, where the $Agent$ is specified in terms of its role and Id. Only if the agent is trusted, it can engage in an interaction. Trusting or distrusting agents to perform specific actions is modelled by $trust(Agent, Sign, Action)$. Actions could either be message passing actions (MPA) — such as sending ($Message \Rightarrow Agent$) or receiving ($Message \Leftarrow Agent$) messages — or non-message passing actions (N-MPA) — such as performing internal computations. We also allow actions to take the form of another trust rule (note the $TrustSpecs$ in the $Action$ definition). This supports the delegation of trust, since it permits the specification of whether some agent's trust in others is to be trusted or not. For example, one can specify that a hospital's trust in doctors can be trusted: $trust(a(hospital, H), +, trust(a(doctor, id), +))$, while a butcher's trust in doctors is not: $trust(a(butcher, B), -, trust(a(doctor, id), +))$.

The conditions attached to trust rules are usually constructed from structured Pro-

$$
\begin{array}{rcl}
TrustRule & := & TrustSpecs[\leftarrow Condition] \\
TrustSpecs & := & trust(interaction(IP), Sign) \mid \\
 & & trust(Agent, Sign) \mid \\
 & & trust(Agent, Sign, Action) \\
Agent & := & a(Role, Id) \\
Sign & := & + \mid - \\
Action & := & MPA \mid N{-}MPA \mid TrustSpecs \\
MPA & := & Message \Rightarrow Agent \mid \\
 & & Message \Leftarrow Agent \\
Condition & := & Condition \wedge Condition \mid \\
 & & Condition \vee Condition \mid \\
 & & Temporal \mid \\
 & & Term \\
Role, N-MPA, Message & := & Term
\end{array}
$$

where,
IP is either a variable or the URI of an LCC interaction model,
Id is either a variable or a unique agent identifier,
$Temporal$ is a unique identifier of a temporal property, specified in the syntax of Figure 2.7,
$Term$ is either a variable or a structured term in Prolog syntax, except for $N{-}MPA$ terms, which cannot be variables, and
$[X]$ denotes the occurrence of either zero or one instance of X.

Figure 4.6: TPL syntax

log terms using the $\wedge$ and $\vee$ logical operators. However, our TPL syntax allows the addition of temporal conditions too. These are especially, but not exclusively, useful for trust rules constraining the interaction model, since trusting an interaction model usually implies that the interaction model satisfies some temporal properties. Trust rule $\mathcal{T}1$ of Figure 4.4 is one example where the condition is a temporal property specified in the μ-calculus. The following section provides further examples on the specification of various trust issues.

An interesting aspect of TPL is that the syntax is very basic, yet very flexible and powerful. The syntax is simple because the language can describe clear and concise trust rules stating whether an interaction or an agent is trusted in general or whether an agent is trusted to perform specific actions. Constraints may also be added to these rules. However, these constraints make use of the underlying μ-calculus temporal logic and the agent's constraint language, which is Prolog in our case. As a result, anything

that may be expressed in either Prolog or the μ-calculus may be expressed as an additional constraint in an agent's trust rule. This results in a flexible and powerful language with a high degree of expressiveness.

This high degree of expressiveness makes one wonder what the limitations on verifiable properties are. Two classes of limitations occur: the first is imposed by the semantics and the second by the verification mechanism. For example, in the first, the semantics limit agent actions in rules of the form $trust(Agent, Sign, Action)$ to actions that occur in the interaction model. This is because the occurrence of an action, whether an MPA or an N-MPA action, needs to be verified in the scope of a given interaction model. Nevertheless, conditions of trust rules could be general Prolog rules that are unrelated to a given interaction model. These are usually based on the agent's knowledge and belief. However, if the condition contained a temporal property, then this property is usually concerned with the temporal occurrence of actions in a given interaction model.

In the second class of limitations, the nature of the verifier limits the system to be verified to a specific instantiation. For instance, model checking requires system models to be finite. Therefore, the property specification should be verified against a finite system model with a predefined number of agents. For example, as illustrated in Section 4.7, the trust constraints in our auction scenario are verified in a system model with one auctioneer and two bidder agents only. Because the system verified is limited to a specific instantiation, this sometimes implies that certain variables should also be instantiated. For example, the next section illustrates how trust issue $\mathcal{T}1$ needs to be verified for six different bidding cases. In each, the bidding value of each of the two bidders is instantiated by the model checker at run time.

4.5.2 Example: the auction system

We take the view that autonomous agents drive and control multiagent systems; therefore, we focus in this chapter on the agents' trust constraints. Nevertheless, these constraints could either be constraints on the interaction model or constraints on the agents one wishes to interact with. In the latter case, the constraints can either specify the agent's general trust in others or its trust in them performing specific actions. Trust issues $\mathcal{T}1$ and $\mathcal{T}2$ of Figure 4.4 are examples of trust at the interaction level. Whereas trust issues $\mathcal{T}3$, $\mathcal{T}4$, and $\mathcal{T}5$ are examples of trust at the agent level. In this section, we show how these trust constraints may be specified in TPL.

Trust constraints $\mathcal{T}3$ and $\mathcal{T}4$ are specified in TPL in a straightforward manner, as shown by Properties 4.3 and 4.4, respectively. Property 4.3 states that an agent A playing the role *auctioneer* is not trusted in delivering CDs if it is known from experience that it has failed to do so in the past. Property 4.4 states that if agent A playing the role *auctioneer* is not trusted in delivering CDs, then it is not trusted in delivering DVDs too. Trust rule $\mathcal{T}5$, specified in TPL through Property 4.5, is slightly more complex. The rule states that agent A is trusted to play the role *auctioneer* only if it has a selling history with ratings for at least 50 transactions, and at least 70% of its ratings are positive, going up to 90% for the latest 20 transactions. The mechanism presented here distrusts new entrants and focuses on the agent's latest ratings rather than the overall one.

The conditions for trusting interaction models, however, make strong use of temporal properties. The 'deadlock free' property $\mathcal{T}1$ is modelled in a straightforward manner, as shown by Property 4.1 of Figure 4.7. Property 4.1 states that the interac-

$$trust(interaction(IM), -) \leftarrow \mu Z.([-]\mathtt{ff} \wedge \neg completed) \vee \langle - \rangle Z. \tag{4.1}$$

$$\begin{aligned} trust(interaction(IM), +) \leftarrow\ & V'_l < C_l \ \wedge\ C_l < V_l \ \wedge\ V_l < V \ \wedge\ V < V_h \ \wedge\ V_h < C_h \ \wedge\ C_h < V'_h \ \wedge \\ & [bid(Bidder, V), bid(Competitor, C_l)]\langle win(Bidder, X)\rangle \mathtt{tt} \ \wedge \\ & [bid(Bidder, V_l), bid(Competitor, C_l)]\langle win(Bidder, Y)\rangle \mathtt{tt} \ \wedge \\ & [bid(Bidder, V'_l), bid(Competitor, C_l)]\langle win(Competitor, \_)\rangle \mathtt{tt} \ \wedge \\ & [bid(Bidder, V), bid(Competitor, C_h)]\langle win(Competitor, \_)\rangle \mathtt{tt} \ \wedge \\ & [bid(Bidder, V_h), bid(Competitor, C_h)]\langle win(Competitor, \_)\rangle \mathtt{tt} \ \wedge \\ & [bid(Bidder, V'_h), bid(Competitor, C_h)]\langle win(Bidder, Z)\rangle \mathtt{tt} \ \wedge \\ & Y \geq X \ \wedge\ Z \geq X. \end{aligned} \tag{4.2}$$

$$trust(a(auctioneer, A), -, deliver(cd, \_)) \leftarrow know(a(\_, A), deliver(cd, \_), fail). \tag{4.3}$$

$$trust(a(auctioneer, A), -, deliver(dvd, \_)) \leftarrow trust(a(auctioneer, A), -, deliver(cd, \_)). \tag{4.4}$$

$$\begin{aligned} trust(a(auctioneer, A), +) \leftarrow\ & rating\_count(a(auctioneer, A), Total) \ \wedge\ Total > 50 \ \wedge \\ & rating\_average(a(auctioneer, A), Average) \ \wedge\ Average > 0.7 \ \wedge \\ & rating\_latest(a(auctioneer, A), 20, Latest) \ \wedge\ Latest > 0.9. \end{aligned} \tag{4.5}$$

Figure 4.7: Auction scenario: TPL trust constraints

tion model IM is not trusted if eventually a deadlock occurs. This is specified in the μ-calculus as $\mu Z.([-]\texttt{ff} \wedge \neg completed) \vee \langle - \rangle Z$, which states that either no action can occur ($[-]\texttt{ff}$) and the interaction has not completed yet ($\neg completed$) or something can happen ($\langle - \rangle Z$) that will eventually (μZ) lead to this deadlock.

The property $\mathcal{T}2$, which specifies the encouragement of truth-telling on the bidders, is more challenging. To prove the protocol encourages truth-telling on the bidders, we prove that the bidders cannot do any better than bidding their true valuation V, that is the maximum value they are willing to pay for the item. We do not verify whether the agent will actually bid its true valuation or not, since it is impossible to predict how agents will actually act. However, we do verify whether the interaction model provides an incentive for bidders to lie and bid a different value than their true valuation. To achieve this, we need to study all possible bidding cases. Note that the bidding cases take into consideration our current bidder and its competing bidding agent only, since all remaining bidders do not have any effect on our bidder's strategy. The bidding cases are:

- **Case 1:** The competing agent bids a value C_l which is lower than our bidder's true valuation V (i.e. $C_l < V$), and:
  - **Case 1(a):** our bidder bids its true valuation V
  - **Case 1(b):** our bidder bids a value V^- between its true valuation V and the competing agent's bid C_l (i.e. $C_l < V^- < V$)
  - **Case 1(c):** our bidder bids a value V^{--} beneath that of the competing agent's bid C_l (i.e. $V^{--} < C_l$)
- **Case 2:** The competing agent bids a value C_h which is higher than our bidder's true valuation V (i.e. $V < C_h$), and:
  - **Case 2(a):** our bidder bids its true valuation V
  - **Case 2(b):** our bidder bids a value V^+ between its true valuation V and the competing agent's bid C_h (i.e. $V < V^+ < C_h$)
  - **Case 2(c):** our bidder bids a value V^{++} beyond that of the competing agent's bid C_h (i.e. $C_h < V^{++}$)

Figure 4.8 presents an illustration of the various bidding cases. We have focused on only six cases, neglecting the case where the bidder may bid a value V_+, such that $C_l < V < V_+$, or the case where it may bid a value V_-, such that $V_- < V < C_h$. Also, we

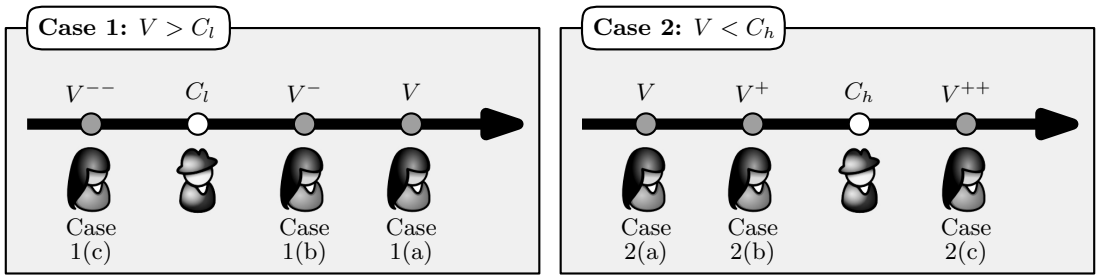


Figure 4.8: Bidding strategies: the 6 different bidding cases

have used strict inequalities, overlooking all equality cases. We intentionally neglected all these cases to keep our example simple and comprehensible; nevertheless, such additional cases may easily be incorporated as needed in a straightforward manner.

Trust Property 4.2 of Figure 4.8 provides a specification of this trust rule in TPL. The six bidding cases are specified through six temporal properties of the form:

$$[bid(Bidder, Bid1), bid(Competitor, Bid2)]\langle win(Winner, Price)\rangle \mathtt{tt}$$

The temporal property is to be interpreted as follows: if the messages $bid(Bidder, Bid1)$ and $bid(Competitor, Bid2)$ are sent, then eventually the message $win(Winner, Price)$ will be received, stating who the winner is and at what price the item has been won [3].

We assume the bidder will lose to its competitor in cases 1(c), 2(a), and 2(b), since it bids a value smaller than that of the competitor. We also assume the bidder to win in cases 1(a), 1(b), and 2(c), since it bids a value higher than that of the competitor. Nevertheless, these assumptions can easily be generalised. As a result of our assumptions, case 1(a) becomes the only winning case in which the bidder would have had bid its true valuation V. The item in that case is won for the price X. The trust constraint requires the interaction model to encourage truth telling on the bidder. In other words, the interaction model should ensure that the bidder is not better off when winning in cases 1(b) and 2(c), where the item is won for prices Y and Z, respectively. This is specified in the TPL Property 4.2 by the condition $Y \geq X \wedge Z \geq X$. Another option is to specify the condition as $Y \geq X \wedge Z \geq V$. This slightly modified version states that if the bidder wins by bidding a lower value than its true valuation, then it will not be

[3] Note that although the temporal property makes use of the *box* and *diamond* modal operators, the property itself does not follow any proper temporal logic syntax. We feel that presenting the property's actual μ-calculus specification at this point will introduce unnecessary additional complexities. Therefore, to keep our example simple and clear, we introduce our own simplified interpretation of the presented syntax. Nevertheless, for the exact specification, we refer the interested reader to Appendix A.2.2.

paying a lower price than bidding its true valuation ($Y \geq X$). Also, if the bidder wins by paying a higher value than its true valuation, then it will be paying a value greater that its true valuation ($Z \geq V$), which is the maximum price the bidder is willing to pay. Clearly, if such conditions hold, then this implies that the bidder should have no incentive for lying in such auction systems.

In our running example, we verify the trust constraints of Figure 4.7 against the interaction model of Figure 4.5. For example, the *IM* in trust property 4.2 is in fact a reference to the LCC interaction model of Figure 4.5. The interaction model is said to be trusted only if the condition of Property 4.2 is satisfied. It is worth noting that this property is restricted to the auctions domain, yet relatively independent of specific interaction models it can be verified against. For example, the six cases of Figure 4.8 are inclusive, even for auction systems consisting of more than two bidding agents. This is because verifying the utility of one agent should take into consideration only the highest competing agent's bid — all other bids are irrelevant. Furthermore, the verification of such a trust rule will terminate with correct results, whether positive or negative, for any auction protocol that requires agents to place their bids before a message is transmitted informing who the winner is. It will fail when verified against more complex auction protocols, such as those selling multiple items and having several winners. However, we believe verification would be successful (regardless of the output) in the most common auctions, such as the English[4], Dutch, sealed first-price, and sealed second-price auctions.

4.6 Verification of Trust in MAS

According to our model, all trust restrictions are imposed by the agents. It is solely the agent's responsibility to answer the *who/when/how* questions of trust. In such highly dynamic systems, this should be done at run time by deciding whether a given interaction model along with a given set of collaborating agents would break any of the agent's trust rules. This automated verification process is carried out by the agents,

[4]While it might be obvious how Property 4.2 would work with interactions where bidders may only bid once, it might not be clear how such a rule would work with other interactions, such as the English auction,were bidders may keep on increasing their bids. We note that existential quantifiers should be used for such cases. For example, if we can say that there exists at least one case in which the bidder may bid a value less than its true valuation and win the item for a lower price, then this implies that the protocol no longer encourages truth-telling on the bidder.

through invoking the model checker of Chapter 3.

As illustrated in Chapter 3, the agent feeds the verifier with the LCC system model, the μ-calculus property specification, and the DPL deontic constraints. However, we now add a new input: the TPL trust constraints. In what follows, we revisit the verification process of Section 3.5.

The verification process is carried out as follows. The interaction model, the general deontic model, the specific trust model, along with any additional temporal properties are fed to the model checker to be verified. The temporal properties are already specified in the μ-calculus. The verifier then converts all DPL and TPL constraints into the μ-calculus as well. It then tries to verify each of those properties against the LCC system model. For each property, verification starts at the initial state s_0 of the interaction model. The model checker tries to verify that the temporal property is satisfied at s_0, following the satisfaction rules of Figure 3.16. If it succeeds, the verifier terminates and the property is said to be satisfied. Otherwise, the verifier will make a transition(s) to the next state(s) in the state-space, following the transition rules of Figure 3.17. The satisfaction of the property is then verified with respect to the new state(s), and the whole process is repeated all over again until a result is reached. The issue of termination is dealt with by making use of the XSB tabled Prolog system, which basically caches previous results in tables to ensure termination. This was explained in more detail in Section 3.5.

The modified model checking algorithm is presented in Figure 4.9. Note that the only difference between this algorithm and that of Figure 3.13 is the addition of a new input: the set D_t consisting of TPL trust rules. This requires the TPL constraints to be translated into μ-calculus formulae. This step is specified in Figure 4.9 as follows: $\forall d_t \in D_t.\ translate(d_t, t) \wedge P \stackrel{\text{is}}{=} \{t\} \cup P$. The translation, expressed as $translate(d_t, t)$, follows the mapping rules of Table 4.1.

Table 4.1 presents the mapping from TPL to the μ-calculus. Any trust rule with a positive sign is automatically satisfied if its condition is satisfied. Such trust rules are modelled as $satisfied(Condition) \vee (\neg satisfied(Condition) \wedge X)$. Mappings number 1, 3, and 5 of Table 4.1 are an example. However, if the condition of such a trust rule is not satisfied, then we assume that the entity in question is not trusted. If the entity in question is an interaction model, then the satisfaction of this trust rule should fail. Hence, X becomes `ff` in mapping number 1. If the entity in question is an agent, then X becomes $\nu Z.[Actions]\texttt{ff} \wedge [-]Z$, where $Actions = [in(a(Role, Id), \_), out(a(Role, Id), \_), \#(a(Role, Id), \_)]$. This implies that if the agent is not trusted, then the trust rule is

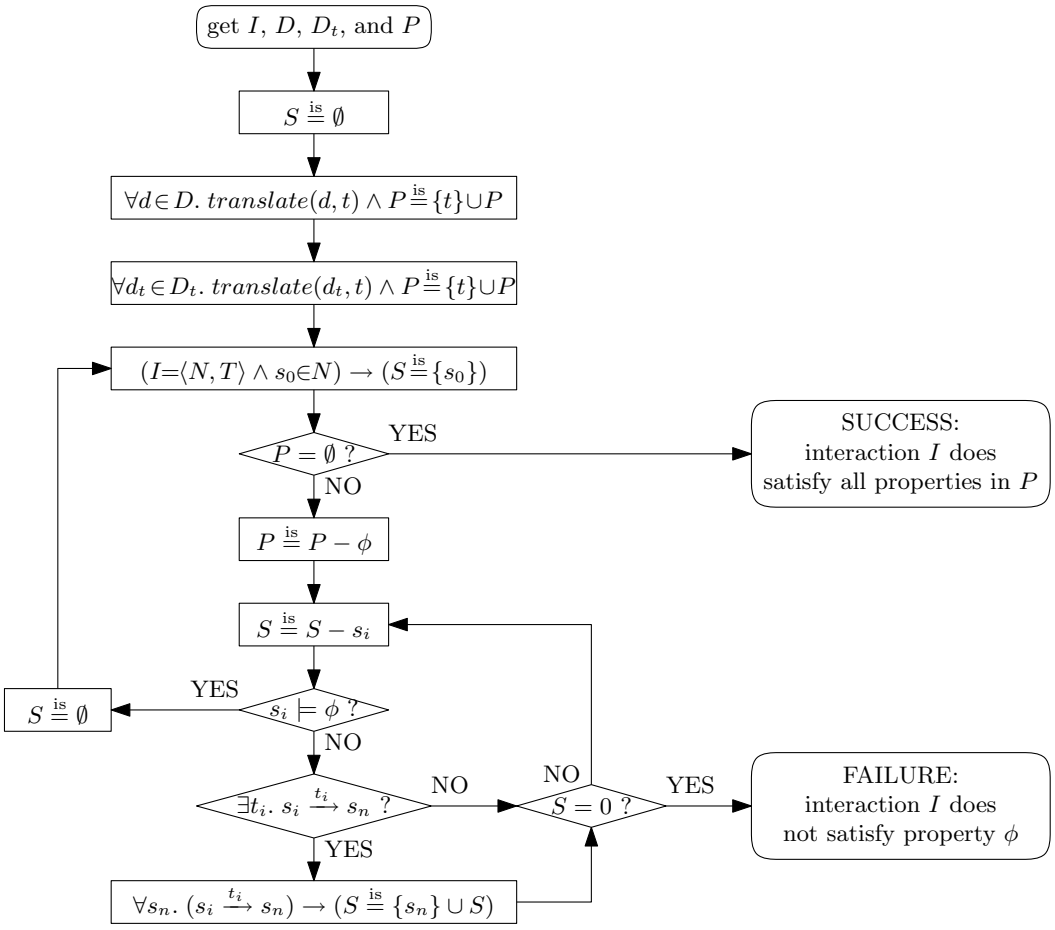


Figure 4.9: MCID's algorithm: a modified version incorporating trust constraints

satisfied if and only if the agent does not perform any action in this interaction. Note that this constraint, which prevents the agent from performing any action, is modelled in the *mu*-calculus. $[Actions]\texttt{ff}$ implies that the *Actions* are not permitted, whereas $[-]Z$ ensures that this property is satisfied for every path of the state graph and for every state in that path (νZ). Also note that actions could be input ($in(a(Role, Id), \_)$), output ($out(a(Role, Id), \_)$), or internal computations ($\#(a(Role, Id), \_)$). If the agent is not trusted to perform a specific action, then X becomes $\nu Z.[Action']\texttt{ff} \wedge [-]Z$, where $Action'$ is the specified input, output, or internal computation action.

For negative trust rules, the complement of the logic above applies. If conditions are satisfied, then the entities in question are not trusted. Such trust rules are modelled as $(satisfied(Condition) \wedge Y) \vee \neg satisfied(Condition)$. Mappings number 2, 4, and 6 of Table 4.1 are an example. Similar to the above logic, if the untrusted entity is an interaction model, then Y becomes $\texttt{ff}$ (see mapping number 2).

#	**TPL Predicate**	μ **– Calculus Equivalence**
1.	$trust(interaction(IM), +) \leftarrow Condition.$	$satisfied(Condition) \vee$ $(\neg satisfied(Condition) \wedge \texttt{ff})$.
2.	$trust(interaction(IM), -) \leftarrow Condition.$	$(satisfied(Condition) \wedge \texttt{ff}) \vee$ $\neg satisfied(Condition)$.
3.	$trust(a(Role, Id), +) \leftarrow Condition.$	$satisfied(Condition) \vee$ $(\neg satisfied(Condition) \wedge \nu Z.[Actions]\texttt{ff} \wedge [-]Z)$. where $Actions = [in(a(Role, Id), \_),$ $out(a(Role, Id), \_),$ $\#(a(Role, Id), \_)]$
4.	$trust(a(Role, Id), -) \leftarrow Condition.$	$(satisfied(Condition) \wedge \nu Z.[Actions]\texttt{ff} \wedge [-]Z) \vee$ $\neg satisfied(Condition)$. where $Actions = [in(a(Role, Id), \_),$ $out(a(Role, Id), \_),$ $\#(a(Role, Id), \_)]$
5.	$trust(a(Role, Id), +, Action) \leftarrow Condition.$	$satisfied(Condition) \vee$ $(\neg satisfied(Condition) \wedge \nu Z.[Action']\texttt{ff} \wedge [-]Z)$. where $Action' = \begin{cases} in(a(Role, Id), M) & \text{, if } Action = in(M) \\ out(a(Role, Id), M) & \text{, if } Action = out(M) \\ \#(a(Role, Id), Action) & \text{, elsewhere} \end{cases}$
6.	$trust(a(Role, Id), -, Action) \leftarrow Condition.$	$(satisfied(Condition) \wedge \nu Z.[Action']\texttt{ff} \wedge [-]Z) \vee$ $\neg satisfied(Condition)$. where $Action' = \begin{cases} in(a(Role, Id), M) & \text{, if } Action = in(M) \\ out(a(Role, Id), M) & \text{, if } Action = out(M) \\ \#(a(Role, Id), Action) & \text{, elsewhere} \end{cases}$

Table 4.1: Mapping TPL predicates into μ-calculus formulae

If the untrusted entity is an agent, then Y becomes $\nu Z.[Actions]\texttt{ff} \wedge [-]Z)$, where $Actions = [in(a(Role, Id), \_), out(a(Role, Id), \_), \#(a(Role, Id), \_)]$. And if the agent is not trusted to perform a specific action, then Y becomes $\nu Z.[Action']\texttt{ff} \wedge [-]Z)$, where $Action'$ is the specified input, output, or internal computation action. However, if the condition of a negative trust rule is not satisfied, then we believe there is not need to distrust the entity. Therefore, the dissatisfaction of a negative trust rule's condition is enough proof that the trust rule is satisfied in the given model.

4.7 Results

The complexity of verifying multiagent systems depends on the complexity of the different roles in the interaction rather than the number of agents involved in such an interaction, as we illustrate shortly. In Figure 4.5, interaction protocols are defined in terms of agent roles rather than the individual agents that might engage in these

interactions. For example, there are two roles for our auction scenario: the role of the auctioneer and that of the bidder. All bidders then share the same protocol specification, irrespective of their different local deontic constraints. The complexity of verifying the five trust constraints of Figure 4.7 depends on the complexity of the auctioneer's and bidder's role definitions. Of course, the number of agents playing each role raise a scalability concern. However, in our auction scenario, if the trust rules are verified for a set of one auctioneer agent and two bidder agents, then the results will hold for a set of one auctioneer agent and n bidder agents, where $n \geq 2$. The trick is to know the minimum number of agents required for verifying certain properties of a given interaction protocol. We note that this is not always trivial and may require additional proofs. However, at the time being, we assume such information is provided with the interaction protocol. We believe this to be an acceptable assumption because the properties are currently being handcrafted by the same designers who design the interaction protocol. Therefore, the designer may easily specify the minimum number of agents required for the verification of each property.

To verify our auction system, we set the scene to incorporate one auctioneer and two bidding agents. The dynamic model checker is then invoked for verifying the five trust issues of Figure 4.7 against the auction scenario of Figure 4.5. The CPU time and the memory usage, from the moment the model checker is invoked until the results are returned to the user, are presented by Figure 4.2.

	$\mathcal{T}1$: **Property 4.1**	$\mathcal{T}2$: **Property 4.2**	$\mathcal{T}3$: **Property 4.3**	$\mathcal{T}4$: **Property 4.4**	$\mathcal{T}5$: **Property 4.5**
CPU time (in *sec*)	0.017	0.204	0.020	0.021	0.010
Memory usage (in MB)	1.327	14.052	1.356	1.376	0.917

Table 4.2: Auction scenario: verification results

Note that results for $\mathcal{T}2$, in Figure 4.2, differ drastically from others since $\mathcal{T}2$ is essentially constructed from 6 temporal properties, while the others are single temporal properties. Also note that the CPU time in both the scenarios of Chapters 3 and 4 do not exceed $1sec$. Although these are preliminary results, they do prove that dynamic verification is in fact possible in such realistic scenarios.

4.8 Conclusion

Trust is the key to the success of distributed open systems. Most of the work carried out in the field of trust has focused on developing strategies that would ensure trusted scenarios. In this chapter, we have presented a mechanism for specifying and verifying such strategies. Our mechanism allows the agents involved to dynamically and automatically invoke the model checker at run-time, when the conditions for verification are met, in order to verify whether their trust requirements will be broken or not for a given scenario with a given set of collaborating agents.

In summary, the novelty of our work is in introducing *interaction time verification* and applying it to the field of trust in multiagent systems. This is made feasible by using a relatively compact (implemented in 200 lines of Prolog code), yet efficient (as shown by the results of Figure 4.2), dynamic model checker. The main achievement is that agents, when faced with new and unexplored interaction models, may now verify whether engaging in such interactions would break their trust constraints.

Also note that the trust specification language presented in this chapter has a very simple and basic syntax (Figure 4.6). Yet, this syntax is very flexible, and hence, very powerful. What makes these trust rules very flexible is the addition of conditions. These could either be defined as Prolog facts and rules or as further temporal properties that should hold in the given system model, resulting in a very expressive trust policy language. An additional interesting aspect, is the ease of verification of these rules. As illustrated by Section 4.6, trust constraints are equivalent to temporal properties on the interaction model. Even if the constraints are on agents, in practise they are verified by checking whether (dis)trusted agents perform any illegal actions in a given interaction. Therefore, all that is needed is the addition of a simple translator to convert DPL constraints in μ-calculus formulae, leaving the original model checker intact.

4.9 Future Work

Both the DPL and TPL languages presented in Chapters 3 and 4 for defining the agent's constraints are simple and basic yet very expressive languages, since they make use of the underlying constraint specification language used, which is Prolog in our case, and the μ-calculus temporal logic. The examples presented in this thesis have also proven the capability of automatically verifying such constraints at run time. However, we recommend further work to be carried out to test the limits of our model checker. This

requires a comprehensive study of possible and realistic scenarios, their interesting temporal properties, and the level of complexity that these could reach. We hope to answer questions such as: What size of a system model (or state-space) would break the model checker? What properties can not be verified against a finite system model with a finite number of agents? If such problems arise, are there any solutions that would work around them? etc.

Another issue which we overlook is addressing conflicting rules. In traditional policy languages, this raises a huge concern. For example, one needs to know whether access should be granted to a given user or not, making conflict resolution a basic concern for policy languages. However, in our case, we do not use the policy language to decide on critical actions, such as granting access permissions. We use the policy language to verify whether certain constraints are satisfied in a given interaction model or not. We currently do this by verifying all constraints, even if they are conflicting. If any of the constraints is not satisfied, then the system is not trusted. To illustrate why conflicting rules are not that critical in our system, consider the following example. Assume there are two conflicting rules: the first states that the agent is trusted to deliver DVDs, and the second states that the agent is not trusted to deliver DVDs. The first rule is always satisfied in any interaction model. The second rule is satisfied in an interaction model only if the agent does not deliver DVDs, and fails otherwise. In other words, our system gives precedence to negative rules over positive ones, that is precedence to prohibitions over permissions. It also assumes that any agent is trusted to perform any action by default, unless stated otherwise. Future work may illustrate how these defaults can be changed and how different precedence rules can be applied according to the verifying agent's requirements.

Another current concern is the automation of the verification process. We have presented an automated verification mechanism that could be used by the agents at run time to aid them select a suitable interaction model and a suitable set of collaborating agents. However, as illustrated in Section 2.1.2.1, every verification process must be preceded by modelling and specification processes. We remove the need for a modelling process, since our model checker verifies the actual executable model specified via the LCC process calculus. We believe such interactions would be made available by various services in the system. The agent can then use our proposed model checker to verify whether a given interaction model is trustworthy or not. The verification process itself is truly automated; however, a current obstacle preventing full automation is the specification of the properties that need to be verified. This raises two questions:

- ✦ Can the agent learn which properties are important?
- ✦ Can the agent specify such properties in an automated manner?

We hope future work would address these questions. At the moment, we assume intervention from the agent's user to specify its required trust constraints. We also hope that a list of commonly used properties would be made available for agents to select from and use for verifying their interaction models. Some of these properties could be general properties, such as properties describing deadlocks. Others may be domain specific, yet independent of a particular interaction model. Trust property 4.2 is one example which is specific to the auctions domain, but general enough to be verified against a range of auction interaction models. However, even in the case of these general or domain specific properties, some tuning should be applied before they can be verified against a new interaction model. For example, the domain specific property 4.2 assumes that the message informing the winner W of the price P to be paid is specified as $win(W,P)$. This message may be specified as $inform\_win(W,I,P)$ in another interaction model. We believe available research, such as that done on matching and ontology mapping, may be used to address this issue. At the moment, we leave this for future work.

Last, but not least, our proposed specification and verification mechanism makes use of existing trust models. We hope to integrate our trust verification mechanism with some of these existing trust models. For example, trust property 4.5 defines (in Prolog) how collected ratings may be aggregated, assuming that a reputation model for collecting these ratings already exists. We plan to work on linking our verifier to an existing reputation model in order to achieve a realistic and complete running scenario.

Chapter 5

Verification of an OpenKnowledge eResponse Scenario

Chapter 3 has introduced the MCID model checker that may be invoked by agents at run time for verifying whether certain temporal system properties and agent deontic constraints hold in a given system model. Chapter 4 illustrated the possible application of MCID to the field of trust; agents may verify whether their specific trust constraints (as opposed to general deontic constraints) will be broken or not. The scenarios verified in Chapters 3 and 4 are a travel agency scenario and an auction one, respectively.

In the previous chapters, the agents were expected to verify a given interaction model against a given set of deontic/trust constraints before joining the interaction. In this chapter, we illustrate how the MCID system may be used in two different ways. In the first, the model checker is invoked online by an agent which pauses one interaction model in order to verify and execute another interaction, before resuming the initial one. Compared to the scenarios of previous chapters, agents in this new scenario can verify and execute interactions from within other interaction models. In the second method, the model checker is invoked offline to verify properties of complex systems composed of a mesh of interconnected interaction models, as opposed to properties of a single interaction model. This implies that the verification results of one interaction may depend on the verification results of others in this mesh of interactions.

The chapter opens with Section 5.1, which provides a brief introduction to the OpenKnowledge project and its eResponse testbed, from which we obtain the scenarios used in this chapter. This is followed by a detailed illustration of the scenarios to be verified and the properties they should be verified against in Section 5.2. Section 5.3 presents the results of our verification process, before concluding with Section 5.4.

5.1 The OpenKnowledge eResponse Testbed

Section 5.1.1 provides a brief overview to the OpenKnowledge project and the reasons behind our choice of such a testbed. This is followed by an introduction to the OpenKnowledge flooding eResponse simulation system in Section 5.1.2.

5.1.1 The OpenKnowledge Project: an Overview

OpenKnowledge (OK) is a three years EC funded project with participants from Edinburgh, Amsterdam, Barcelona, Trento, Southampton, and the Open University. The general goal of the project is to achieve practical open and distributed systems in which agents may participate at low and acceptable costs. To achieve this, the project proposes to shift the focus from the different system entities (such as peers, agents, web services, etc.) to the interactions that connect these entities together. An interaction, defining precisely how each agent role may execute its part of the interaction, is specified through an LCC interaction model. Issues, such as commitments or semantic agreements, are then dealt with at the interaction level. Grounding such issues to a specific interaction model facilitates the process of addressing them in a more realistic and dynamic approach. The OK system provides the methods needed for finding, sharing, and executing interactions. However, a crucial question is: *"how do the agents select their interactions?"* Several methods, making use of matching, reputation, and trust mechanisms, are currently being implemented in the OK project to address this issue. In this chapter, however, we show how this issue may be addressed through the analysis of interactions and the properties they satisfy. Our MCID system seems to be an appropriate candidate for this job. Furthermore, the LCC nature of the OK system interactions provide our model checker with the opportunity of verifying richer scenarios.

In this thesis, we are not concerned with the OK system architecture and its technical details. For instance, we are not concerned with how peers search for interaction models, how peers search for other peers in the network, how LCC interactions are executed, and so on. The MCID system offers peers the ability to verify their interactions before executing them. It assumes the interaction model, along with the properties it should be verified against, have already been obtained by the peer. To perform the verification, peers should have the MCID system installed.

The MCID system itself is a lightweight system, coded in less than 200 lines of Prolog. This compact size and the efficient nature of the model checker contribute to the success of interaction time verification. Nevertheless, our verifier resides on top

of the XSB system. The reason behind using the XSB system is to ensure termination by making use of its tabling mechanism that effectively caches results in tables. This issue has been discussed in more detail earlier in Section 3.5. The XSB system requires a bit less than 20MB to install. We believe it is safe to assume that agents can have this commodity. In cases where this is not feasible, agents may make use of a model checking web service that receives verification requests and returns the results of the verification process.

What interests us in the OK system, and what this chapter focuses on, are the various LCC scenarios and how they may be verified. Therefore, for further information on the OpenKnowledge project and its implementation details, we refer the interested reader to the published paper by Siebes et al. (2007) and the project website: `www.openk.org`.

The OK project currently has two testbeds. The first is in the bio-informatics field, and the second is in the emergency response (eResponse) field. We choose to focus on the interaction models of the second merely because they provide us with a richer selection of interactions along with more interesting properties to verify. The following section provides an overview of the flooding eResponse simulation system[1].

5.1.2 The Flooding eResponse Simulation System

The flooding eResponse simulation system, presented in Figure 5.1, is composed of two major components: the eResponse simulator and the peer network. The simulator is composed of three peers: the controller, the flood sub-simulator, and the visualiser. The controller constitutes the core of the simulator; it drives the simulation cycles. It has one main goal: to keep track of the current state of the simulated world. In order to achieve that, the controller needs to know what changes are happening to the world and update its state accordingly. After updating its state, it informs the relevant peers of these changes. The goal of the flood sub-simulator is to simulate the flood. It is composed of a set of predefined equations that specify how the flood may evolve with time. The goal of the visualiser is to present all the changes that are happening in the world through a graphical user interface.

Simulation is carried out through cycles, or time steps. The simulation steps are defined as follows:

[1]The introduction to the OK eResponse system provided in Section 5.1.2 is based on a brief section written by the author of this thesis in the OK project deliverable Marchese et al. (2008).

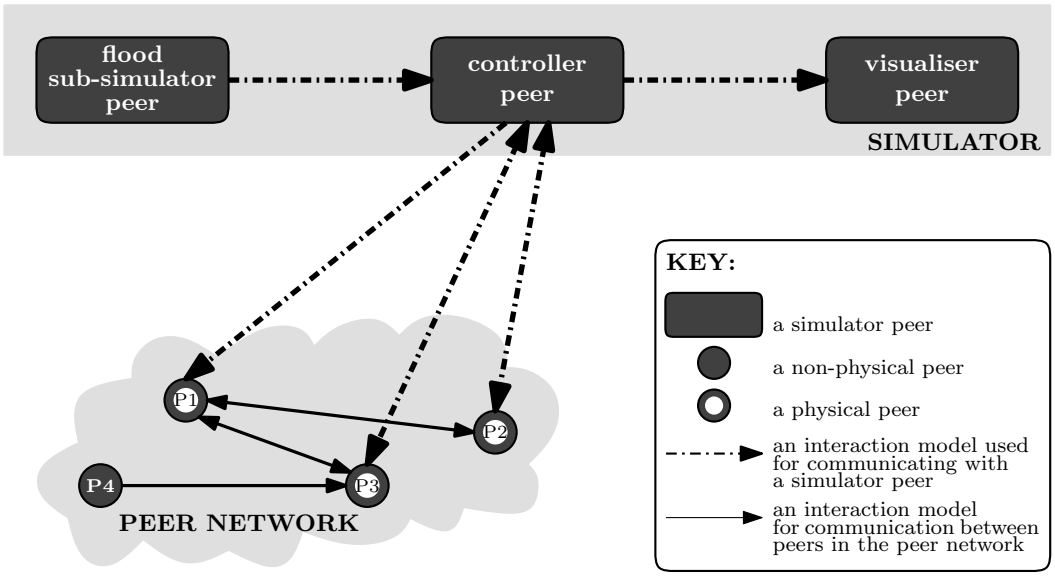


Figure 5.1: The eResponse simulation system

1. The controller coordinates the initial topology of the world with the other simulator peers: the flood sub-simulator and the visualiser.

2. The controller receives information about the changes that have occurred in the world:

   (a) it receives the flood changes from the flood sub-simulator, and

   (b) it receives other changes from the peers in the peer network that inflicted these changes by performing some physical actions. Note that for each action performed, the controller is also responsible for verifying its validity. This is discussed shortly in more detail.

3. The controller sends information to the relevant peers about the changes that have occurred in the world:

   (a) it sends a list of all the changes to the visualiser peer, and

   (b) for each physical peer in the peer network, it sends the peer the changes that occurred in its vicinity.

4. The controller updates its time step, and goes back to step 2.

The flow of information between peers is illustrated in Figure 5.1 through the arrows connecting the various peers. These arrows are essentially LCC interaction models. Peers in the peer network are of two kinds: physical peers and non-physical peers.

Physical peers represent peers that interact with the physical world. For example, firemen are physical peers since they may perform physical actions. Sensors are also considered physical peers, since they perceive information from and about the physical world. A web service, however, does not need to interact with the physical world. Hence, it is considered to be a non-physical peer.

Non-physical peers do not need to connect to the simulator. They may communicate only with other peers in the peer network. However, it is essential that physical peers are connected to the simulator's controller for two reasons. First, they should be able to sense the changes around them in the world. This is achieved by receiving sensory information about the changes in their vicinity from the simulator's controller at every time step, as depicted by step 3(b) of the simulation cycle. Second, they should inform the simulator of any physical actions they perform. For example, if a fireman rescues people by transporting them from one site to another, then this should be communicated to the controller, which is responsible for keeping track of the current state of the world. This is depicted as step 2(b) of the simulation cycle.

The interactions of the peer network presented in Figure 5.1 are a representation of a realistic eResponse scenario (Marchese et al., 2008). Peer $P1$ represents the firefighters coordinator. The firefighters coordinator is a physical peer; hence, it needs to be communicating with the controller. However, in this scenario, all that peer $P1$ does is request the firefighter peers, $P2$ and $P3$, to carry out certain actions. Therefore, peer $P1$ does not need to inform the controller of any physical actions it performs, since there are none. Since it still needs to receive sensory information about its vicinity at every time step, the flow of information is bidirectional: from the controller to the firefighters coordinator. The firefighters, on the other hand, need to receive sensory information as well as send the details of their physical actions to the controller. This is depicted by the double arrow lines connecting the controller peer to both $P2$ and $P3$. Note that peers $P2$ and $P3$ are also communicating with $P1$. In order to perform the action of transporting people from location $L1$ to location $L2$, $P3$ needs to know the path that links $L1$ to $L2$. It gains this information by communicating with a route service peer, $P4$.

Simulator peers are static peers with predefined functionalities. The interaction models for interacting with the simulator peers are also predefined with respect to the steps of the simulation cycle. On the other hand, the peer network is dynamic. It is impossible to predict which peers will be connected in the current simulation, what actions will they perform, which interactions will be executed, etc. Since the peer

network provides the peers with the opportunity of selecting their interactions from a variety of models, we choose to focus on its interaction models. The following section introduces the eResponse scenario to be verified by our MCID system. This scenario is similar to the firefighters scenario presented above.

5.2 An eResponse Scenario

The system model of our eResponse scenario is introduced in Section 5.2.1. This is followed by Section 5.2.2, which introduces two different ways for applying the MCID system, resulting in local online and global offline verification, along with the properties the system model is verified against.

5.2.1 The System Model

In a flooding eResponse scenario, consider the case where the coordinator of some firefighters group needs to instruct its firefighters to perform certain actions. We consider this to be the main interaction model that we wish to verify. We refer to this interaction as interaction IM_0. The basic actions that may be requested from firefighters, and are currently accepted by the simulator, are the *move*, *pick*, and *drop* actions. Pick and drop actions may be trivial, as long as the firefighter is at the right location and has the capability of picking objects and/or citizens. Performing the move action is a bit trickier, since it requires the firefighter to know the path it should take to reach its destination. In a flooding scenario, the status of roads may change with time, providing an additional challenge for the firefighters. Obtaining the path between two locations is specified in interaction IM_0 as the LCC constraint: $realise\_goal(get\_path(Node1, Node2, Vehicle, Path))$. How this constraint is satisfied at run time depends on the specific firefighter executing that interaction. In what follows, we present a variety of ways for satisfying such a constraint.

- If the firefighter already knows the path between two nodes, then the constraint $realise\_goal(get\_path(...))$ is satisfied locally by consulting the firefighter's knowledge base (KB).

- If the firefighter does not know the path between two nodes, then it needs to engage in some other interaction for obtaining such information. We present two different interactions that could help the agent achieve its goal:

- ✧ In the first interaction, the firefighter asks a route service about the path linking the two nodes. We refer to this interaction as interaction IM_1.
- ✧ In the second interaction, the firefighter asks other agents in the network, possibly its trusted group of friends or agents in its destination's vicinity, about the path linking the two nodes. We refer to this interaction as interaction IM_2. We note that such an interaction is especially useful in emergency situations where regular services may fail (for instance, due to heavy traffic), since it allows peers to rely on any other available peer in the network for help.

This collection of various scenarios, which constitute the main scenario we wish to verify, is illustrated in Figure 5.2. There are three different interaction models involved in this example: IM_0, IM_1, and IM_2. The LCC specification of these interactions are presented in Figures 5.3, 5.4, and 5.5, respectively.

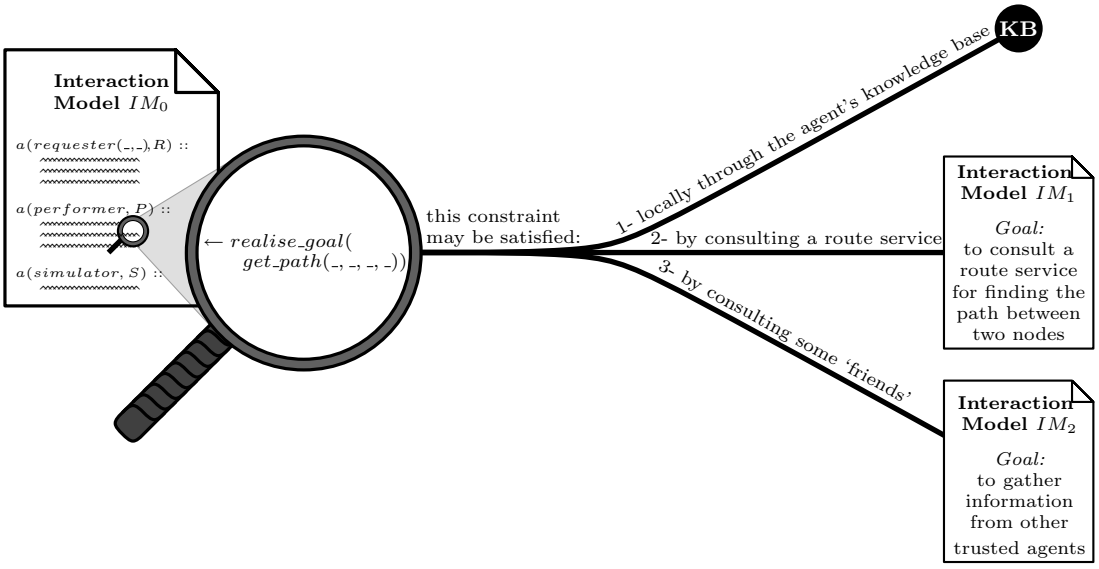


Figure 5.2: eResponse scenario: an overview

The interaction model of Figure 5.3 specifies that the peer coordinating other peers' actions should play the role *requester*. Given the list of actions, which should explicitly specify which peer needs to perform which action, the *requester* peer takes on three consecutive roles. First, it takes the role *requester*2, which performs a recursion over the list of actions. For each action, a message is sent out the appropriate peer asking it to perform this action. After all the action messages have been sent out, the requester peer takes the role *collector* for collecting the results of the peers' actions. After all

$$
\begin{array}{l}
a(requester(Actions, Results), R) \ :: \\
\quad null \leftarrow get\_peers(Actions, [\,], PeersSet) \ then \\
\quad a(requester2(Actions), R) \ then \\
\quad a(collector(Actions, [\,], Results), R) \ then \\
\quad a(informing\_completion(PeersSet), R).
\end{array}
$$

$$
\begin{array}{l}
a(requester2(Actions), Co) \ :: \\
\quad \left(\begin{array}{l} action(A) \Rightarrow a(performer, Peer) \leftarrow Actions = [(Peer, A)|T] \ then \\ a(requester2(T), Co) \end{array} \right) \\
\quad or \\
\quad null \leftarrow Actions = [\,].
\end{array}
$$

$$
\begin{array}{l}
a(collector(Actions, OldResults, NewResults), R) \ :: \\
\quad null \leftarrow Actions = [\,] \ and \ NewResults = OldResults \\
\quad or \\
\quad \left(\begin{array}{l} result(A, Result) \Leftarrow a(performer, P) \ then \\ null \leftarrow my\_select((P, A), Actions, Actions2) \ and \ my\_append((P, A, Result), OldResults, IntResults) \ then \\ a(collector(Actions2, IntResults, NewResults), R) \end{array} \right).
\end{array}
$$

$$
\begin{array}{l}
a(informing\_completion(PeersSet), R) \ :: \\
\quad \left(\begin{array}{l} done \Rightarrow a(performer, P) \leftarrow PeersSet = [P|T] \ then \\ a(informing\_completion(T), R) \end{array} \right) \\
\quad or \\
\quad null \leftarrow PeersSet = [\,].
\end{array}
$$

$$
\begin{array}{l}
a(performer, C) \ :: \\
\quad \left(\begin{array}{l}
action(transport(O, N1, N2)) \Leftarrow a(requester2(\_), Co) \ then \\
\left(\begin{array}{l} null \leftarrow at(N1) \\ or \\ a(performer2(move(N0, N1, Path0, Vehicle), R1), C) \\ \quad \leftarrow at(N0) \ and \ not(N0 = N1) \ and \ realise\_goal(get\_path(N0, N1, Vehicle, Path0)) \\ \quad\quad and \ route\_details(Path, Vehicle, Fuel0, Time0) \end{array} \right) \ then \\
a(performer2(pick(O, N1), R2), C) \leftarrow at(N1) \ then \\
a(performer2(move(N1, N2, Path, Vehicle), R3), C) \\
\quad \leftarrow realise\_goal(get\_path(N1, N2, Vehicle, Path)) \ and \ route\_details(Path, Vehicle, Fuel, Time) \ then \\
a(performer2(drop(O, N2), R4), C) \leftarrow at(N2) \ then \\
result(transport(O, N1, N2), R) \Rightarrow a(collector(\_, \_, \_), R) \leftarrow aggregate\_results([R1, R2, R3, R4], R) \ then \\
a(performer, C)
\end{array} \right) \\
\quad or \\
\quad \left(\begin{array}{l}
action(move(N2)) \Leftarrow a(requester2(\_), Co) \ then \\
a(performer2(move(N1, N2, Path, Vehicle), R), C) \\
\quad \leftarrow at(N1) \ and \ realise\_goal(get\_path(N1, N2, Vehicle, Path)) \ and \ route\_details(Path, Vehicle, Fuel, Time) \ then \\
result(move(N2), R) \Rightarrow a(collector(\_, \_, \_), R) \ then \\
a(performer, C)
\end{array} \right) \\
\quad or \\
\quad done \Leftarrow a(informing\_completion(\_), R).
\end{array}
$$

$$
\begin{array}{l}
a(performer2(Action, Result), Id) \ :: \\
\quad action(Action) \Rightarrow a(simulator, S) \leftarrow simulator\_id(S) \ then \\
\quad ok \Leftarrow a(simulator, S) \ then \\
\quad null \leftarrow perform(Action, Result).
\end{array}
$$

$$
\begin{array}{l}
a(simulator, S) \ :: \\
\quad action(Action) \Leftarrow a(performer2(\_, \_), Id) \ then \\
\quad ok \Rightarrow a(performer2(\_, \_), Id) \ then \\
\quad a(simulator, S).
\end{array}
$$

Figure 5.3: eResponse scenario: the LCC specification of interaction model IM_0

results have been received, the requester takes the role $informing\_completion$, which sends *done* messages to all the peers in order to notify them that no further actions will be requested in this interaction, signalling the completion of their roles.

The actions that may be requested from peers to perform are of two kinds: (1) asking a peer to transport someone or something, and (2) asking a peer to move to another location. To illustrate the first case, consider that the peer is asked to transport object O from node $N1$ to node $N2$. If the peer is not already at node $N1$, then it should first move to $N1$. This is followed by picking up the object, moving to node $N2$, and dropping the object at $N2$. Finally, the results of its action are sent back to the requester (now playing the role *collector*) through a $result(\_, \_)$ message. Note that performing move actions requires the peer to obtain the path it needs to take by satisfying the constraint $realise\_goal(get\_path(...))$. Similarly, in the second case, when the peer is asked to perform a move action, the peer will have to get the path it needs to take, play the role $performer2$ to perform the action, and send the results back to the requester.

Performing the move, pick, and drop actions is carried out by the role $performer2$, which is responsible for coordinating these actions with the simulator's controller before they are actually carried out. The role $performer2$ sends the action to the simulator's controller, receives the green light from the controller through the *ok* message, and finally performs the action by satisfying the LCC constraint $perform(Action, Result)$. To keep things simple, we currently assume that the simulator will accept all actions. Of course, this may easily be modified as needed.

$$
\begin{array}{l}
a(route\_finder(Location1, Location2, Vehicle, UnacceptableRoads, Path), Id) \ :: \\
\quad request\_route(Location1, Location2, Vehicle, UnacceptableRoads) \Rightarrow a(route\_service, RS) \\
\quad\quad \leftarrow route\_service\_id(RS) \ then \\
\quad route(Path) \Leftarrow a(route\_service, RS).
\end{array}
$$

$$
\begin{array}{l}
a(route\_service, RS) \ :: \\
\quad request\_route(L1, L2, V, UnacceptableRoads) \Leftarrow a(route\_finder(\_, \_, \_, \_, \_), Id) \ then \\
\quad \left(\begin{array}{l}
route(Result) \Rightarrow a(route\_finder(\_, \_, \_, \_, \_), Id) \\
\quad\quad \leftarrow find\_route(L1, L2, V, UnacceptableRoads, Result) \\
or \\
route([\,]) \Rightarrow a(route\_finder(\_, \_, \_, \_, \_), Id)
\end{array} \right).
\end{array}
$$

Figure 5.4: eResponse scenario: the LCC specification of interaction model IM_1

$$
\begin{array}{l}
a(inquirer(Question, Result), I) \ :: \\
\quad \left(\begin{array}{l} null \leftarrow trusted\_agents(Question, Agents) \\ or \\ a(inquirer1(Question, Finders, [\,], Agents), I) \leftarrow finders(Question, Finders) \end{array} \right) then \\
\quad a(inquirer2(Question, Agents, Result), I).
\end{array}
$$

$$
\begin{array}{l}
a(inquirer1(Q, Fs, As, FinalAs), I) \ :: \\
\quad \left(\begin{array}{l} request\_suitable\_peers(Q) \Rightarrow a(finder, F) \leftarrow Fs = [F|NewFs] \ then \\ append(Ps, As, NewAs) \leftarrow suitable\_peers(Q, Ps) \Leftarrow a(finder, F) \ then \\ a(inquirer1(Q, NewFs, NewAs, FinalAs), I) \end{array} \right) \\
\quad or \\
\quad null \leftarrow Fs = [\,].
\end{array}
$$

$$
\begin{array}{l}
a(inquirer2(Q, Rs, Result), I) \ :: \\
\quad null \leftarrow empty(Rs) \ and \ Result = nil \\
\quad or \\
\quad \left(\begin{array}{l} question(Q) \Rightarrow a(responder, R) \leftarrow Rs = [R|NewRs] \ then \\ \left(\begin{array}{l} answer(Result) \Leftarrow a(responder, R) \\ or \\ \left(\begin{array}{l} no\_answer \Leftarrow a(responder, R) \ then \\ a(inquirer2(Q, NewRs, Result), I) \end{array} \right) \end{array} \right) \end{array} \right).
\end{array}
$$

$$
\begin{array}{l}
a(finder, F) \ :: \\
\quad request\_suitable\_peers(Q) \leftarrow a(inquirer1(\_, \_, \_, \_), I) \ then \\
\quad \left(\begin{array}{l} suitable\_peers(Q, Ps) \Rightarrow a(inquirer1(\_, \_, \_, \_), I) \Leftarrow fetch\_suitable\_peers(Q, Ps) \\ or \\ suitable\_peers(Q, [\,]) \Rightarrow a(inquirer1(\_, \_, \_, \_), I) \end{array} \right).
\end{array}
$$

$$
\begin{array}{l}
a(responder, R) \ :: \\
\quad question(Q) \Leftarrow a(inquirer2(\_, \_, \_), I) \ then \\
\quad \left(\begin{array}{l} answer(A) \Rightarrow a(inquirer2(\_, \_, \_), I) \leftarrow answer(Q, A) \\ or \\ no\_answer \Rightarrow a(inquirer2(\_, \_, \_), I) \end{array} \right).
\end{array}
$$

Figure 5.5: eResponse scenario: the LCC specification of interaction model IM_2

The interaction model of Figure 5.4 specifies that a peer may take the role *route_finder* in order to find the path between two nodes, or locations. The peer may also include further restrictions on the path by specifying a specific vehicle type to be used for transportation and/or a list of undesirable roads. The interaction is simple and straightforward. The *route_finder* sends a *request_route* message to the *route_service* peer. After computing the route, the *route_service* sends back the route details, composed of a list of nodes, to the route finder. If the *route_service* fails in computing the route, it sends back an empty list.

The interaction model of Figure 5.5 specifies that an *inquirer* may initiate this interaction to ask other peers in the network some question. If the *inquirer* already knows the peers to be questioned, then it automatically jumps to role *inquirer*2. Otherwise, it takes the role *inquirer*1, in which it asks a list of *finder* peers about the appropriate peers that can answer its question. For instance, if the question is about the status of a given road, then the *inquirer* might need to take the role *inquirer*1 to ask some service about the peers in the vicinity of the road in question. After obtaining the list of peers to be questioned, the *inquirer* moves to play the role *inquirer*2. In the role *inquirer*2, the peer sends its question to the selected peers, one by one. If an answer is received from one, then the *inquirer* terminates its role successfully. However, if no result is received, then the question is sent to the next peer in the list, and so on. If all peers have failed to answer the question, then the final result is set to *nil*.

The *finder* peer has a straightforward task. After receiving a *request_suitable_peer* message for a specific question Q, it tries to find out which peers in the network are best candidates for answering such a question. It then sends the list of candidates back to the inquirer. If no suitable peers are found, then the *finder* peer sends back an empty list.

The peers being questioned play the role *responder* in IM_2. Their task is also straightforward. They first receive the question Q from *inquirer*2. If they have an answer A, they send it back to *inquirer*2. Otherwise, they reply with a *no_answer* message.

5.2.2 The Property Specification

After introducing the system model in the previous section, this section illustrates how the MCID model checker may be used to verify such a model. The MCID system is use in two different ways: (1) by the agents at run time to verify whether the interactions

they wish to join do not break any of their constraints, and (2) by the system engineers for verifying the correctness of the system they have built. In the first, agents jump from one interaction to the other, verifying each interaction online just before its execution. In the second, the global mesh of interactions for a given scenario is verified entirely offline. These two different applications of the verifier, along with the properties the system model is verified against, are presented in Sections 5.2.2.1 and 5.2.2.2, respectively.

5.2.2.1 Local Online Verification

In the local online verification method, the model checker is invoked locally by the agents in order to verify at run time whether the interaction they are about to join violates any of their deontic (or trust) constraints. In what follows, we present a sample of the properties the agents may wish to verify interaction IM_0 against.

- If some agent is invited to engage in an interaction for performing action X, then it should first check whether it is actually capable of performing X. For example, transporting one's neighbours from one location to another requires the peer to have access to some vehicle for making this transportation possible. Property 5.1 of Figure 5.6 specifies this deontic constraint in DPL in a straightforward manner.

- In a flooding scenario, it is normal for someone to be interested in saving itself and its family and friends first. Different people have different priorities for the actions they accept to perform. If someone is invited to engage in an interaction in which it is asked to perform action X, then it should first make sure that performing X does not waste its time allocated for performing more urgent actions. Property 5.2 of Figure 5.6 specifies this deontic constraint in DPL.

  The maximum amount of time the agent is willing to spend on actions in this interaction is specified as *Limit*, and obtained through the constraint $my\_time\_limit(Limit)$. The total amount of time that the agent will eventually spend on actions if it does engage in this interaction is specified as $Time$, and it is initially set to zero ($Time = 0$).

  For every action the peer might perform, the temporal property increases the value of $Time$ accordingly. This is done through the following sub-formulae:

The property, ensuring that an agent may accept to transport its neighbours in interaction IM_0 only if it has the appropriate vehicle to do so, is defined as:

$$\begin{aligned} &can(a(performer, civilian),\\ &\quad in(action(transport(neighbours(house\#23), node1, node2)), a(requester, \_)), +) \leftarrow\\ &\quad have\_vehicle(VehicleDetails) \wedge\\ &\quad capable\_of(VehicleDetails, transport(neighbours(house\#23), node1, node2)). \end{aligned} \tag{5.1}$$

The property, ensuring that an agent may accept action requests in interaction IM_0 only if it believes that performing such actions can be realised in an acceptable time frame, is defined as:

$$\begin{aligned} &can(a(performer, civilian), in(action(\_), a(requester, \_)), +) \leftarrow\\ &\quad my\_time\_limit(Limit) \wedge Time = 0 \wedge\\ &\quad \mu X.\ ([-]\mathtt{ff} \wedge Time \leq Limit) \vee\\ &\qquad (\langle - \rangle \mathtt{tt} \wedge\\ &\qquad [\#(route\_details(\_, \_, \_, T), a(performer, civilian))](Time = Time + T \wedge X) \wedge\\ &\qquad [-\#(route\_details(\_, \_, \_, \_), a(performer, civilian))]X). \end{aligned} \tag{5.2}$$

The property, ensuring that a result will be received for every requested action in interaction IM_0, is defined as:

$$\begin{aligned} &trust(interaction(IM_0), +) \leftarrow\\ &\quad \nu X.\ [-]X \wedge\\ &\qquad [out(action(Action), a(\_, Id))]\mu Y.\ (\langle - \rangle \mathtt{tt} \wedge [-in(result(Action, \_), a(\_, Id))]Y) \end{aligned} \tag{5.3}$$

The property, ensuring that the peer asking other peers for information will try to collect more than one answer in interaction IM_2, is defined as:

$$\begin{aligned} &trust(interaction(IM_2), +) \leftarrow\\ &\quad \mu X.\ [-in(answer(\_), a(\_, \_))]X \wedge\\ &\qquad [in(answer(\_), a(\_, \_))]\mu Y.\ (\langle - \rangle \mathtt{tt} \wedge\\ &\qquad\qquad [-\{\#(empty(\_), a(\_, \_)), out(question(\_), a(\_, \_))\}]Y) \end{aligned} \tag{5.4}$$

Figure 5.6: eResponse scenario: the properties to be verified online

$$\langle - \rangle \texttt{tt} \wedge$$
$$[\#(route\_details(\_, \_, \_, T), a(performer, civilian))]\ (Time = Time + T\ \wedge\ X) \wedge$$
$$[-\#(route\_details(\_, \_, \_, \_), a(performer, civilian))]\ X$$

which states that an action can occur ($\langle - \rangle \texttt{tt}$), and for every move action that requires time T to be performed ($[\#(route\_details(\_, \_, \_, T), a(performer, civilian))]$), $Time$ is increased by T and the property is said to be satisfied at the next state(s) of the state-space ($Time = Time + T\ \wedge\ X$). If no move action occurs, then $Time$ is not increased and the property is said to be satisfied at the next state(s) of the state-space ($[-\#(route\_details(\_, \_, \_, \_), a(performer, civilian))]X$).

When no more actions can be performed ($[-]\texttt{ff}$), the temporal property verifies that the total amount of time spent on performing actions is less than or equal to the time dedicated for performing these actions ($Time \leq Limit$).

- ✦ The agent that needs to invite others to perform certain actions may decide that the interaction model may only be accepted if it is guaranteed that agents will always send back the results of their actions. Property 5.3 of Figure 5.6 specifies this temporal constraint in the μ-calculus. The property states that the interaction model is trusted if every time an $action(\_)$ message is sent out ($[out(action(Action), a(\_, Id))]$) then it is always the case that a $result(Action, \_)$ message will be received from the appropriate peer ($\mu Y.\ \langle - \rangle \texttt{tt} \wedge [-in(result(Action, \_), a(\_, Id))]Y$ [2]). Finally, the first part of the temporal property, $\nu X.\ [-]X$, states that this property should always be satisfied at every state of the interaction's state-space.

Only after successfully verifying the satisfaction of Properties 5.1, 5.2, and 5.3 is interaction IM_0 executed. During the execution, one of the performer agents will eventually reach a state where it has to satisfy the LCC constraint: $realise\_goal(get\_path(...))$. The agent searches the list of interaction models for an appropriate one that would help it solve the above constraint. We assume it selects IM_2 first. Again, before executing

[2] The μ-calculus property $\mu X.\ \langle - \rangle \texttt{tt} \wedge [-A]X$ states that something can happen ($\langle - \rangle \texttt{tt}$) and for every action other than the action A, the same property should hold again ($[-A]X$). Finally, the least fixed point operator μX ensures that the recursion should eventually terminate. Basically, this implies that it is always the case that the action A will eventually occur. Temporal formulae of this form are used in this chapter by Properties 5.3, 5.4, 5.5, 5.7, and 5.9.

IM_2, the agent will need to verify that this interaction satisfies certain properties. For example:

- If the interaction allows the agent to ask other agents some question, then the agent might be interested to know whether the interaction allows it to collect all replies or not. For instance, in some cases, the agent might have a list of a hundred friends to ask. However, it might be in a hurry, and would be satisfied with the first reply it receives. In other cases, the agent might be interested in collecting all replies and selecting the most suitable one itself. Property 5.4 of Figure 5.6 provides the μ-calculus specification of the latter temporal property. It states that every time an answer is received ($[in(answer(\_), a(\_, \_))]$) either the question is sent out again to another peer or all peers have already been questioned and the list of peers has become an empty list ($\mu Y.\ \langle - \rangle \mathtt{tt} \wedge [-\{\#(empty(\_), a(\_, \_)), out(question(\_), a(\_, \_))\}]Y^{\ 2}$). Recursion is specified by $\mu X.\ [-in(answer(\_), a(\_, \_))]X$, which states that for every action that differs from that of receiving an answer, the property should be satisfied again at the next state(s) of the interaction's state-space.

5.2.2.2 Global Offline Verification

The MCID system may also be used offline by system engineers to verify the correctness their system. For example

- The system engineer might need to verify that every action requested from a peer will eventually be performed by that peer. Property 5.5 of Figure 5.7 provides the μ-calculus specification of this temporal property. It states that for every 'move action' request that is sent out ($[out(action(move(N)), a(\_, \_))]$), it is always the case that the agent will eventually perform the move ($\mu Y.(\langle - \rangle \mathtt{tt} \wedge [-\#(perform(move(N), \_), a(\_, \_))]Y)^{\ 2}$). Similarly, for every 'transport action' request that is sent out ($[out(action(transport(O, \_, N2)),\ a(\_, \_))]$), it is always the case that the object will eventually be dropped at the right location ($\mu Z.\ \langle - \rangle \mathtt{tt} \wedge\ [-\#(perform(drop(O, N2), \_), a(\_, \_))]Z^{\ 2}$). Finally, the first part of this property, $\nu X.\ [-]X$, states that the property should hold at all states of the interaction's state-space.

While traversing the state-space of interaction IM_0 trying to prove the satisfaction of Property 5.5, the verifier will eventually hit the constraint $realise\_goal(get\_path(...))$.

The property, ensuring all actions are eventually performed in interaction IM_0, is defined as:

$$\begin{array}{l} \nu X.\ [-]X \wedge \\ \quad [out(action(move(N)), a(\_, \_,))]\mu Y.\ (\langle - \rangle \mathtt{tt} \wedge [-\#(perform(move(N), \_), a(\_, \_))]Y) \wedge \\ \quad [out(action(transport(O, \_, N2)), a(\_, \_))]\mu Z.\ (\langle - \rangle \mathtt{tt} \wedge \\ \qquad\qquad [-\#(perform(drop(O, N2), \_), a(\_, \_))]Z) \end{array} \tag{5.5}$$

The property, ensuring that interaction IM_1 will terminate for the peer that initiated it, is defined as:

$$\mu X.\ terminates(a(route\_finder(\_, \_, \_, \_, \_), \_)) \ \vee \ (\langle - \rangle \mathtt{tt} \ \wedge \ [-]X) \tag{5.6}$$

The property, ensuring that the goal is achieved in interaction IM_1 by making sure route information is sent back to the peer that requested it, is defined as:

$$\mu X.\ \langle - \rangle \mathtt{tt} \ \wedge \ [-in(route(\_), a(route\_service, \_))]X \tag{5.7}$$

The property, ensuring that interaction IM_2 will terminate for the peer that initiated it, is defined as:

$$\mu X.\ terminates(a(inquirer(\_, \_), \_)) \ \vee \ (\langle - \rangle \mathtt{tt} \ \wedge \ [-]X) \tag{5.8}$$

The property, ensuring that the goal is achieved in interaction IM_2 by making sure an answer is sent back to the peer that asked the question, is defined as:

$$\mu X.\ \langle - \rangle \mathtt{tt} \ \wedge \ [-in(answer(\_), a(responder, \_))]X \tag{5.9}$$

Figure 5.7: eResponse scenario: the properties to be verified offline

This constraint should be satisfied by the peer for it to proceed with its attempt in performing the requested action. To verify whether the LCC constraint will be satisfied by the peer, the MCID model checker has been modified as follows. Every time a constraint of the form *realise_goal*(*Goal*) is encountered, the model checker assumes that the constraint is satisfied in one of the following two cases: (1) if the peer's knowledge base contains appropriate inference tools for dealing with such a constraint, or (2) if there exists an interaction model that guarantees the satisfaction of such a constraint. In the first case, our model checker does not attempt to solve the actual constraint by making use of the peer's knowledge base. It is sufficient for the model checker to know that the peer's knowledge base contains the appropriate Prolog facts and/or rules for address-

ing such a constraint. In the second case, an interaction model is believed to satisfy such a constraint only if the peer initiating this interaction is guaranteed to complete its role successfully and realise the goal it was expected to fulfil. For example, to verify whether interaction IM_1 is sufficient for realising the goal $realise\_goal(get\_path(...))$, the following two properties should hold:

- The interaction model should guarantee that a *route* message will eventually be received from the peer playing the *route_service* role. Property 5.7 of Figure 5.7 provides the μ-calculus specification of this temporal property [2] .

- The goal of initiating interaction IM_1 is to obtain the details of the route connecting two nodes so that this information may be used in the main interaction model IM_0. However, the route details, even if obtained by the peer in interaction IM_1, are not returned to interaction IM_0 unless the peer completes its role successfully in interaction IM_1. Hence, there is a need to ensure the successful termination of interactions, or at least the successful termination of the peer in question. Property 5.6 of Figure 5.7 provides the μ-calculus specification of this temporal property. It states that either the route finder's role has terminated ($terminates(a(route\_finder(\_,\_,\_,\_,\_),\_))$), or something can happen and for everything that happens the property should be satisfied again ($\langle - \rangle \mathtt{tt} \ \wedge \ [-]X$). The least fixed point operator μX ensures that eventually the recursion should be broken, implying that the termination of the route finder's role is guaranteed.

Similar to Properties 5.6 and 5.7, which are verified against interaction IM_1, Properties 5.8 and 5.9 are to be verified against interaction IM_2. While Property 5.8 specifies the successful termination of the role $inquirer(Question, Answer)$, Property 5.9 guarantees that an answer to the *inquirer*'s question is eventually received.

5.3 Results

Two different ways of applying the MCID model checker have been introduced: local online verification versus global offline verification. Sections 5.2.2.1 and 5.2.2.2 have presented the various scenarios and properties to be verified in each of these application methods. The verification results are presented in the remainder of this section.

5.3.1 Local Online Verification

In our online verification example, the agents are expected to verify their deontic constraints before engaging in an interaction. For example, before interaction IM_0 is executed, it is verified against Properties 5.1, 5.2, and 5.3. Only after the MCID system proves the satisfaction of all three properties in interaction IM_0, the interaction is executed. During execution, the performer agents will eventually have to satisfy the LCC constraint *realise_goal*(*get_path*(...)). We assume one of the performer agents does not know the path it should take, and needs to engage in some other interaction for obtaining this information. The execution of interaction IM_0 is paused and interaction IM_2 is obtained and verified against Property 5.4. The satisfaction fails and the agent searches for some other interaction model. Interaction IM_1 is obtained and executed. After the successful completion of interaction IM_1, the agent marks the LCC constraint *realise_goal*(*get_path*(...)) of interaction IM_0 as satisfied and the execution of interaction IM_0 is resumed.

The CPU time and the memory usage for verifying each of these properties, from the moment the model checker is invoked until the results are returned to the peer, are presented in Table 5.1.

Property #	**Verified against**	**CPU time (in *sec*)**	**Memory usage (in MB)**
Property 5.1	IM_0	0.000	0.001
Property 5.2		1.608	91.884
Property 5.3		1.772	169.097
Property 5.4	IM_2	0.004	0.181

Table 5.1: eResponse scenario: results of online verification

Note that these properties are verified at run time. This implies that the execution of interactions is sometimes paused, not only for the agent performing the verification but also for other agents interacting with the verifying agent and waiting for its input. Verification time is therefore crucial. Fortunately, our results prove that online verification via the MCID system is indeed possible and realistic, since the time peers spend waiting for verification results is acceptable. In our worst scenario, verification completes in less than $2sec$. As for the memory usage, the results show that verification could consume a few hundred bytes, going up to 170MB in our worst scenario. Nevertheless, even the worst case scenario is still considered to be acceptable for many agents. If

it happens that some minimalist agent needs quick verification results yet lacks the memory requirements for performing the verification itself, then a web service fitted with the MCID verification mechanism may be consulted.

5.3.2 Global Offline Verification

One of the properties any system engineer is interested to verify is whether interactions fulfil their goals. In our flooding eResponse scenario, the system engineer needs to verify whether interaction IM_0 allows all peers to eventually perform the actions requested from them. This is a temporal property of the interaction that is specified in the μ-calculus as Property 5.5.

In our offline verification example, the MCID system is invoked offline to verify Property 5.5 against interaction IM_0. The CPU time and the memory usage for verifying this property, from the moment the model checker is invoked until the results are returned to the peer, are presented in Table 5.2.

Property #	**Verified against**	**CPU time (in *sec*)**	**Memory usage (in MB)**
Property 5.5	IM_0	1.680	163.895
Property 5.8	IM_2	0.200	1.681
Property 5.9		0.000	0.044
Property 5.6	IM_1	0.004	0.098
Property 5.7		0.000	0.041

Table 5.2: eResponse scenario: results of offline verification

As opposed to local online verification, global offline verification is not driven by the execution of interactions. Nevertheless, the properties verified against one interaction model may still depend on other interaction models. For example, while traversing the state-space of interaction IM_0 trying to prove the satisfaction of Property 5.5, the model checker will eventually hit the constraint $realise\_goal(get\_path(...))$. The model checker should be able to verify whether this constraint will be satisfied or not. To do this, it checks whether there exists an interaction model that the peer may invoke and that guarantees the fulfilment of the goal $get\_path(...)$. In our running scenario, we assume that only two interactions exist that may be used by agents to fulfil the $get\_path(...)$ goal. These are interactions IM_1 and IM_2 of Figures 5.4 and 5.5, respectively. The model checker needs to verify that there exists at least one interaction

model in which the goal is guaranteed to be realised and the interaction is guaranteed to terminate for the initiating peer. We assume IM_2 is fetched before IM_1. The model checker automatically verifies IM_2 against these two properties, Properties 5.8 and 5.9. Property 5.8 succeeds, since the interaction is guaranteed to eventually terminate for the role *inquirer*(_, _). Property 5.9 fails, since an answer is not guaranteed to be delivered. Therefore, the model checker moves on to look for other interaction models. IM_1 is retrieved and verified against Properties 5.6 and 5.7. Both of these properties are satisfied; therefore, the model checker concludes that the LCC constraint *realise_goal*(*get_path*(...)) of interaction IM_0 can be satisfied by agents at run time. Proving the satisfaction of Property 5.5 in interaction IM_0 is then resumed by the model checker.

As a result, the verification time and the memory usage for checking Property 5.5 in IM_0 includes the verification time and the memory usage for both checking Properties 5.8 and 5.9 against IM_2 and checking Properties 5.6 and 5.7 against IM_1.

Finally, we note that although this chapter presents the global verification technique as a technique to be performed offline by the engineers, the results proves that some global verification techniques may also be performed by the agents if they wish to verify global properties spanning several interaction models.

5.4 Conclusion

After introducing the MCID system in Chapter 3 and its possible application to the field of trust in Chapter 4, this chapter illustrates the possible use of the MCID system in two different ways for verifying an OpenKnowledge eResponse flooding scenario. In the first, interaction time verification is demonstrated, illustrating how agents may pause the execution of one interaction in order to verify and execute another interaction, before going back to fulfilling the previous interaction. Compared to the scenarios of previous chapters, agents in this new scenario can verify and execute interactions from within other interaction models. In the second method, the model checker is invoked offline to verify properties of complex systems composed of a mesh of interconnected interaction models, as opposed to properties of a single interaction model. This implies that the verification results of one interaction may depend on the verification results of others in this mesh of interactions.

A lightweight model checker has resulted in remarkably efficient results in both verification methods, as presented by Tables 5.1 and 5.2. However, the success of

our verification mechanism lies in knowing the minimum number of agents needed to play each role for every property that is verified. For example, interaction IM_1 is verified while having one agent play the role *route_finder* and another play the *route_service* role. On the other hand, interaction IM_2 is verified while having one agent play the *inquirer* role and two agents for each of the *finder* and *responder* roles. We note that knowing the minimum number of agents required for verifying certain properties of a given interaction protocol is not always trivial and may require additional proofs. At the time being, the properties are specified by hand. Hence, the same person specifying the property may specify the minimum number of agents needed to play each role when verifying that property. We believe it will be hard for agents to automatically specify properties from scratch. However, in the future, we hope agents will be provided with a collection of properties that they may choose from whenever needed. Information, such as the minimum number of agents needed to play each role, may easily be provided with the property, possibly with a proof for the suggested number of agents. Agents can then modify such properties as needed to suit their current interaction model, as discussed shortly. Nevertheless, this raises the question: *"How does the application of one property to a different interaction model affect the number of agents needed?"* Inductive reasoning could provide one solution for this problem. However, for the time being, we leave this issue for future work.

The complication of the automatic specification process of properties has been raised earlier in the previous chapter. In what follows, we revisit this issue by providing more concrete examples from the eResponse scenario. For instance, when the model checker hits any LCC constraint of the form $realise\_goal(G)$, the model checker might end up searching for other interactions that might lead to the fulfilment of such a constraint. Every interaction obtained should be verified against two general properties: (1) the guaranteed termination of the role played by the peer in question, and (2) the guaranteed realisation of the goal G. The first is specified in the μ-calculus as $\mu X.\ terminates(a(my\_role, my\_id)) \vee (\langle - \rangle \mathtt{tt} \wedge [-]X)$, where *my_role* is the role played by the agent that needs to realise the goal G and *my_id* is the agent's unique identifier. Properties 5.6 and 5.8 present different instantiations of this property. In these instantiations, *my_role* is simply replaced with either $a(route\_finder(\_, \_, \_, \_, \_), \_)$ or $a(inquirer(\_, \_), \_)$. These are the initiating roles of interactions IM_1 and IM_2, respectively. Before engaging in an interaction, the peer already knows the role it will be playing. Therefore, the automated specification of such a property for a specific interaction model may be performed in a straightforward manner. However, the second

property, which guarantees that goal G is fulfilled, is a bit trickier. The property in its general form is specified in the μ-calculus as $\mu X.\ \langle - \rangle \mathtt{tt} \wedge [-Action]X$, where $Action$ is the action responsible for fulfilling the goal G. For instance, Property 5.7 replaces $Action$ with $in(route(\_), a(route\_service, \_))$, since the receipt of a $route(\_)$ message from the route service peer signals the fulfilment of the goal $get\_path(...)$ in interaction IM_1. In interaction IM_2, the fulfilment of the same goal is signalled by the receipt of an $answer(\_)$ message from some responder peer, as illustrated by Property 5.9. Spotting the right action is tricky. For peers to be able to specify such properties in an automated way, the peers will need further tools that would allow them to read and understand interactions. For instance, one way of achieving this is by spotting the variable in question (that is the variable relating to the goal G) and the actions instantiating such a variable. We currently leave this complex issue for future work.

Chapter 6

Literature Review

After presenting our proposed verification mechanism in Chapter 3, its possible application to the field of trust in Chapter 4, and some results on verifying more realistic scenarios in Chapter 5, this chapter provides an overview to the various verification mechanisms in the field of multiagent systems.

In the early 1980's, model checking emerged as a tool for verifying hardware systems. Systems were modelled via a process calculus which relies on messages passing through channels as a way to represent the interaction between the various processes. After its success as an automatic verification technique for hardware systems, model checking was then used, since the late 1990s, for verifying software models such as communication protocols, distributed algorithms, etc. Again, messages and channels were sufficient to model various software systems such as the Needham-Schroeder protocol, the Real-Time Ethernet protocol, etc. Since the late nineties and early noughties, the agent community has become interested in adopting model checking techniques for verifying multiagent systems. Multiagent systems, however, are a collection of autonomous agents rather than a set of predefined processes. Process calculus, as we know it, seems no longer sufficient for modelling such systems. To accommodate agents' autonomy in such systems, various logics (and concepts) have been applied to multiagent system specification, such as epistemic logic (logic of knowledge, uncertainty and ignorance), doxastic logic (logic of belief and disbelief), alethic logic (logic of necessity, possibility and contingency), deontic logic (logic of obligation and permission), and temporal logic (logic of time). This triggers the question of how to specify, or model, multiagent systems? This question, as this chapter illustrates, is crucial since it has a huge impact on the verification process.

The chapter opens with Section 6.1, which presents the literature's different multi-

agent verification techniques, focusing on those most related to our line of work. This is followed by Section 6.2, which analyses and synthesises those techniques in comparison with that presented in this thesis. The conclusions are drawn in Section 6.3.

6.1 Verification Mechanisms

Generally speaking, there are two main elements to consider when formally verifying any system: the system to be verified and the property it should be verified against. These constitute the inputs of model checkers, the predominant verification technique in the field of multiagent systems. Therefore, we choose to categorise the various verification mechanisms of the literature based to these two elements. As a result, our categorisation stems from our view of multiagent system models, in which the model is split into two layers: the agents layer and the interaction one (Section 3.1).

Table 6.1 divides multiagent verification mechanisms into nine different categories. The main three categories are presented by the diagonal of Table 6.1. In the first category (the top left cell of Table 6.1), the model verified is a purely interaction model that does not take into consideration any of the agents involved in such interactions. As a result, the type of properties that may be verified in such a model are properties of the interaction. These are usually safety, liveness, or fairness properties. Examples of verifiers in this category are presented in Section 6.1.1. In the second category (the middle cell of Table 6.1), the model verified is a purely agent model, usually specified through the agents' beliefs, desires, and intentions. The type of properties that may be verified in such categories are those relating to agents' mental states. These are usually concerned with agents' knowledge, beliefs, etc. Examples of verifiers in this category are presented in Section 6.1.2. In the third category (the bottom right cell of Table 6.1), the model verified is a combination of interaction and agents models. As a result, the type of properties that may be verified in such a model are properties of both the interactions and agents. Examples of verifiers in this category are presented in Section 6.1.3. Of course, if a verifier can verify properties of both interactions and agents, then it automatically falls into the other two categories of verifying either properties of interactions or properties of agents. Therefore, the mechanisms of Section 6.1.3 are extended to the remaining two cells of the bottom row of Table 6.1.

An interesting point to note is that the verification mechanisms that verify agent specifications are also capable of verifying properties of interactions, as illustrated by the middle row of Table 6.1. This is because these mechanisms usually rely on

Model Verified	Verifying Properties of		
	Interactions	**Agents**	**Interactions & Agents**
Interaction Model	Sections 6.1.1 & 6.1.4	Section 6.1.4	Section 6.1.4
Agents Models	Section 6.1.2	Section 6.1.2	Section 6.1.2
Interaction & Agents Models	Section 6.1.3	Section 6.1.3	Section 6.1.3

Table 6.1: Verification mechanisms of multiagent systems

combining all the agent models into one huge system model in the style a of finite state machine before feeding it to the model checker. Naturally, this allows the verification of temporal properties of the system.

The more unusual mechanisms are those verifying properties of both agents and interactions in interaction models that do not take the agents specification into consideration. These mechanisms are presented in Section 6.1.4 and occupy the cells of the first row of Table 6.1. For example, they could verify properties about the evolution of knowledge of some agent in a given interaction without the need for the agent's specification. This is usually achieved by explicitly specifying the propositions that may hold in a state and the rules under which they hold, offering a way to label states with propositions.

With the currently available literature, the nine categories of Table 6.1 may be cut down to only four categories: (1) mechanisms that verify interaction properties of interaction models, (2) mechanisms that verify agents models, (3) mechanisms that verify properties of a mesh of interaction and agents models, and (4) mechanisms that verify both interaction and agent properties of interaction models. The mechanisms of each of these four categories are presented in detail in the following four sections, respectively.

6.1.1 Model Checking Interaction Models

Several attempts have been made in order verify the interaction layer of multiagent systems. This section briefly introduces some of these techniques.

Wen and Mizoguchi (1999) make use of the SMV model checker (McMillan, 1993) to verify a couple of multiagent scenarios. The SMV system is a freely available model checker that verifies CTL temporal properties of finite Kripke structures. One of the scenarios verified is a producer-consumer scenario, where two consumers would request a certain product from a producer (each under different circumstances) and the

producer would deliver each requested item. The other scenario verified is a similar, yet more complicated, scenario where requests may be cancelled, suspended, accepted, declined, etc. The properties verified in both scenarios described liveness and safety properties. For example, properties describe whether the producer will eventually deliver the product every single time it is requested by a consumer, the producer will deliver products to both consumers infinitely often, the consumer would complete its role in the interaction while the producer does not, the consumer will eventually either cancel or resume a previously suspended request, etc.

The translation of the system model into the language of SMV is done by hand and the task is a relatively straightforward task. Nevertheless, the model checker is very efficient and properties are verified in a fraction of a second.

Walton (2004) addresses the same problem using different languages and an automated translation process. In his method, the system to be verified is specified in the MAP language, a process calculus. In MAP, an interaction is defined by a set of roles where each role has its own method definition. Methods, also known as agent protocols, are defined by the actions an agent can take. These actions could be a decision action, a message output action, and a message input action. Protocols can get more complex by making use of the sequence, choice, parallel, iteration, and recursion operators. For performing the verification, the SPIN model checker (Holzmann, 2003) is used. It verifies whether LTL properties are satisfied in a system model specified in Promela, the language of SPIN. However, instead of translating the system model from MAP into Promela by hand, an automatic translator is created for performing this job. The properties verified by Walton (2004) are traditional safety properties, such as those describing the successful termination of programs.

Similar to Walton (2004), Huget (2002) also uses the SPIN model checker for the verification of multiagent interactions. However, in his approach, interactions are specified via the Agent UML language. Agent UML allows the graphical specification of interactions by defining the agents' roles, their message passing actions, the constraints on messages, and how the protocol may be executed (i.e. the state graph of the protocol). The textual version of an Agent UML diagram is specified in an XML-based language. This is then translated into Promela and the result is fed to SPIN along with the LTL properties to be verified. The property verified by Huget (2002) is a reachability property stating that all the states of a given interaction are reachable.

Cliffe and Padget (2002) present a similar mechanism for the verification of electronic institutions (Esteva et al., 2001). In their approach, the model checker used is

NuSMV (Cimatti et al., 1999), a variant of SMV. Similar to the two approaches presented above, the interactions are defined through the specification of generic agent roles, along with a dialogical framework, scenes, performative structure, and norms. The general idea behind electronic institutions is that acceptable actions are defined for general agent roles. These roles may be instantiated at run time by various agents. Their specifications compose the dialogical framework. Furthermore, one scene may be composed of more than one dialogical framework and agents may jump between scenes following specific rules. The performative structures are used to specify the connections between scenes. Finally, the norms are used to specify the general rules of what is permitted, prohibited, or obligatory.

A translator is used to convert the specification of an electronic institution into the input language of NuSMV. The output of the translator is fed to the model checker along with the properties that need to be verified. Again, verification turns out to be efficient. For instance, the verification process of a fish market auction scenario against properties expressing fairness and successful termination was completed in a fraction of a second.

All of the mechanisms presented in this section relied on existing model checkers, such as the SMV, NuSMV, and SPIN systems, for the verification of multiagent interactions. The system model in each case was translated into the model checker's language either automatically (e.g. Walton (2004), Huget (2002), and Cliffe and Padget (2002)) or by hand (e.g. Wen and Mizoguchi (1999)). As expected, the results (when mentioned) were all in the range of fractions of a second.

Despite their efficiency, all techniques in this category share one major drawback. They view multiagent interactions to be just another software program whose outcome is predicted by solely analysing the program's specification. They do not take into consideration the autonomous agents that in reality are driving these interactions and hugely affecting their outcome by directing them as they please. The mechanisms described in the following section take completely the opposite approach by considering only the agents model.

6.1.2 Model Checking Agents' Metal States

Many researchers have showed more interest in verifying properties of the agents' mental states than traditional temporal properties of the interaction. For example, instead of verifying whether a seller will eventually deliver an item, one may verify that

if the buyer believes the seller will eventually deliver the item, then the item will be delivered.

Benerecetti et al. (1998) present a mechanism that allows the verification of an agent's mental state by treating belief atoms as propositional atoms. Therefore, they can be verified like any other propositional atom in traditional model checking techniques. In this mechanism, multiagent systems are specified through the MATL temporal logic. MATL extends CTL, the branching time propositional temporal logic, by the HML logic. HML introduces a formalisation for BDI attitudes by allowing the specification of agents' beliefs, desires, and intentions. The main idea is that the notion of agents is built on top of the traditional notion of processes. Each agent has its own beliefs, desires, and intentions. The system is composed of a collection of views. An agent may have at most three views that are used to express its beliefs (about itself and/or others), desires, and intentions. In addition to the agent views, there is also a need for a global view on a higher level that describes the view of an external observer. For example, an agent may have false beliefs about others and a global view becomes necessary for specifying the behaviour of the entire system.

MATL system models are translated into a multiagent finite state machine (MAFSM) before they are fed to the model checker for verification. The model checker's algorithm is similar to the CTL labelling algorithm (Edmund M. Clarke et al., 1999). However, it is modified to take into account the data structures of the MALT language. In summary, the model checker traverses the system's state-space, labelling states that satisfy the sub-formulae of the property to be verified. After inspecting all the states, the property is said to be satisfied in the system only if the initial states of the system have been labelled with this property. Benerecetti and Giunchiglia (2001) investigate the possible application of this mechanism to various multiagent system scenarios, focusing on security protocols.

Wooldridge et al. (2002) introduce a different approach for the verification of agent mental states. In their approach, the MABLE language is used for specifying and verifying multiagent systems. The system is specified through the specification of the various agents and their *mental states*, or their BDI logic. In addition to the specification of agents, MABLE allows the specification of *claims*. Claims describe the properties that one is interested in verifying.

Both the agent models and the claims to be verified are specified in a simplified version of $\mathcal{LORA}$. $\mathcal{LORA}$ (Wooldridge, 2000) is a branching temporal logic that is extended with both modal connectives specifying beliefs, desires, and intentions as

well as concepts from dynamic logic for the specification of agents and the actions they may perform. The logic contains operators that describe properties of states as well as properties of paths. For example, the belief, desire, and intention operators describe properties of states and one can specify whether a given belief holds or not in a given state. On the other hand, traditional modal and temporal operators specify properties of paths. For example, *Happens* α specifies that action α happens next.

Agent specifications make use of *do* instructions, *while* iterations, *if-then* selections, etc. For performing the verification, MABLE uses the SPIN model checker. The MABLE compiler takes as input a MABLE system specification and its associated claims. The compiler translates claims into LTL properties and the agents' specifications into PROMELA. The output of the compiler is fed to the SPIN model checker for verification. An interesting example of the properties than can be verified states that "some agent *i* eventually comes to believe that *agent*1 intends that *i* believes *a* has the value 10". Clearly, verifying such properties seems to be a limitation of our current MCID model checker. This issue is investigated in more detail in Section 6.2.

An unconventional application to MABLE is the verification of electronic institutions (Esteva et al., 2001), as illustrated by Huguet et al. (2002). For specifying multiagent systems, electronic institutions focus on the general organisational approaches, or the social norms. These represent the rules of the game in the society. Instead of modelling agents, the focus is on specifying a dialogical framework, scenes, performative structure, and norms. As a result, MABLE should be used in such a way that it focuses on the specification of *societal* aspects as opposed to agents *internal* aspects. The trick is to define the system through the specification of each agent *role* as a traditional MABLE agent. A translator is created to automatically perform this translation.

The result of the translation is appended to a set of basic MABLE claims resulting in the complete MABLE specification. This specification is then fed to the MABLE compiler for translating the claims into LTL properties and the agent specifications into Promela. The results of this second translation are fed to the SPIN model checker for verification. Unfortunately, Huguet et al. (2002) do not elaborate on the drawback of translating the system twice, where each translator is performing a translation between two contrasting models.

An example of the properties verified are those specifying that it is possible to exit the system "cleanly" from any state in the system, or that whenever the system is in state X, it is guaranteed that it will eventually reach state Y. We note that this unconventional application of MABLE focuses on verifying properties of the interaction,

rather than properties of agents' mental states. This makes this particular application of MABLE seem closer to the mechanisms of Section 6.1.1. Nevertheless, in practise, these interactions are specified through agent models.

Along the same lines as model checking multiagent systems with MABLE, model checking approaches have been applied in a fairly similar manner to multiagent systems specified via AgentSpeak. The main difference being that AgentSpeak is a logic-based language, while MABLE is an imperative language.

Bordini et al. (2003a) introduce the toolkit CASP for checking AgentSpeak programs. The toolkit provides an automatic translation from AgentSpeak to the input languages of existing model checkers. Two different model checkers have been tested: SPIN and the JPF2 general purpose Java model checker.

The research carried out on verifying AgentSpeak via the SPIN model checker is presented in the published paper by Bordini et al. (2003b). Similar to MABLE, AgentSpeak specifies multiagent systems through the specification of the various agents in the system. In summary, an agent is created by specifying its set of base beliefs and its set of plans. A belief atom is simply a first order predicate. A plan, however, is composed of a triggering event, a set of belief literals, and a set of goals. A triggering event specifies the event that triggers the plan. It could be the addition or deletion of mental attitudes (i.e. beliefs or goals). The belief literals are used to specify the conditions for executing the goals. Only if the belief literals are true is the specified goal executed. Goals could either require the execution of actions, such as sending messages, printing some data, etc., the fulfilment of certain goals, the testing of certain believes, or even the addition/deletion of beliefs. Finally, intentions are viewed as plans that agents have committed to. In practise, they are specified as partially instantiated plans.

To be able to perform the verification in SPIN, AgentSpeak(F) (a variant of AgentSpeak that is used for restricting the system to finite state systems) should be translated to Promela. The translation turns out to be a rather complicated process. This is because Promela allows the specification of an interaction's state-space and not the specification of belief predicates and plans. Therefore, a slight modification to the use of Promela's channels is introduced. Channels are typically used in process calculus to connect the various processes together. However, in multiagent systems, all the processes/agents should be connected to each other. The sender and receiver of a certain message is specified in the message itself and does not rely on a channel to direct it. Therefore, only one channel m is used for inter-agent communication. All messages are delivered to and collected from that channel. Other channels are then used to store

the various data structures of AgentSpeak. For instance, channel *b* is used to store the agent's beliefs. Channel *e* is used to store events. Channel *i* is used for scheduling intentions. And so on. For further technical details on the translation of AgentSpeak to Promela and the use of Promela channels in this process, we refer the interested reader to the published paper by Bordini et al. (2003b).

The research carried out by Bordini et al. (2003c) translates AgentSpeak to the input language of JPF2, a general purpose Java model checker. The translation to a Java model turns out to be much more elegant and straightforward than the translation to Promela. For example, the Java model may use objects and their instances. This makes the specification of plans and their instances, or intentions, a simple task. Furthermore, having a plan library is easy in the Java mode and cumbersome in the Promela model. Last, but not least, the use of JPF2 sounds more appealing simply because Java is the language used in the implementation of many multiagent systems.

However, the true drawback of using JPF2 is its efficiency. Using SPIN to verify AgentSpeak was already seen to be demanding in terms of memory consumption and processing time. Using JPF2 turned out to be even more demanding. A comparison between the two techniques was carried out on the "Mars scenario" (Bordini et al., 2003c), where one robot is responsible for collecting garbage and sending it to another robot that places the garbage it receives in an incinerator. One of the properties verified states that it is possible for the first robot to have the intention of continuing to check for garbage and to believe that it is currently checking the slots for garbage. The verification of this property took 65.78*sec* to complete and consumed 210.51MB of memory when verified by SPIN, while it took 18:49:16 hours to complete and consumed 366.68MB of memory when verified by JPF2. In another setting, with the garbage placed at different locations, the same property took 5.25*sec* to complete in SPIN and 76.63*sec* in JPF2.

As a result, the authors felt the need to investigate further techniques for improving efficiency. In their following research, presented by Bordini et al. (2004), a slicing mechanism for reducing the state-space is investigated. Slicing has originally been used on logic programs (Zhao et al., 1994). It also has been used for a long time in imperative programs (e.g. Berzins, 1995). The idea behind slicing is to eliminate chunks of the system that are not relevant and do not affect the verification of the property in question.

To perform this slicing mechanism, an environment should be specified in addition to the traditional AgentSpeak agents. The environment is used in the slicing mecha-

nism to link the plans of different agents together. It is represented as a set of rules. The head of each rule is either an action or an empty head. The body specifies the possible belief changes resulting in performing the action specified in the head.

In the first step of slicing, all dependencies are specified by linking the various goals in each agent plan to the matching triggering event of the agent's remaining plans. The head of each of the environment rules is linked to the matching action in the various agent plans. Each element in the body of an environment rule is linked to its match in the agent plans. In the second step of slicing, an algorithm takes in the AgentSpeak agent programs, the environment specification, the specified links generated from the previous step, and the property to be verified. The property is broken down into atomic sub-properties. For each of these sub-properties, the agents' plans are traversed to see which nodes are reachable, using the environment as a link between the plans of different agents. Finally, all the plans that have not been marked in the previous step are deleted. If one of the agents had all its plans deleted, then the entire agent may be removed from the system.

In one example, slicing achieved a 25.6% reduction in time and 33% in memory usage when using SPIN. In another example, slicing achieved a 26% reduction in time and 21% in memory usage. However, using SPIN's built-in slicing algorithms did not result in any reduction in the state-space.

All the mechanisms presented in this section have focused on the verification of agent models. This might be useful in closed systems were the user performing the verification has access to the agent specification. In open systems, however, this is not feasible for several reasons, as discussed in Section 3.3.1. For example, agents are usually weary of providing access to their internal specification. Moreover, even if we assume that this does happen, agents are expected to be modelled in various ways and the required information may not be expressed declaratively. In closed systems, one may argue that the verifier can have access to the agents' specification and be able to extract and interpret the required information correctly. However, there still would be an efficiency issue to be addressed. Relying purely on the agent constraints to construct the system's state-space results in massively sized system models, as shown by the results of the techniques presented in this section.

For these reasons, mechanisms other than those focusing solely on either interaction or agent models are needed. The following section presents a third class of mechanisms that consider a combination of interaction and agent models.

6.1.3 Model Checking the Interaction/Agents Mesh

Giordano et al. (2003) identify the problems of using *mental approaches* for the verification of open multiagent systems. They argue that although the history of communication in open systems may be observable, the internal states of single agents is not. Therefore, they propose a *social approach* which is based on the view that communication actions affect the social state of a system and not the internal states of agents. The actions performed by agents are saved as part of the social state instead of individual mental states.

Interaction protocols are modelled by specifying the actions, their pre-conditions, their effects on the social state, and the resulting commitments. Interactions may further be constrained by adding temporal formulae restricting the flow of the interaction. The system model is specified in DLTL, a dynamic linear time temporal logic.

The verifier is capable of verifying four different types of properties. These are defined by the following questions:

1. Will agents always satisfy their social facts (or requirements)?

2. Will an agent eventually reach its desired state?

3. Do certain properties of the interaction protocol hold?

4. Do agents respect social facts (or requirements) at run time?

The final question is answered by observing the history of communication and verifying that it does not conflict with any of the rules of the interaction protocol. Note that this does not predict whether agents will abide to social requirements. Verification is not performed before the interaction is carried out, but after the execution of the interaction to double check that agents have indeed followed the rules.

To answer the first question, the agent's *program code* is required in order to verify whether agents are actually capable of performing the actions they need to perform. This brings us back to the main issue of verifying agent models. An agent's program code will most likely not be available for others to verify. However, every agent clearly has access to its own program code. Therefore, answering this question is performed by agents to help them decide whether they will be capable of performing the actions requested from them before committing to the protocol.

The second and third questions are properties of the interaction, and they are independent of any agent engaging in this interaction.

Since both the system model and the properties to be verified are specified in a dynamic temporal logic, then the verification problem becomes a satisfiability problem, as opposed to a traditional model checking problem. Verification is performed using Büchi automaton techniques. The automaton of both the interaction protocol IP ($\mathcal{B}_{IP}$) and the negation of the property P to be verified ($\mathcal{B}_{\neg P}$) are constructed on the fly. Finally, a verifier checks that the language accepted by the product of $\mathcal{B}_{IP}$ and $\mathcal{B}_{\neg P}$ is empty.

In their later published work (Giordano et al., 2004), the authors acknowledge that such a technique is highly inefficient, since both the system model and the temporal properties to be verified are represented as logical formulae. As a result, they suggest a slight modification to their technique: instead of generating the automaton of the interaction protocol $\mathcal{B}_{IP}$ in the traditional way, the automaton describing all possible computations is generated by making use of a $trans_a(S)$ function. This function is responsible for transforming one state into the next, for a given action a. However, the overall verification mechanism remains, more or less, the same.

The problem with this technique is that it does indeed acknowledge the pitfalls of mental approaches, but does not provide any new mechanisms for addressing these pitfalls. The reason behind this failure lies in the adopted view of multiagent systems: communication actions are believed to affect the social state of the system and not the internal states of agents. As a result, the verification of the properties described by their second and third questions is similar to the mechanisms presented in Section 6.1.1. This is because they only attempt to verify traditional temporal properties of interactions. The contribution of verifying properties described by their first question is not very strong. This is because an agent's program is accessed by its owner only to check whether the owner is capable of fulfilling certain requirements. Finally, the contribution of verifying properties described by their fourth question is also weak. This is because what is labelled as verification is in fact the process of observing the outcome of an interaction, matching it to the rules, and checking whether some agent has violated the rules or not.

The following section presents our final category of multiagent verification mechanisms. The mechanisms in this category seem to be more promising for our purposes since they offer to verify both temporal properties of interactions as well as properties of the agent mental states without any need for accessing the agent specifications.

6.1.4 Model Checking Agent Properties in Interactions Models

The mechanism presented by Bentahar et al. (2006) for verifying dialogue game protocols seems to be extremely similar to that presented by Giordano et al. (2003) and discussed in the preceding section. The interaction protocol is specified by SCCTL*, an extension to the CTL* temporal logic. Similar to Giordano et al. (2003), both the system model and the property specification are defined in one temporal language, SCCTL*. The specification is then translated into a Büchi automaton, and the model checking problem is reduced to the verification of the emptiness of the automaton resulting from the product graph of the system model and the property specification.

However, the SCCTL* language used for specifying the system model and the temporal properties is enriched with a list of predefined agent actions allowing the explicit specification of an agent's commitment towards another, along with other action formulae describing the withdrawal, satisfaction, violation, acceptance, refusal, and challenge actions. Technically speaking, the verification mechanism is one that verifies properties of the interaction models only, disregarding the possible agents involved. Nevertheless, specifying the commitments on the interaction level presents one way of verifying properties of agents, such as those describing their commitments.

The trick of enriching interaction models with non traditional interaction actions, such as commitments, is one way for verifying agent properties in interaction models. The remaining mechanisms of this section use an additional trick that enriches states with propositions about those states, or state properties.

Penczek and Lomuscio (2003) present a bounded model checking technique for the verification of epistemic properties of multiagent systems. To be able to verify epistemic properties, the system should allow the specification of such notations. As a result, the system is specified as an *interpreted system*. In addition to specifying actions, protocols, and transitions, interpreted systems make use of epistemic modalities that are defined not on an abstract model, but on the actual instantiated model.

Encoding these models is largely done by hand. For each agent, the agent's local states, its actions, and its plans are specified. The system's possible states are also encoded in binary form by hand. Additional propositional formulae may be specified to encode, for instance, the particular state of an agent. Propositions may be defined to describe state properties, along with the rules under which they hold. Finally, properties to be verified are specified in CTLK. CTLK extends the CTL temporal logic by adding knowledge modalities describing an agent's knowledge, common knowledge, global

knowledge (or "everyone knows"), and distributed knowledge. As a result, the CTLK language permits the verification of temporal occurrences of knowledge over certain propositions. For instance, in the attacking generals problem (Penczek and Lomuscio, 2003), one may verify that no general will attack before it is common knowledge that they will both attack.

In summary, although the model checkers do not have access to the agent's internal structure, verifying their knowledge is made possible since the user would have specified all the states along with the propositions that would hold in each state (through the specification of the rules under which each proposition holds). The semantics of CTLK are then sufficient for carrying out the verification.

Verification is performed on a bounded model of the system. All the possible runs are bounded to a specific length. As a result, the model checking problem is reduced to a satisfiability problem. The proof of the correctness of the transition from a model checking problem to a satisfiability one is presented in the published paper by Penczek and Lomuscio (2003). The system model and the property specification are translated into propositional formulae that are fed to a SAT solver for verification.

This verification mechanism has been applied by Woźna et al. (2005) to verify knowledge in real time, where timed autonomas are used for the specification of real time interpreted systems and the TECTLK logic for the property specification. While TECTL extends the CTL logic with real time semantics, TECTLK extends TECTL by introducing additional knowledge operators. Similar to the above technique, the system model and the properties to be verified are translated into propositional formulae which are fed to a SAT solver.

A limitation of these bounded approaches is that properties, when verified in unbounded system models, may either prove that a universal property is actually false or that an existential property is valid. This imposes a severe limitation on the verification of many protocols, especially security protocols where it might be needed to verify that some crucial security property will hold forever in the future. As a result, Kacprzak et al. (2004) extend the work of Penczek and Lomuscio (2003) to permit unbounded model checking over unbounded system models. The CTLK logic is extended by introducing past operators and making use of fixed point semantics (the *mu*-calculus semantics), resulting in CTL_pK. Similarly to the techniques above, CTL_pK properties are also translated into propositional formulae and the model checking problem is once again reduced to a satisfiability problem.

Raimondi and Lomuscio (2007) present a different mechanism for the verification

of CTLK properties in interpreted systems from that of Penczek and Lomuscio (2003). In their approach, the interpreted system is translated into a set of boolean formulae. The local states of agents, global states, and actions are all translated into boolean variables. The protocols of agents and the temporal transitions are translated into boolean functions. The evaluation function that associates the global states to propositions and the functions computing the initial and reachable states are also translated into different boolean functions. A property is then said to be satisfied in a system model if the property is satisfied in the initial states of the system model. To compute the set of states in which a CTLK property is satisfied, the verifier's algorithm makes use of the specified boolean functions to traverse the state-space and check the satisfaction of the property at each state. The result is the MCMAS model checker for multiagent systems (Lomuscio and Raimondi, 2006a).

One of the most interesting applications of the MCMAS model checker, in our view, has been presented by Lomuscio and Raimondi (2006b). The goal is to verify strategies of concurrent game structures. To achieve this, the ATL temporal logic is used for the property specification. The ATL temporal logic extends CTL with one additional operator, $\langle\!\langle\ \rangle\!\rangle$. An ATL formula $\langle\!\langle\Gamma\rangle\!\rangle\phi$ is then read as "the set of players (or agents) Γ can enforce the temporal property ϕ". An exact interpretation of such properties raises an important issue. Consider the following example, which is a simplified version of that presented by Lomuscio and Raimondi (2006b). Road Runner could either be in tunnel A or B. In order to achieve its goal of killing Road Runner, the Coyote may place its ACME Inc. bomb either at the exit of tunnel A or at the exit of B . The property $\langle\!\langle Coyote\rangle\!\rangle kill(RoadRunner)$ specifying that the Coyote can enforce the killing of Road Runner is false, since the system is non-deterministic and Road Runner could be in either of the tunnels. Nevertheless, a different and weaker interpretation of $\langle\!\langle Coyote\rangle\!\rangle kill(RoadRunner)$ holds. It is true that due to non-determinism the Coyote cannot *enforce* its goal; nevertheless, it is still Coyote's actions that can *bring about* this goal. The MCMAS model checker has been modified to verify properties expressing notations of time, knowledge, and strategies. The user may invoke the model checker with the option of specifying whether the model is to be considered deterministic or not.

The most interesting aspect of the mechanisms presented in this section is their ability to verify compelling properties of the agents without any need for accessing the agents' internal specification. Although most of these techniques are labelled as model checking techniques, in practise a SAT solver is used instead of a model checker. The

drawback of these approaches, in our opinion, lies in the complication of the verification process and their inadequacy in presenting a fully automated technique. Furthermore, the system model is not specified through a clear process calculus, but through a collection of states that are specified by hand. Usually, transition functions should also be specified to link the states together. This complication in both the specification and verification process might not be a problem when a user is verifying the system offline. However, it surely raises lots of issues (including efficiency issues) when verification needs to be performed by the agents at interaction time, as our thesis proposes.

6.2 Analysis and Synthesis

The previous section has presented a collection of the available literature on the verification of multiagent systems, focusing on model checking techniques in particular. Each paper has proposed a different language for modelling the system, a different temporal logic for the property specification, and a different verification mechanism. These various mechanisms have contributed hugely to the fields of both multiagent system specification and verification. However, it seems that the verification field of multiagent systems has not reached solid grounds yet, and this is evident from the limited application of the available mechanisms.

The multiagent verification problem can be traced back to the history of multiagent system development. Initial research on multiagent systems has followed a bottom up approach for the specification of these systems. The initial focus was on how to build intelligent agents that are autonomous, reactive, proactive, social, etc. (Wooldridge, 2001), and BDI approaches gained tremendous popularity. However, with time, researchers started realising that although having intelligent agents is crucial for the success of these systems, defining global social norms, organisational approaches, or distributed dialogues is also needed to address some of the most important issues, such as achieving distributed coordination.

Naturally, verification followed in the footsteps of specification. For this reason, this chapter has categorised the various verification techniques of the literature based on the system models and the properties they verify. We overlook the verification techniques in our categorisation because although we believe it is crucial to have a reliable, fast, and efficient model checker (especially in our case of interaction time verification), we also believe that verification is strongly dictated by the modelling approaches of the system and the type of properties it can verify. Furthermore, we

state that the system model is even more important that the property specification since the type of properties verified are also dictated by the system model. For instance, one cannot verify a property concerning an agent's knowledge if the system model did not contain any notions of this knowledge.

In comparison with the specification techniques, the mechanisms presented in Section 6.1.2 have focused solely on the verification of the agents' BDI layer. These mechanisms offer the user the opportunity of verifying rich type of properties concerning agent's beliefs and intentions. Nevertheless, the limitation of such mechanisms is clear when one tries to apply them to open systems, where agent specifications are not (and should not be) made public.

On the other hand, the mechanisms of Section 6.1.1 followed a rather naive approach that applied the model checking techniques, used traditionally in hardware and software systems, to multiagent systems without any modifications. The system model is viewed as a finite state machine and agents are treated as traditional processes that abide to their specifications. However, in reality, autonomous agents are not processes that follow rules blindly. On the contrary, they are expected to have their own say in interactions resulting in directing the flow of interactions as they see fit. This poses a severe limitation to the kind of properties that users may want to verify.

The mechanism of Section 6.1.3 properly identifies the drawbacks of the earlier mechanisms, although it fails to provide any tangible solutions.

In our view, the mechanisms of Section 6.1.4 seem to be the most promising. This is because they provide the user with the chance of verifying interesting agent properties without the need for the agent specifications. However, for the verification mechanism of this thesis, we propose agents to be equipped with a suitable verifier that allows them to automatically verify scenarios at interaction time. This is highly useful in helping agents select appropriate scenarios to join. With the complicated specification and verification mechanisms of Section 6.1.4, which require states and transition functions to be specified by hand, clearly automatic verification is not an option.

But how does the proposed mechanism of this thesis deal with each of the interaction and agent models? In our system model, the interaction layer is clearly separated from the agent BDI layer. The verification process does not require access to the agent models, but may use agent deontic constraints whenever necessary. While we argue that an agent's BDI layer is not accessible in open systems, we believe it is in the interest of agents to make some of their deontic constraints public. For instance, it would be wise for the seller in an electronic commerce scenario to publicly state its inability

to accept any payments other than those made by credit cards. Furthermore, an agent might learn about other agent's constraints from previous experience. For instance, after a couple of purchases from seller S, the buyer might decide that S is incapable of delivering good quality goods. Such information could be used during the verification process of future scenarios.

The model checker of this thesis uses a traditional process calculus approach for modelling systems and a traditional temporal logic for the property specification. As a result, it may seem incapable of verifying properties about agents' knowledge, strategies, etc. In what follows, we explore the capabilities of MCID in more detail.

Consider the cards game presented in Lomuscio and Raimondi (2006b) where a *player* agent is playing against an *environment* agent. The game is played with three cards in the deck: an Ace, a King, and a Queen. The rules of the game state that the Ace wins over the King, the King wins over the Queen, and the Queen wins over the Ace. In the first round, two cards are randomly chosen by the environment and one is given to player while the other is kept for the environment. In the second and final round, the player has the choice of either keeping its card or switching it with the third and last card in the deck. The property stating that "the player has a strategy to win the game in the initial state" is verified by Lomuscio and Raimondi (2006b). At first, this property seems to be difficult to verify with our model checker. However, careful inspection proves that the verification of such properties is indeed feasible.

Figure 6.1 presents the state graph of this game. The game starts at state s_0. The first action to be performed is to draw two cards from the deck, illustrated as $draw(X, Y)$, where X is the card handed to the player and Y the card kept for the environment. The player may then choose to either $keep$ the card or *switch* it with the remaining card in the deck. If the final state reached is s_7 then the player wins the game. However, if it is s_8, then the player loses to the environment. The property of whether the player has a strategy to win from the very beginning is a bit tricky. This is because it is not sufficient to prove whether there exists a path in which the player can win, but whether the player can actually enforce this path. Let us consider the following μ-calculus equation[1]:

[1]Note that the action an agent can perform is either an input action specified as $in(a(player, P), \_)$, an output action specified as $out(a(player, P), \_)$, or some internal computation specified as $\#(a(player, P), \_)$. To simplify things, we replace the list of all possible actions that the player can perform with $action(a(player, P), A)$. This states that the player agent can perform some action A.

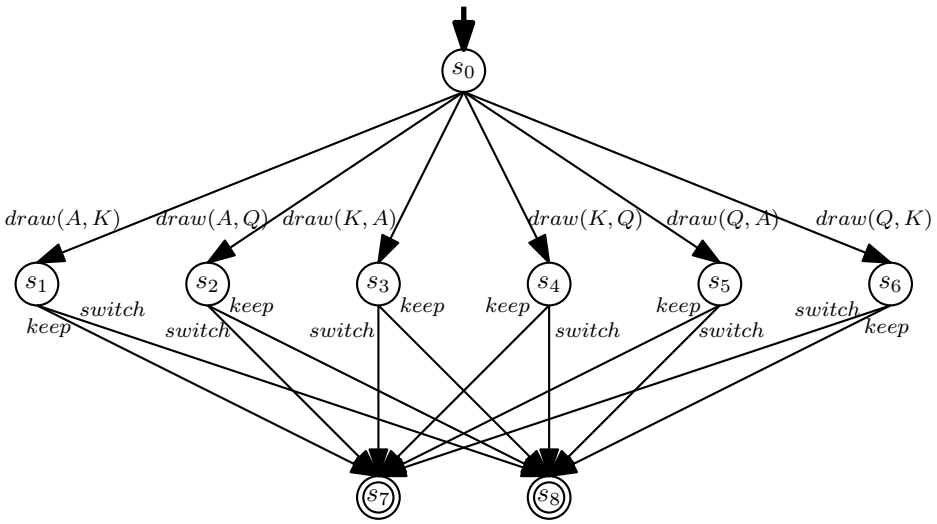


Figure 6.1: A simple card game: the interaction's state graph

$$\begin{aligned} &\mu X.\ wins(a(player,P)) \vee \\ &\quad \langle action(a(player,P),A)\rangle X \vee \\ &\quad [action(a(environment,E),\_)]X \end{aligned} \tag{6.1}$$

The equation states that either the player wins ($wins(a(player,P))$), or the player can perform an action after which the same property should hold again ($\langle action(a(player,P))\rangle X$), or for every single action the environment can take the same property will still hold ($[action(a(environment,E)]X$). Termination is guaranteed by using the least fixed point operator (μX), which ensures that eventually the player is guaranteed to win.

This property states that the player's actions can indeed *bring about* its goal of winning. However, it does not state whether the player can enforce its winning. For instance, after receiving an Ace card, the player does not actually know whether it is in state s_1 or s_2, since it does not know the environment's card. It is true that its action decides who wins. However, in each of these states the player should perform a different action to win. Therefore, it seems that if the player does not know which state it is in, then it cannot enforce its winning. Nevertheless, such a property can be specified in the μ-calculus without the need to rely on the agent's knowledge. To achieve this, Equation 6.1 is slightly modified into Equation 6.2, as follows:

$$\begin{aligned} &\mu X.\ wins(a(player,P)) \vee \\ &\quad (\langle action(a(player,P),A)\rangle \mathtt{tt} \ \wedge\ [action(a(player,P),A)]X) \vee \\ &\quad [action(a(environment,E),\_)] \end{aligned} \tag{6.2}$$

The only difference between the two equations is that the second equation does not state that the player *can* perform an action after which the same property should hold again ($\langle action(a(player, P))\rangle X$), but that the player is capable of performing some action A ($\langle action(a(player, P), A)\rangle \mathtt{tt}$) and that all the states resulting from performing action A will have to satisfy the same property all over again ($[action(a(player, P), A)]X$).

The purpose of this example is to illustrate yet another aspect of novelty of our proposed model checking technique. For instance, verifying the above two properties using the MCMAS model checker requires the user to get into the trouble of hand coding all possible states, defining the transition relations, specifying the functions that return the initial and reachable states, etc. However, all the MCID system requires is an LCC specification of the state graph of Figure 6.1. This is a simple and direct task, as in any process calculus. After the specification is ready, the first property is verified in the MCMAS system by feeding the system model to the model checker while setting the state model option to deterministic. The second property is verified by feeding the model checker the exact same property, but by setting the state model option to non-deterministic. In our approach, these two different properties are specified by two different μ-calculus equations and fed to the model checker in the same manner. Furthermore, the verification mechanism of MCID is fully automated, giving the agents the opportunity to perform the verification themselves whenever needed. Last, but not least, our model checker uses a basic process calculus for modelling systems and a basic temporal logic for the specification of properties. Unlike the specification languages of MCMAS, the languages of MCID are not enriched with extra and explicit modalities for the specification of knowledge, strategies, etc., yet they are capable of verifying properties related to these notions.

We believe this issue to be one of the major drawbacks of some of the available mechanisms. Almost every attempt in this field suggests a modification to existing languages by adding new modalities. This is done to allow the model checker to verify different classes of properties. The result, unfortunately, is that none of these techniques have been widely used.

On the other hand, our proposed mechanism does not introduce new modalities to the languages. For instance, the μ-calculus is a widely used traditional temporal logic. The only modification done is to perceive constraints as actions performed by agents. This minute modification to the language suffices to allow the verification of numerous interesting properties of both interactions and agents. As for the LCC language, although the syntax might be new to some, the notation is very similar to the most basic process

calculus, ccs. The result is a traditional, lightweight, efficient, and fully automated verification mechanism that is capable of verifying compelling properties of interaction and agent models.

Of course, further research is needed to investigate the full spectrum of properties that could be verified by our system in comparison with those of the available literature. However, to set the path for future research, we suggest that the languages of this model checker remain intact as much as possible. We do not believe adding new modalities for every new notion is a move in the right direction. Furthermore, additional notions may be addressed on a different level. For example, consider the property verified by Wooldridge et al. (2002) and states that "some agent i eventually comes to believe that $agent1$ intends that i believes a has the value 10". We view such notions to be dependent on the agent's own interpretation of messages. For instance, if agent i is weary of agent 1, then receiving a message $a = 10$ from agent 1 might lead agent i to believe that agent 1 would want it to belief that $a = 10$. However, in a different scenario where agent i trusts agent 1, receiving a message $a = 10$ from agent 1 would simply lead agent i to belief that $a = 10$. Therefore, a different interpretation of messages and their effects on agent beliefs should be modelled on a different layer than that of the model checker, especially that such layers might not be fixed and could change with different agents or scenarios. At the time being, we leave this issue for future work.

6.3 Conclusion

In comparison with the available verification mechanisms of the literature, the research conducted and introduced in this thesis presents two major aspects of novelty. First, it overcomes the dilemma of interaction versus agent model verification by combining interaction models to deontic models at run time. We argue that while the agent specification cannot be accessed by a verifier in open systems, deontic models that specify some of the agent constraints and are crucial for the verification process may be made available to the verifier. As a result, in addition to verifying static temporal properties of interactions, the verifier is capable of verifying more dynamic properties of these highly dynamic systems.

The second aspect of novelty is introducing interaction time verification for multiagent systems. This presents the agents with the opportunity of performing the verification themselves whenever needed. This is highly useful for allowing agents to automatically decide which interaction model and which group of agents is suitable

to join. Although not investigated in this thesis, the automated model checker could also be used to aid the agent's decision process in dialogue games or negotiation protocols by helping them decide which action would result in the best outcome. These applications require further research, which we currently leave for future work.

Finally, we provide the first demonstration of the above two aspects of novelty without requiring language extensions beyond traditional process calculi and temporal logics. This thesis presents a successful minimalist approach for multiagent system verification. One of the main problems of multiagent system verification is the continuous introduction of new modalities for specifying various notations. We show how a basic process calculus for modelling systems, a basic temporal logic for specifying properties, and a basic model checking algorithm is sufficient for verifying a rich variety of properties of both interaction and agent models. For future research, we suggest to increase the richness of these properties by allowing the verification of properties addressing issues such as the change of agent's beliefs within interactions. However, to achieve this, we propose the extension to the current verification process to be made on a layer different than that of the model checker for two main reasons. First, keeping the model checker as basic and general as possible implies that the model checker could be used for verifying a large group of multiagent systems. More additions to the model checker's languages would simply imply more restrictions on the set of verifiable systems. Second, we view such models to be agent dependent, i.e. different agents could have different models for their beliefs. Our ultimate goal is to have a general purpose model checker that could be used across a wide category of scenarios, yet succeed in verifying various types of properties by making use of additional layers, such as deontic, trust, belief, etc.

Chapter 7

Conclusion

Multiagent system verification poses an interesting challenge on traditional verification mechanisms, due to the highly dynamic nature of these systems. A multiagent scenario is heavily influenced by the autonomous agents executing that scenario. This usually implies that verifiers might need to take the agents specification into consideration in order to produce more reliable results. However, assuming the verifier has access to an agent's internal specification is not feasible or practical in open systems where agents are built independently with different implementation designs and languages. Furthermore, such access raises crucial security, trust, and privacy concerns.

This thesis addresses the issue of multiagent system verification by allowing the agents to perform the verification themselves, when the conditions for verification are met. The verifier is a dynamic and fully automated model checker. The system model fed to the model checker is a combination of a global interaction model and a set of local deontic models. The argument is that while internal agent specifications cannot be accessed by model checkers, a set of deontic constraints that could affect the interaction might need to be specified. This allows the prediction and prevention of failure, which could be due to errors within the interaction model, conflicts between the interaction protocol and the agents' requirements, as well as clash of interests between the agents, as early as possible in an interaction.

The resulting model checker seems promising with its varied range of potential applications, as illustrated by Section 7.2. However, the trust domain receives special attention in this thesis. This is due to the criticality of the trust issue in distributed open system, especially that this field lacks reliable solutions. When faced with new and unexplored interactions, the MCID system can be used to help agents verify which interaction along with which group of agents may be trusted.

To conclude this thesis, Section 7.1 revisits the design and implementation plans, discussing their effects on the resulting MCID verifier. Section 7.2 presents the range of applications the MCID system may be applied to. Section 7.3 recapitulates the novel contributions of this thesis. Lastly, Section 7.4 closes this thesis with a collection of interesting and stimulating ideas for future work.

7.1 Results

One of the main contributions of this thesis is introducing interaction time verification. The success of interaction time verification stems from the full automation, efficient, and lightweight nature of the MCID model checker. Previous chapters have shown the MCID model checker to be remarkably efficient. Verification results are usually returned in a fraction of a second, going up to a second or two in more complex scenarios. The model checker resides on top of the XSB tabled Prolog system, which requires around 20MB to install. However, the model checker itself is extremely compact. It is written in around 200 lines of Prolog. Another important feature of success is the model checker's ability to verify a rich variety of properties in a wide range of scenarios and applications. However, all these accomplishments are a direct consequence of the selected design and implementation plans, which we discuss below.

The use of the LCC process calculus specifically is very appealing from a verification point of view. This is because LCC, which is used for modelling multiagent systems, is also an executable process calculus. This allows the MCID model checker to avoid the complexity of translating a system into another language, eliminating the possibility of introducing errors in doing so. Many of the available verification mechanisms either require the user to translate the system model by hand, or use an automatic translator to translate between two contrasting modelling approaches. An example of the latter case are verifiers that translate between local agent models and global interaction models. Unfortunately, it is not clear what kind of information is lost, or possibly misrepresented, in such a translation.

By using the LCC executable language, the MCID system eliminates this troublesome translation process altogether. This largely contributes to both increasing the efficiency and decreasing the model checker's complexity, hence promoting automation. Permitting the verification of executable models provides a strong base for the success of interaction time verification, allowing agents to easily extract the interaction protocol and feed it to the verifier when needed.

The extremely compact nature of the MCID system and its high efficiency are largely due to two main things: the logic-based model checking algorithm and the choice of the μ-calculus temporal logic.

The logic based nature of MCID provides the efficiency needed in computing constraints and complex data structures. This is crucial for verifying system models that make heavy use of constraints and structured terms, such as LCC. The result is an efficient model checker that throws the actual burden of searching the state-space on the underlying XSB system. The logic based algorithm coupled with the μ-calculus temporal logic results in a compact Prolog code. This is because the μ-calculus notations are based on recursion. As a result, the core Prolog predicates of the model checker that are responsible for specifying the rules of property satisfaction are written in less than 30 lines of Prolog code.

Furthermore, the addition of extra layers, such as general deontic or trust layers, is done carefully in such a way that these layers do not increase the complexity of the model checker. As illustrated by Sections 3.5.3.1 and 4.6, both the deontic and trust constraints are equivalent to temporal properties of the interaction. Even if the constraints are on specific agents, in practise they are verified by checking whether the agents in question perform any illegal action in a given interaction. As a result, the addition of these extra layers does not affect the model checker itself, but require the addition of a simple and basic translator for mapping these constraints to the μ-calculus.

7.2 Possible Applications

Like any other verification mechanism, the MCID system is used to prove the satisfaction of certain properties in a given system. Traditionally, these properties have expressed safety and liveness features of interactions. However, by treating some LCC constraints as actions, similar to input and output actions, the MCID system is capable of verifying a richer set of properties. The trustworthiness property of the auction system presented in Section 4.5.2, which describes the interaction's enforcement of truth telling on the bidder agents, is one example. Furthermore, the addition of a deontic layer to the system model contributes further to the richness of the properties, allowing the verification of trust and other compatibility issues.

To broaden the type of properties that may be verified, one may introduce additional agent layers on top of the current MCID system. For instance, the addition of

a belief meta layer could be useful for verifying the progress of agent beliefs within interactions. Section 7.4, in which future work is discussed, elaborates further on this issue.

Proving the satisfaction of properties in a given system could be useful in a variety of applications. For instance, it could address issues such as collaboration and coalition formation. This is achieved by allowing the agents to find a suitable set of collaborating agents for a given interaction model, and vice versa. The suitability of scenarios is calculated with respect to the verifying agent's deontic and trust constraints.

Another useful application, which has not been investigated further in this thesis, is using the MCID system for aiding an agent's decision process. This could be very helpful in dialogue games, negotiation protocols, argumentation, etc. This issue is revisited in Section 7.4, when discussing possible future work.

7.3 Novel Contributions

This thesis presents two main aspects of novelty. First, it combines global interaction models to local deontic models when addressing the interaction versus agent model verification dilemma. Second, it introduces interaction time verification for multiagent systems. Finally, it provides the first demonstration of the above without requiring language extensions beyond traditional process calculi and temporal logics. The remainder of this section discusses these issues in further detail.

The toughest challenge of multiagent system verification is addressing the strong dependence of interaction models on the agents engaged in and directing those interactions. Some of the available literature neglects the agents involved. This results in the successful verification of global properties of interactions that could hold for any group of agents. However, the major drawback of these mechanisms is that the properties verified do not go beyond traditional safety and liveness properties. In such highly dynamic and complex systems, the verification of these properties fail to catch the interest of the agent community.

In contrast, other approaches neglect the interaction model. The focus is on the agent's BDI specification. The interaction's state-space is automatically constructed from the agent's specification. This permits the verification of compelling properties that deal with the change of an agent's belief, desire, and intentions within a given interaction. However, the major drawback of these approaches is that they assume a verifier may have access to an agent's internal specification. Although possible in

closed systems, this is neither feasible nor acceptable, due to security, trust, and privacy issues, in open systems.

The mechanism presented in this paper acknowledges the need for some way of considering the agent's constraints. For this, the MCID model checker is fed both the global interaction model and agents deontic models, instead of their internal specifications. The argument is that while internal specifications should not be accessible, it might be in the agent's interest to make some of its deontic constraints public. In other circumstances, agents might learn about each other's capabilities, for instance, through experience.

In summary, the combination of deontic models to the interaction model permits the verification of richer properties of these complex systems. This, we believe, is one of the main contributions of this thesis.

The second novel contribution lies in the successful introduction of interaction time verification. Again, the dynamic nature of the system imposes further constraints on verification. Many properties may only be verified at interaction time, when the conditions for verification are met. The MCID systems provides the agents with the opportunity of performing the verification themselves when needed. This proves to be a useful tool for agents faced with new and unexplored interactions, since it aids the agent's decision process in selecting a suitable interaction model with a suitable group of collaborating agents. Interaction time verification is made possible due to the lightweight, efficient, and fully automated nature of the MCID system. Nevertheless, automatic property specification remains an open issue and is discussed in Section 7.4.

Finally, this thesis illustrates how these novel contributions may be achieved by using a minimalist approach. It turns out that something as simple as a basic process calculus and a temporal logic as simple as the traditional μ-calculus is sufficient for proving compelling properties of these complex systems. For instance, Section 4.5.2 illustrates how the MCID system is used to verify the property stating that "an interaction model enforces truth telling on the bidders". Section 6.2 illustrates how the property stating that "an agent has a winning strategy that it can enforce" may easily be specified and verified by the MCID system. The trick lies in the addition of simple and basic LCC constraints, which are dealt with by the model checker as agents' internal actions.

We also note that one of the major advantages of sticking with a basic process calculus for modelling systems is that it considerably restricts the state-space of the interaction, a problem suffered by many multiagent verification mechanisms, especially those relying solely on agent's internal specification.

Furthermore, keeping the specification language basic seems to leave more scope for practical applications. The examples provided in this thesis, along with those presented by the OpenKnowledge project (`www.openk.org`), are proof of the language's expressive power. Many verifiers have fallen into the trap of endlessly expanding their system specification language and their temporal logic. For every new notion that needed to be verified, new modalities were added. The result was an increasing number of complex languages that were restricted to specific applications. Our proposed solution is more basic and makes use of layering.

The model checker is kept as simple and basic as possible. For verifying a new category of properties that could not be specified with the traditional μ-calculus, a new layer is added to address the new notions. For instance, a deontic policy language may be used to verify the possible violation of agent constraints. A trust policy language may be used to verify trust constraints. Moreover, these additional layers do not affect the model checking algorithm and its complexity. This is because they require the addition of a simple translator for translating each of these languages into the μ-calculus.

Further layers may be added as needed. For instance, in Section 7.4, we propose to work on adding a belief meta layer to allow the verification of agent beliefs.

This simple and basic model checker is a general purpose model checker that could be used across a wide category of scenarios. Yet, it is dynamic and powerful enough to verify various types of properties by making use of additional deontic, trust, and possibly other layers.

7.4 Improvements and Future Work

Throughout this thesis, different chapters have suggested different ideas for future work, in the hope of introducing further improvements to the MCID model checker. In this section, we recollect these suggestions into one long list. These are presented in an increasing order of significance with respect to their contribution to academic research.

- ✦ We suggest to apply the MCID system to different fields. For instance, the model checker may be used by agents to help them decide their next move in an interaction. It would be interesting to see how much help could the model checker offer agents in dialogue games, negotiation protocols, argumentation, etc.

  Another offline application in the same field could help verify properties of

these dialogue games, possibly aiding the design of argumentation or negotiation strategies.

In the field of trust, the MCID system relies on the specification of existing trust models. Therefore, we may integrate the MCID verifier with existing trust mechanisms, such as a reputation system, to achieve a realistic and complete running scenario.

- An issue that is overlooked in this thesis is that of addressing conflicting rules in DPL and TPL policy languages. This is usually a crucial issue in traditional policy languages, where one needs to know whether access should be granted to a given user or not. However, our verifier does not use the DPL or TPL languages for performing crucial actions, such as granting access permissions. The purpose of DPL and TPL constraints is to verify them against an interaction. Currently, the MCID system verifies all constraints, whether conflicting or not. If any of the constraints is not satisfied, then the system is not trusted. In other words, the MCID system gives precedence to negative rules over positive ones. It also assumes that any agent is trusted by default, unless stated otherwise. Future work can illustrate how these defaults may be changed, and how different precedence rules may be used, based on the verifying agent's requirements. Probabilistic and stochastic measures may also be useful in trust evaluation.

- A comprehensive study of possible scenarios, their interesting properties, and the level of complexity that these could reach is needed to test the limits of the model checker. The goal is to answer questions such as: What size of a system model (or state-space) would break the model checker? What properties can not be verified against a finite system model with a finite number of agents? If such problems arise, are there any solutions that would work around them? etc.

- As the state-space grows, theorem proving could be used to encourage and facilitate compositional model checking. For example, consider interaction IM_2 of Figure 5.5, where an inquirer needs to ask agents in the system some question Q. It first plays the role $inquirer1$, to ask the $finder$ agents to find suitable agents in the network for answering its question. It then plays the role $inquirer2$ to ask each of these agents its question Q. To verify that the inquirer will always receive an answer, the system is set to incorporate one inquirer, two finder peers, and two responding agents. However, another solution would be to verify

that the inquirer will always eventually play role *inquirer*2. This is followed by verifying that *inquirer*2 will always receive an answer. As systems grow, such compositional verification mechanisms could help address the state-space explosion problem, when it arises. Theorem proving may then be used to verify the transition from these sub-formulae to the general formula one intends to verify.

- To increase the richness of the verifiable properties, we suggest the addition of an extra belief layer. This will allow agents to define the rules that govern the change of their beliefs within interactions. For example, a naive rule may state that if the message $inform(m)$ is received, then the receiving agent will believe m to be true. The addition of such a layer implies that the verifier will be able to verify properties concerning the evolution of agent beliefs within interactions.

- Although verification has succeeded in being fully automated, the automation of property specification is still far-fetched. Currently, the properties to be verified are specified by the human users and are fed by the agents to the model checker. We believe that expecting agents to construct properties from scratch is currently implausible. Therefore, we assume that in the future, properties will be made available via services the same way interactions are introduced. We believe this to be a reasonable assumption, since it does not impose any additional requirements on the system. However, it would be interesting if agents can modify available properties to suit their specific interaction models. This raises two questions: (1) Can the agent learn which properties are important in a given situation? and (2) Can the agent specify such properties in an automated manner? We believe further research needs to be carried out to address these two issues. Nevertheless, we do propose some ideas for addressing the second question.

  Many properties are general properties that may be applied to several scenarios. For instance, the property stating that "an interaction model enforces truth telling on the bidders" (Property 4.2 of Chapter 4) is a domain specific property that may be verified against a range of auction interaction models. Nevertheless, every time the property is verified against a new auction model, some modifications and tweakings are necessary. For example, this property assumes that the message informing the winner W of the price P to be paid is specified as $win(W, P)$. In a different interaction, this message could be specified as $inform\_win(W, I, P)$, where I stands for the item won. Therefore, there needs to be a mapping between these two messages. We believe that current matching

and ontology mapping technologies could be useful in addressing this issue.

Another interesting property to think about is one that guarantees the realisation of a goal G. To modify such a general property, agents should be capable of replacing G with the new goal G'. However, G' should be tightly linked to the interaction model that will be verified. Therefore, the agent should have means for extracting the exact specification of the goal G' from the interaction model. This implies that the agent should have the appropriate tools for reading, understanding, and extracting such information. One way to achieve this is by spotting the variable in question (that is the variable relating to the goal G') and the actions instantiating such a variable. We leave these issues for future work.

Appendix A

The Scenarios

Some of the code presented in this thesis has been simplified to keep our examples short and comprehensible. This appendix presents the exact LCC, DPL, TPL, and μ-calculus code that is fed to the model checker for verifying the various scenarios of this thesis. The code of the travel agency scenario of Chapter 3 is presented in Section A.1, the code of the auction scenario of Chapter 4 is presented in Section A.2, and the code of the OpenKnowledge eResponse scenario of Chapter 5 is presented in Section A.3. We note that the verification results that have been presented earlier in this thesis are the results of verifying the code of this appendix.

A.1 The Travel Agency Scenario

A.1.1 The LCC Interaction Model

```
a(customer(T), C) ::=
     vacation_details(SD,ED,From,To) => a(travel_agent(_,_,_),T)
       <-- get_vacation_details(SD,ED,From,To) then
     available_flights(FL) <= a(travel_agent2(_,_),T) then
     chosen_flight(Fx) => a(travel_agent2(_,_),T)
       <-- select_flight(FL,Fx) then
     available_hotels(HL) <= a(travel_agent2(_,_),T) then
     chosen_hotel(Hx) => a(travel_agent2(_,_),T)
       <-- select_hotel(HL,Hx) then
     due_payment(TA,CD) <= a(travel_agent2(_,_),T) then
     payment(PD) => a(travel_agent2(_,_),T)
```

```
        <-- get_payment_details(CD,PD) then
      ( confirm_booking(FId,HId) <= a(travel_agent2(_,_),T)
        or
        fail_payment(PD,R) <= a(travel_agent2(_,_),T) ).

a(travel_agent(As,HD,CD),T) ::=
      vacation_details(SD,ED,From,To) <= a(customer(_),C) then
      a(query_airlines(As,SD,ED,From,To),T) then
      a(get_airline_replies(As, [], [], FL,HD,CD),T) then
      a(travel_agent2(HD,CD),T).

a(query_airlines(As, SD,ED,From,To),T) ::=
      ( flights(SD,ED,From,To) => a(airline,A1) <-- As=[A1|At] then
        a(query_airlines(At, SD,ED,From,To),T) )
      or
      null <-- As=[].

a(get_airline_replies(As, FL1, Ag, FL,HD,CD),T) ::=
      ( myappend([X],Ag,Ag2) and myappend(Flights,FL1,FL2) <--
        flights(Flights) <= a(airline,X) then
        a(get_airline_replies(As, FL2, Ag2, FL,HD,CD),T) )
      or
      null <-- all_done(As,Ag).
a(travel_agent2(HD,CD),T) ::=
      available_flights(FL) => a(customer(_),C) then
      chosen_flight(Fx) <= a(customer(_),C) then
      null <-- query(HD,D,HL) then
      available_hotels(HL) => a(customer(_),C) then
      chosen_hotel(Hx) <= a(customer(_),C) then
      null <-- calculate_bill(Fx,Hx,TA)
        and get_airline_agent(Fx,A)
        and get_hotel_agent(Hx,H) then
      due_payment(TA,CD) => a(customer(_),C) then
      payment(PD) <= a(customer(T), C) then
      a(verify_payment(PD,CD,PId,Sign,R),T) then
```

```
     ( ( null <-- \+ PId=null and \+ Sign=null then
          a(book_flight(F,Fx,CD,PId,Sign,FId),T) then
          a(book_hotel(H,Hx,CD,PId,Sign,HId),T) then
          confirm_booking(FId,HId) => a(customer(_),C) )
       or
       ( null <-- \+ R=null then
          fail_payment(PD,R) => a(customer(_),C) ) ).

a(verify_payment(PD,CD,PId,Sign,R),T) ::=
     verify_payment(PD) => a(credit_card,CD) then
     ( R=null <-- verified(PId,Sign) <= a(credit_card,CD)
       or
       PId=null and Sign=null <-- not_verified(R) <= a(credit_card,CD) ).

a(book_flight(F,Fx,CD,PId,Sign,FId),T) ::=
     book(Fx,CD,PId,Sign) => a(airline_booking,A) then
     confirm_flight(FId) <= a(airline_booking,A).

a(book_hotel(H,Hx,CD,PId,Sign,HId),T) ::=
     book(Hx,CD,PId,Sign) => a(hotel_booking,H) then
     confirm_hotel(HId) <= a(hotel_booking,H).

a(credit_card, CD) ::=
     verify(PD,PId,Sign,R) <--
        verify_payment(PD) <= a(verify_payment(_,_,_,_,_),T) then
     ( verified(PId,Sign) => a(verify_payment(_,_,_,_,_),T)
          <-- \+ PId=null and \+ Sign=null
       or
       not_verified(R) => a(verify_payment(_,_,_,_,_),T)
          <-- PId=null and Sign=null ).

a(credit_card2,CD) ::=
     make_payment(X,PId,Sign,Confirm) <--
        get_payment(X,PId,Sign) <= a(Role,Id) then
     payment(Confirm) => a(Role,Id) then
```

```
     a(credit_card2,CD).

a(airline,A) ::=
     flights(SD,ED,From,To) <= a(query_airlines(_,_,_,_,_),T) then
     flights(Flights) => a(get_airline_replies(_,_,_,_,_,_),T)
       <-- get_available_flights(SD,ED,From,To, Flights).

a(airline_booking,A) ::=
     book(Fx,CD,PId,Sign) <= a(book_flight(_,_,_,_,_,_),T) then
     get_payment(X,PId,Sign) => a(credit_card2,CD)
       <-- require_payment(Fx,X) then
     payment(Confirmation) <= a(credit_card2,CD) then
     confirm_flight(FId) => a(book_flight(_,_,_,_,_,_),T).

a(hotel_booking,H) ::=
     book(Hx,CD,PId,Sign) <= a(book_hotel(_,_,_,_,_,_),T) then
     get_payment(X,PId,Sign) => a(credit_card2,CD)
       <-- require_payment(Hx,X) then
     payment(Confirmation) <= a(credit_card2,CD) then
     confirm_hotel(HId) => a(book_hotel(_,_,_,_,_,_),T).
```

A.1.2 The DPL Deontic Constraints

A.1.2.1 Deontic Rule $\mathcal{D}1$

```
can(rule1, a(travel_agent2(_,_),T), #(query(HD,_,_)),+) <-
    member(T,syta).
```

A.1.2.2 Deontic Rule $\mathcal{D}2$

```
can(rule2, a(customer(_),C), #(get_payment_details(CD,_)), +) <-
    customer(C,CD).
```

A.1.2.3 Deontic Rule $\mathcal{D}3$

```
must(rule3, a(customer(_),C), out(payment(_),a(_,_)), -) <-
    encrypt(X) and not(X=openPGP).
```

A.1.2.4 Deontic Rule $\mathcal{D}4$

```
must(rule4, a(credit_card,CD), _, +) <-
    authenticate(x509).
```

A.1.3 The μ-Calculus Temporal Properties

A.1.3.1 Property 3.1

```
get_confirmation -=
    box([out(payment(_),a(travel_agent2(_,_),_))],prop(get_confirmation2))
    and
    box(-[],prop(get_confirmation)).

get_confirmation2 +=
    diam(-[], tt)
    and
    box(-[in(confirm_booking(_,_),a(travel_agent2(_,_),_))],
        prop(get_confirmation2)).
```

A.1.3.2 Property 3.2

```
get_result -=
    box([out(payment(_),a(travel_agent2(_,_),_))],prop(get_result2))
    and
    box(-[],prop(get_result)).

get_result2 +=
    diam(-[], tt)
    and
    box(-[in(confirm_booking(_,_),a(travel_agent2(_,_),_)),
          in(fail_payment(_,_),a(travel_agent2(_,_),_))],prop(get_result2)).
```

A.2 The Auction Scenario

A.2.1 The LCC Interaction Model

```
a(auctioneer(I,R,Bs,Bx), A) ::=
     ( invite(I,R) => a(bidder,B) <-- Bs=[B|T] then
       a(auctioneer(I,R,T,Bx), A) )
     or
     a(auctioneer2(Bx,[],R), A) <-- Bs=[].

a(auctioneer2(Bs,Vs,R), A) ::=
     add_bid([B,V], Vs, Vn) <-- bid(B,V) <= a(bidder,B) then
     ( a(auctioneer2(Bs,Vn,R), A) <-- not(all_bid(Bs,Vn))
       or
       ( won(Bx,Vx) => a(bidder,Bx)
            <-- all_bid(Bs,Vn) and winner(Vn,R,Bx,Vx) then
         deliver(I,Bx) <-- payment(P) <= a(bidder,Bx) ) ).

a(bidder, B) ::=
     invite(I,R) <= a(auctioneer(_,_,_,_), A) then
     bid(B,V) => a(auctioneer2(_,_,_), A) <-- valuation(I,V) then
     won(B,V2) <= a(auctioneer2(_,_,_), A) then
     payment(P) => a(auctioneer2(_,_,_), A) <-- payment(P).
```

A.2.2 The TPL Trust Constraints

A.2.2.1 Property 4.1

```
trust(trule1, interaction(IM), -) <-
     verify_temporal(IM,[deadlock]).
```

A.2.2.2 Property 4.2

```
trust(trule2, interaction(IM), +) <-
     my_valuation(V) and h_competitor(Ch) and l_competitor(Cl) and
     pick(Vl,Cl,V) and pick(Vll,_,Cl) and
     pick(Vh,V,Ch) and pick(Vhh,Ch,_) and
```

```
verify_temporal(IM,[bid_case(b,c,V,Cl,b,X)]) and
(winner(c,Cl,b,V,b,X) or winner(b,V,c,Cl,b,X)) and
verify_temporal(IM,[bid_case(b,c,Vl,Cl,b,Y)]) and
(winner(c,Cl,b,Vl,b,Y) or winner(b,Vl,c,Cl,b,Y)) and
verify_temporal(IM,[bid_case(b,c,Vll,Cl,c,_)]) and
(winner(c,Cl,b,Vll,c,_) or winner(b,Vll,c,Cl,c,_)) and
verify_temporal(IM,[bid_case(b,c,V,Ch,c,_)]) and
(winner(c,Ch,b,V,c,_) or winner(b,V,c,Ch,c,_)) and
verify_temporal(IM,[bid_case(b,c,Vh,Ch,c,_)]) and
(winner(c,Ch,b,Vh,c,_) or winner(b,Vh,c,Ch,c,_)) and
verify_temporal(IM,[bid_case(b,c,Vhh,Ch,b,Z)]) and
(winner(c,Ch,b,Vhh,b,Z) or winner(b,Vhh,c,Ch,b,Z)) and
\+ Y<X and \+ Z<X.
```

A.2.2.3 Property 4.3

```
trust(trule3, a(auctioneer2(_,_,_),A), #(deliver(cd,_)), -) <-
     performance(a(_,A), #(deliver(cd,_)), failure).
```

A.2.2.4 Property 4.4

```
trust(trule3, a(auctioneer2(_,_,_),A), #(deliver(dvd,_)), -) <-
     trusted(a(_,A), #(deliver(cd,_)), -).
```

A.2.2.5 Property 4.5

```
trust(trule5, a(auctioneer(_,_,_,_),A), +) <-
     rating_count(a(auctioneer(_,_,_,_),A), Total) and Total>50 and
     rating_average(a(auctioneer(_,_,_,_),A), Average) and Average>0.7 and
     rating_latest(a(auctioneer(_,_,_,_),A), 20, Latest) and Latest>0.9.
```

A.2.3 The μ-Calculus Temporal Properties

A.2.3.1 Property 4.1

```
%% Property 4.1 is a TPL property with the following temporal constraint:
deadlock +=
     diam(-[],prop(deadlock))
```

```
     or
     ( box(-[],ff) and not_terminates(a(_,_)) ).
```

A.2.3.2 Property 4.2

```
%% Property 4.2 is a TPL property with the following temporal constraint:
bid_case(B1,B2,V1,V2,W,P) -=
     box([out(bid(B,V),a(auctioneer2(_,_,_),_))],
             prop(bid_case2(B,V,B1,B2,V1,V2,W,P)))
     and
     box(-[out(bid(_,_),a(auctioneer2(_,_,_),_))],
             prop(bid_case(B1,B2,V1,V2,W,P))).

bid_case2(B,V,B1,B2,V1,V2,W,P) +=
     ( satisfied(B=B1 and V=V1 and X=B2 and Y=V2)
       or
       satisfied(B=B2 and V=V2 and X=B1 and Y=V1) )
     and
     prop(bid_case2(B,V,X,Y,W,P)).

bid_case2(B,V,X,Y,W,P) +=
     diam(-[], tt)
     and
     box(-[out(bid(_,_),a(auctioneer2(_,_,_),_))],
             prop(bid_case2(B,V,X,Y,W,P)))
     and
     box([out(bid(X,Y),a(auctioneer2(_,_,_),_))],
             prop(bid_case3(B,V,X,Y,W,P))).

bid_case3(B,V,X,Y,W,P) +=
     box([in(won(W,P),a(auctioneer2(_,_,_),_))],
             satisfied(assert(winner(B,V,X,Y,W,P))))
     and
     box(-[in(won(_,_),a(auctioneer2(_,_,_),_))],
             prop(bid_case3(B,V,X,Y,W,P))).
```

A.3 The OpenKnowledge E-Response Scenario

A.3.1 The LCC Interaction Models

A.3.1.1 Interaction IM_0

```
a(requester(Actions,Results),R) ::=
     null <-- get_peers(Actions,[],PeersSet) then
     a(requester2(Actions),R) then
     a(collector(Actions,[],Results),R) then
     a(informing_completion(PeersSet),R).

a(requester2(Actions),Co) ::=
     ( action(A) => a(performer,Peer) <-- Actions=[(Peer,A)|T] then
       a(requester2(T),Co) )
     or
     null <-- Actions=[].

a(collector(Actions,OldResults,NewResults),R) ::=
     null <-- Actions=[] and NewResults=OldResults
     or
     ( result(A,Result) <= a(performer,P) then
       null <-- my_select((P,A),Actions,Actions2) and
          my_append((P,A,Result), OldResults,IntResults) then
       a(collector(Actions2,IntResults,NewResults),R) ).

a(informing_completion(PeersSet),R) ::=
     ( done => a(performer,P) <-- PeersSet=[P|T] then
       a(informing_completion(T),R) )
     or
     null <-- PeersSet=[].

a(performer,C) ::=
     ( action(transport(Object,N1,N2)) <= a(requester2(_),Co) then
       ( null <-- at(N1)
         or
```

```
        a(performer(move(N0,N1,Path0,Vehicle),R1),C)
          <-- at(N0) and not(N0=N1) and
            realise_goal(get_path(N0,N1,Vehicle,Path0)) and
            route_details(Path,Vehicle,Fuel0,Time0) ) then
      a(performer(pick(Object,N1),R2),C) <-- at(N1) then
      a(performer(move(N1,N2,Path,Vehicle),R3),C)
        <-- realise_goal(get_path(N1,N2,Vehicle,Path)) and
          route_details(Path,Vehicle,Fuel,Time) then
      a(performer(drop(Object,N2),R4),C) <-- at(N2) then
      result(transport(Object,N1,N2),R) => a(collector(_,_,_),Co)
        <-- aggregate_results([R1,R2,R3,R4],R) then
      a(performer,C) )
    or
    ( action(move(N2)) <= a(requester2(_),Co) then
      a(performer(move(N1,N2,Path,Vehicle),R),C)
        <-- at(N1) and realise_goal(get_path(N1,N2,Vehicle,Path)) and
          route_details(Path,Vehicle,Fuel,Time) then
      result(move(N2),R) => a(collector(_,_,_),Co) then
      a(performer,C) )
    or
    done <= a(informing_completion(_),R).

a(performer(Action,Result),Id) ::=
    action(Action) => a(simulator,S) <-- simulator_id(S) then
    ok <= a(simulator,S) then
    null <-- perform(Action,Result).

a(simulator,S) ::=
    action(Action) <= a(performer(_,_),Id) then
    ok => a(performer(_,_),Id) then
    a(simulator,S).
```

A.3.1.2 Interaction IM_1

```
a(route_finder(Node1,Node2,Vehicle,RejectedRoads,Path),Id) ::=
    request_route(Node1,Node2,Vehicle,RejectedRoads)
```

```
      => a(route_service,RS) <-- route_service_id(RS) then
    route(Path) <= a(route_service,RS).

a(route_service,RS)::=
    request_route(L1,L2,V,RRoads) <= a(route_finder(_,_,_,_,_),Id) then
    ( route(Result) => a(route_finder(_,_,_,_,_),Id)
        <-- find_route(L1,L2,V,RRoads,Result)
      or
      route([]) => a(route_finder(_,_,_,_,_),Id) ).
```

A.3.1.3 Interaction IM_2

```
a(inquirer(Question,Result), I) ::=
    ( null <-- trusted_agents(Question,Agents)
      or
      a(inquirer1(Question,Finders,[],Agents),I)
        <-- finders(Question,Finders) ) then
    a(inquirer2(Question,Agents2,Result),I)
      <-- agents_set(Agents,Agents2).

a(inquirer1(Q,Fs,As,FinalAs),I) ::=
    ( request_suitable_peers(Q) => a(finder,F)
        <-- Fs=[F|NewFs] then
      my_append(Ps,As,NewAs) <--
        suitable_peers(Q,Ps) <= a(finder,F) then
      a(inquirer1(Q,NewFs,NewAs,FinalAs),I) )
    or
    null <-- Fs=[] and FinalAs=As.

a(inquirer2(Q,Rs,Result),I) ::=
    null <-- empty(Rs) and Result=nil
    or
    ( question(Q) => a(responder,R) <-- Rs=[R|NewRs] then
      ( answer(Result) <= a(responder,R)
        or
        ( no_answer <= a(responder,R) then
```

```
            a(inquirer2(Q,NewRs,Result),I) ) ) ).

a(finder,F) ::=
     request_suitable_peers(Q) <= a(inquirer1(_,_,_,_),I) then
     ( suitable_peers(Q,Ps) => a(inquirer1(_,_,_,_),I)
          <-- fetch_suitable_peers(Q,Ps)
       or
       suitable_peers(Q,[]) => a(inquirer1(_,_,_,_),I) ).

a(responder,R) ::=
     question(Q) <= a(inquirer2(_,_,_),I) then
     ( answer(A) => a(inquirer2(_,_,_),I) <-- answer(Q,A)
       or
       no_answer => a(inquirer2(_,_,_),I) ).
```

A.3.2 The DPL Deontic Constraints

A.3.2.1 Property 5.1

```
can(capability, a(performer,civilian),
    in(action(transport(neighbours(h23),n1,n2)),a(requester,_)),+) <-
     have_vehicle(X) and
     capable_of(X,transport(neighbours(h23),n1,n2)).
```

A.3.2.2 Property 5.2

```
can(timing, a(performer(_,_),civilian), #(perform(Action,_)),+) <-
     my_time_limit(L) and
     verify_temporal(request_perform_actions,
                     [timing(performer,civilian,0,0,L)]).
```

A.3.3 The TPL Trust Constraints

A.3.3.1 Property 5.3

```
trust(trule_im0, interaction(IM0), +) <-
     verify_temporal(IM0,[sending_results]).
```

A.3.3.2 Property 5.4

```
trust(trule_im2, interaction(IM2), +) <-
     verify_temporal(IM2,[get_all_replies]).
```

A.3.4 The μ-Calculus Temporal Properties

A.3.4.1 Property 5.2

```
%% Property 5.2 is a DPL property with the following temporal constraint:
timing(Role,Id,X,Y,Limit) +=
     satisfied(Z is X+Y) and
     ( ( box(-[],ff) and satisfied(Z =< Limit) )
       or
       ( diam(-[],tt)
         and
         box([#(route_details(_,_,_,T),a(Role,Id))],
             prop(timing(Role,Id,Z,T,Limit)))
         and
         box(-[#(route_details(_,_,_,_),a(Role,Id))],
             prop(timing(Role,Id,Z,0,Limit))) ) ).
```

A.3.4.2 Property 5.3

```
%% Property 5.3 is a TPL property with the following temporal constraint:
sending_results -=
     box(-[],prop(sending_results))
     and
     box([out(action(Action),a(performer,Id))],
         prop(sending_results(Action,Id))).

sending_results(Action,Id) +=
     diam(-[],tt)
     and
     box(-[in(result(Action,_),a(performer,Id))],
         prop(sending_results(Action,Id))).
```

A.3.4.3 Property 5.4

```
%% Property 5.4 is a TPL property with the following temporal constraint:
get_all_replies +=
     box([in(answer(_),a(_,_))], prop(get_all_replies2))
     and
     box(-[in(answer(_),a(_,_))], prop(get_all_replies)).

get_all_replies2 +=
     diam(-[],tt)
     and
     box(-[#(empty(_),a(_,_)),out(question(_),a(_,_))],
         prop(get_all_replies2)).
```

A.3.4.4 Property 5.5

```
all_actions_performed -=
     box([out(action(move(N1)),a(_,_))],
         prop(move_performed(N1)))
     and
     box([out(action(transport(O,_,N2)),a(_,_))],
         prop(transport_performed(O,N2)))
     and
     box(-[],prop(all_actions_performed)).

move_performed(N) +=
     diam(-[],tt)
     and
     box(-[#(perform(move(_,N,_,_),_),a(_,_))],
         prop(move_performed(N))).

transport_performed(O,N) +=
     diam(-[],tt)
     and
     box(-[#(perform(drop(O,N),_),a(_,_))],
         prop(transport_performed(O,N))).
```

A.3.4.5 Properties 5.6 and 5.8

```
terminates(a(Role,Id)) +=
     terminates(a(Role,Id))
     or
     ( diam(-[],tt)
       and
       box(-[],prop(terminates(a(Role,Id)))) ).

%% Note that Properties 5.6 and 5.8 are both defined in the same way.
%% Property 5.6 is verified against interaction IM1, and the property
%% is verified by instantiating its variables as follows: 'terminates(
%% a(route_finder(_,_,_,_,_),_Id))'. Property 5.8 is verified against
%% interaction IM2, and the property is verified by instantiating its
%% variables as follows: 'terminates(a(inquirer(_,_),_Id))'.
```

A.3.4.6 Properties 5.7 and 5.9

```
realises_goal(G) +=
     diam(-[],tt)
     and
     box(-[G],prop(realises_goal(G))).

%% Again, Properties 5.7 and 5.9 are both defined in the same way.
%% Property 5.7 is verified against interaction IM1, and the property
%% is verified by instantiating its variables as follows: 'realises_goal(
%% in(route(_),a(route_service,_)))'. Property 5.8 is verified against
%% interaction IM2, and the property is verified by instantiating its
%% variables as follows: 'realises_goal(in(answer(_),a(responder,_)))'.
```

Appendix B

The MCID Model Checker

This appendix presents the model checker's main code. As illustrated by Figure 3.14, the model checker is composed of three different entities: (1) a deontic (or trust) to temporal translator, (2) the μ-calculus proof rules, and (3) the LCC transition rules. The code for each of these entities is presented in the following three sections, respectively.

B.1 Deontic/Trust to Temporal Translator

```
%%% The following predicates present the deontic to temporal translation

d2t(can(X,Y,Z,S), RuleID) :-
     d2t(can(X,Y,Z,S) <- tt, RuleID).
d2t(must(X,Y,Z,S), RuleID) :-
     d2t(must(X,Y,Z,S) <- tt, RuleID).
d2t(can(RuleID,Agent,Action,+) <- Condition, RuleID) :-
     agent_action_set(Agent,Action,Action2),
     concat_list([RuleID,'_2'],RuleID2),
     concat_list([RuleID,'_3'],RuleID3),
     assert_rule(RuleID += (satisfied(Condition) and prop(RuleID2)) or
                           (not_satisfied(Condition) and prop(RuleID3))),
     assert_rule(RuleID2 += diam(Action2,tt) or diam(-[],prop(RuleID2))),
     assert_rule(RuleID3 -= box(Action2,ff) and box(-[],prop(RuleID3))).
d2t(can(RuleID,Agent,Action,-) <- Condition, RuleID) :-
     agent_action_set(Agent,Action,Action2),
     concat_list([RuleID,'_2'],RuleID2),
```

```
     concat_list([RuleID,'_3'],RuleID3),
     assert_rule(RuleID += (satisfied(Condition) and prop(RuleID2)) or
                           (not_satisfied(Condition) and prop(RuleID3))),
     assert_rule(RuleID2 -= diam(-Action2,prop(RuleID2)) or box(-[],ff)),
     assert_rule(RuleID3 += box(-Action2,prop(RuleID3)) and diam(-[],tt)).
d2t(must(RuleID,Agent,Action,+) <- Condition, RuleID) :-
     agent_action_set(Agent,Action,Action2),
     concat_list([RuleID,'_2'],RuleID2),
     concat_list([RuleID,'_3'],RuleID3),
     assert_rule(RuleID += (satisfied(Condition) and prop(RuleID2)) or
                           (not_satisfied(Condition) and prop(RuleID3))),
     assert_rule(RuleID2 += box(-Action2,prop(RuleID2)) and diam(-[],tt)),
     assert_rule(RuleID3 -= diam(-Action2,prop(RuleID3)) or box(-[],ff)).
d2t(must(RuleID,Agent,Action,-) <- Condition, RuleID) :-
     agent_action_set(Agent,Action,Action2),
     concat_list([RuleID,'_2'],RuleID2),
     concat_list([RuleID,'_3'],RuleID3),
     assert_rule(RuleID += (satisfied(Condition) and prop(RuleID2)) or
                           (not_satisfied(Condition) and prop(RuleID3))),
     assert_rule(RuleID2 -= box(Action2,ff) and box(-[],prop(RuleID2))),
     assert_rule(RuleID3 += diam(Action2,tt) or diam(-[],prop(RuleID3))).

%%% The following predicates present the trust to temporal translation

d2t(trust(X,Y,S), RuleID) :-
     d2t(trust(X,Y,S) <- tt, RuleID).
d2t(trust(X,Y,Z,S), RuleID) :-
     d2t(trust(X,Y,Z,S) <- tt, RuleID).
d2t(trust(RuleID,interaction(_IM),+) <- Condition, RuleID) :-
     assert_rule(RuleID += satisfied(Condition) or
                           (not_satisfied(Condition) and prop(ff))).
d2t(trust(RuleID,interaction(_IM),-) <- Condition, RuleID) :-
     assert_rule(RuleID += (satisfied(Condition) and prop(ff)) or
                           not_satisfied(Condition)).
d2t(trust(RuleID,Agent,+) <- Condition, RuleID) :-
```

```
     Action2=[in(_,Agent),out(_,Agent),#(_,Agent)],
     concat_list([RuleID,'_2'],RuleID2),
     assert_rule(RuleID += satisfied(Condition) or
                           (not_satisfied(Condition) and prop(RuleID2))),
     assert_rule(RuleID2 -= box(Action2,ff) and box(-[],prop(RuleID2))).
d2t(trust(RuleID,Agent,-) <- Condition, RuleID) :-
     Action2=[in(_,Agent),out(_,Agent),#(_,Agent)],
     concat_list([RuleID,'_2'],RuleID2),
     assert_rule(RuleID += (satisfied(Condition) and prop(RuleID2)) or
                           not_satisfied(Condition)),
     assert_rule(RuleID2 -= box(Action2,ff) and box(-[],prop(RuleID2))).
d2t(trust(RuleID,Agent,Action,+) <- Condition, RuleID) :-
     agent_action_set(Agent,Action,Action2),
     concat_list([RuleID,'_2'],RuleID2),
     assert_rule(RuleID += satisfied(Condition) or
                           (not_satisfied(Condition) and prop(RuleID2))),
     assert_rule(RuleID2 -= box(Action2,ff) and box(-[],prop(RuleID2))).
d2t(trust(RuleID,Agent,Action,-) <- Condition, RuleID) :-
     agent_action_set(Agent,Action,Action2),
     concat_list([RuleID,'_2'],RuleID2),
     assert_rule(RuleID += (satisfied(Condition) and prop(RuleID2)) or
                           not_satisfied(Condition)),
     assert_rule(RuleID2 -= box(Action2,ff) and box(-[],prop(RuleID2))).

%%% The following are other minor predicates used by d2t/2

agent_action_set(a(Role,Id), Var, Actions) :-
     var(Var), !,
     Actions=[in(_,a(Role,Id)),out(_,a(Role,Id)),#(_,a(Role,Id))].
agent_action_set(a(Role,Id),in(M),[in(M,a(Role,Id))]).
agent_action_set(a(Role,Id),out(M),[out(M,a(Role,Id))]).
agent_action_set(a(_Role,_Id),in(M,X),[in(M,X)]).
agent_action_set(a(_Role,_Id),out(M,X),[out(M,X)]).
agent_action_set(a(Role,Id),#(Constraint),[#(Constraint,a(Role,Id))]).
```

```
assert_rule(Clause) :-
     rule_id(Clause,Id),
     retractall(+=(Id,_)),
     retractall(-=(Id,_)),
     assert(Clause).
rule_id(Rule,Id) :- Rule = +=(Id,_).
rule_id(Rule,Id) :- Rule = -=(Id,_).

concat_list([X], X).
concat_list([H|T], Atom) :-
     \+ T = [],
     concat_list(T, AT),
     concat(H, AT, Atom).
concat(Name1, Name2, Concatenated) :-
     convert_to_atomic(Name1,Atom1),
     name(Atom1, A1),
     convert_to_atomic(Name2,Atom2),
     name(Atom2, A2),
     append(A1, A2, AA),
     name(Concatenated, AA).
convert_to_atomic(H,H) :- atomic(H),!.
convert_to_atomic(H,Hname) :- term_to_atom(H,Hname).
```

B.2 μ-Calculus Proof Rules

```
satisfies(_P, satisfied(X), _Mi) :- agent_call(_,X).
satisfies(_P, not_satisfied(X), _Mi) :- \+ agent_call(_,X).
satisfies(S, terminates(a(Role,Id)), _Mi) :- terminates(a(Role,Id),S).
satisfies(S, not_terminates(a(Role,Id)), _Mi) :- \+ terminates(a(Role,Id),S).
satisfies(_S, tt, _Mi).
satisfies(P, prop(F), Mi) :-
     \+ F=tt,
     clause(F += FDef, _),
     satisfies(P, FDef, Mi).
satisfies(P, prop(F), Mi) :-
     clause(F -= FDef, _),
     comp(FDef, GDef),
     sk_not(satisfies(P, GDef, Mi)).
satisfies(P, neg_prop(F), Mi) :-
     clause(F += FDef, _),
     sk_not(satisfies(P, FDef, Mi)).
satisfies(P, neg_prop(F), Mi) :-
     clause(F -= FDef, _),
     comp(FDef, GDef),
     satisfies(P, GDef, Mi).
satisfies(S, F1 or F2, Mi) :-
     satisfies(S, F1, Mi) ; satisfies(S, F2, Mi).
satisfies(S, F1 and F2, Mi) :-
     satisfies(S, F1, Mi) , satisfies(S, F2, Mi).
satisfies(S, diam(L, F), Mi) :-
     get_transition(S, L, NS, Mi, Mo), satisfies(NS, F, Mo).
satisfies(S, box(L, F), Mi) :-
     findall(satisfies(NS,F, Mo),
             get_transition(S, L, NS, Mi, Mo),
             SatisfiesClauses),
     all_true(SatisfiesClauses).
```

```
%%% The following are other minor predicates used by satisfies/3

comp(tt, ff).
comp(ff, tt).
comp(F1 and F2, G1 or G2) :- comp(F1, G1), comp(F2, G2).
comp(F1 or F2, G1 and G2) :- comp(F1, G1), comp(F2, G2).
comp(diam(L, F), box(L, G)) :- comp(F, G).
comp(box(L, F), diam(L, G)) :- comp(F, G).
comp(prop(F), neg_prop(F)).
comp(neg_prop(F),prop(F)).
comp(satisfied(X),not_satisfied(X)).
comp(not_satisfied(X),satisfied(X)).
comp(terminates(X),not_terminates(X)).
comp(not_terminates(X),terminates(X)).

get_transition(S, L, NS, Mi, Mo) :-
     transition(S, A, NS, Mi, Mo), member_check(A, L).

member_check(A, L) :- \+ var(A), member(A, L).
member_check(A, -L) :- \+ var(A), \+ member(A, L).

all_true([]).
all_true([H|T]) :- call(H), all_true(T).

terminates(a(_Role,_Id),nil).
terminates(a(Role,Id), S /// P) :-
     terminates(a(Role,Id), S), terminates(a(Role,Id), P).
terminates(a(Role,Id), a(Role2,Id2)) :-
     check_variables(Role,Id,Role2,Id2).
terminates(a(Role,Id), a(Role2,Id2) ::= _) :-
     check_variables(Role,Id,Role2,Id2).
check_variables(Role,Id,Role2,Id2) :-
     ( \+ var(Role) ; \+ var(Id) ) ,
     ( ( \+ var(Role), \+ Role2 = Role ) ; var(Role) ) ,
     ( ( \+ var(Id), \+ Id2 = Id ) ; var(Id) ).
```

B.3 LCC Transition Rules

```
transition(P, A, Q, Mi, Mo) :-
     clause(P ::  PDef, _),
     transition(PDef, A, Q, Mi, Mo).
transition(a(R,Id), A, Q, Mi, Mo) :-
     clause(a(R, Id) ::= PDef, _),
     transition(a(R, Id) ::= PDef, A, Q, Mi, Mo).
transition(a(R,Id) ::= a(R,Id), A, Q, Mi, Mo) :-
     transition(a(R, Id), A, Q, Mi, Mo).
transition(a(R,Id) ::= a(R2,Id2), A, a(R,Id) ::= Q, Mi, Mo) :-
     (\+ R2=R ; \+ Id2=Id),
     transition(a(R2, Id2), A, Q, Mi, Mo).
transition(a(_R,_Id) ::= (a(R2,Id2) ::= X), A, nil, Mi, Mo) :-
     transition(a(R2, Id2) ::= X, A, nil, Mi, Mo).
transition(a(R,Id) ::= (a(R2,Id2) ::= X), A, a(R,Id) ::= Q, Mi, Mo) :-
     transition(a(R2, Id2) ::= X, A, Q, Mi, Mo),
     \+ Q = nil.
transition(P /// Q, A, P1 /// Q, Mi, Mo) :-
     transition(P, A, P1, Mi, Mo).
transition(P /// Q, A, P /// Q1, Mi, Mo) :-
     transition(Q, A, Q1, Mi, Mo).
transition(a(R, Id) ::= P1 then P2, A, Q, Mi, Mo) :-
     transition(a(R, Id) ::= P1, A, a(R2,Id2) ::= Q1, Mi, Mo),
     \+ Q1 = nil,
     ( ( R2=R,Id2=Id,
         Q = (a(R,Id) ::= Q1 then P2)) ;
       ( (\+ R2=R; \+ Id2=Id ),
         Q = (a(R, Id) ::= (a(R2,Id2) ::= Q1) then P2) ) ).
transition(a(R, Id) ::= P1 then P2, A, a(R, Id) ::= P2, Mi, Mo) :-
     transition(a(R, Id) ::= P1, A, nil, Mi, Mo).
transition(a(R, Id) ::= P1 or P2, A, Q, Mi, Mo) :-
     transition(a(R, Id) ::= P1, A, Q, Mi, Mo) ;
     transition(a(R, Id) ::= P2, A, Q, Mi, Mo).
transition(a(_,_) ::= null, #, nil, Mi,Mi).
```

```
transition(a(R, Id) ::= M <= A, in(M,A), nil, Mi, Mo) :-
     select([A, M, a(R,Id)], Mi, Mo).
transition(a(R, Id) ::= C <-- M <= A, in(M,A), nil, Mi, Mo) :-
     select([A, M, a(R,Id)], Mi, Mo),
     agent_call(a(R,Id), C).
transition(a(R, Id) ::= M => A, out(M,A), nil, Mi, [[a(R,Id),M,A]|Mi]) :-
     length(Mi, L), L<50.
transition(a(_, _) ::= M => A, out(M,A), nil, Mi, Mi) :-
     length(Mi, L), \+ L<50,
     print_MSG('ABORTING: limit of message list exceeded...'), fail.
transition(a(R, Id) ::= P <-- C, #(X,a(R,Id)), a(R, Id) ::= P, Mi, Mi) :-
     element(X,C),
     agent_call(a(R,Id), C).

%%% The following are other minor predicates used by transition/5

element(X,X).
element(X, Y and Z) :- element(X,Y) ; element(X,Z).
element(X, Y or Z) :- element(X,Y) ; element(X,Z).

agent_call(_,tt) :- !.
agent_call(_, true(_)) :- !.
agent_call(A,realise_goal(G)) :- !, achievable_goal(A,G).
agent_call(a(R,Id), assert(C)) :- !, assert(known(a(R,Id),C)).
agent_call(a(R,Id), retract(C)) :- !, retract(known(a(R,Id),C)).
agent_call(a(R,Id), not(C)) :- !, \+ agent_call(a(R,Id), C).
agent_call(a(R,Id), A and B) :- !, agent_call(a(R,Id),A), agent_call(a(R,Id),B).
agent_call(a(R,Id), A or B) :- !,
     ( agent_call(a(R,Id), A) ; agent_call(a(R,Id), B) ).
agent_call(a(_,_), C) :-
     ( predicate_property(C, loaded); predicate_property(C, built-in) ),
     !, call(C).
agent_call(a(R,Id), C) :- known(a(R,Id), C).
agent_call(a(R,Id), C) :- known(a(R,Id), C <-- X), agent_call(a(R,Id), X).
```

Appendix C

Published Papers

The research carried out in this thesis has been published in the following conferences and workshops:

- Osman, N. (2007). A contextualised trust model for distributed open systems. In *Proceedings of the Trust in E-Systems and the Grid Workshop (Tegrid'07)*.
- Osman, N. and Robertson, D. (2007). Dynamic verification of trust in distributed open systems. In Veloso, M. M., editor, *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI'07)*, pages 1440–1445.
- Osman, N., Robertson, D., and Walton, C. (2006a). Dynamic model checking for multi-agent systems. In Baldoni, M. and Endriss, U., editors, *Proceedings of the 4th International Workshop on Declarative Agent Languages and Technologies IV (DALT'06)*, volume 4327 of *Lecture Notes in Computer Science*, pages 43–60. Springer.
- Osman, N., Robertson, D., and Walton, C. (2006b). Run-time model checking of interaction and deontic models for multi-agent systems. In Nakashima, H., Wellman, M. P., Weiss, G., and Stone, P., editors, *Proceedings of the 5th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'06)*, pages 238–240. ACM.

All these publications are accessible from the author's website on:
`http://homepages.inf.ed.ac.uk/s0233771/`

Bibliography

Benerecetti, M. and Giunchiglia, F. (2001). Model checking-based analysis of multi-agent systems. In *Proceedings of the First International Workshop on Formal Approaches to Agent-Based Systems (FAABS'00)*, pages 1–15, London, UK. Springer-Verlag.

Benerecetti, M., Giunchiglia, F., and Serafini, L. (1998). Model checking multiagent systems. *Journal of Logic and Computation*, 8(3):401–423.

Bentahar, J., Moulin, B., and Meyer, J.-J. C. (2006). A new model checking approach for verifying agent communication protocols. In *Proceedings of the Canadian Conference on Electrical and Computer Engineering (CCECE'07)*, pages 1586–1590. IEEE.

Berzins, V. (1995). *Software Merging and Slicing*. IEEE Computer Society Press, Los Alamitos, CA, USA.

Blackburn, P. and van Benthem, J. (2006). Modal logic: A semantic perspective. In *Handbook of Modal Logic*. Elsevier North-Holland.

Boley, H. (2003). Object-oriented RuleML: User-level roles, URI-grounded clauses, and order-sorted terms. In *Proceedings of the Second International Workshop on Rules and Rule Markup Languages for the Semantic Web*, pages 1–16.

Bollig, B., Leucker, M., and Weber, M. (2002). Local parallel model checking for the alternation-free μ-calculus. In *Proceedings of the 9th International SPIN Workshop on Model checking of Software (SPIN '02)*. Springer-Verlag Inc.

Bordini, R. H., Fisher, M., Pardavila, C., Visser, W., and Wooldridge, M. (2003a). Model checking multi-agent programs with CASP. In Jr., W. A. H. and Somenzi, F., editors, *Proceeding of the 15th International Conference on Computer Aided*

Verification (CAV'03), volume 2725 of *Lecture Notes in Computer Science*, pages 110–113. Springer.

Bordini, R. H., Fisher, M., Pardavila, C., and Wooldridge, M. (2003b). Model checking agentspeak. In *Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'03)*, pages 409–416, New York, NY, USA. ACM Press.

Bordini, R. H., Fisher, M., Visser, W., and Wooldridge, M. (2003c). Verifiable multi-agent programs. In Dastani, M., Dix, J., and Fallah-Seghrouchni, A. E., editors, *Proceeding of the First International Workshop on Programming Multi-Agent Systems (PROMAS'03)*, volume 3067 of *Lecture Notes in Computer Science*, pages 72–89. Springer.

Bordini, R. H., Fisher, M., Visser, W., and Wooldridge, M. (2004). State-space reduction techniques in agent verification. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'04)*, pages 896–903, Washington, DC, USA. IEEE Computer Society.

Bradfield, J. and Stirling, C. (2001). Modal logics and mu-calculi: an introduction. In Bergstra, J., Ponse, A., and Smolka, S., editors, *Handbook of Process Algebra*, pages 293–330. Elsevier, North-Holland.

Bradfield, J. C. (1998). The modal mu-calculus alternation hierarchy is strict. *Theoretical Computer Science*, 195(2):133–153.

Chaki, S., Clarke, E. M., Grumberg, O., Ouaknine, J., Sharygina, N., Touili, T., and Veith, H. (2005). State/event software verification for branching-time specifications. In Romijn, J., Smith, G., and van de Pol, J., editors, *Proceedings of the 5th International Conference on Integrated Formal Methods (IFM'05)*, volume 3771 of *Lecture Notes in Computer Science*, pages 53–69. Springer.

Cimatti, A., Clarke, E. M., Giunchiglia, F., and Roveri, M. (1999). NUSMV: A new symbolic model verifier. In *Proceedings of the 11th International Conference on Computer Aided Verification (CAV'99)*, pages 495–499, London, UK. Springer-Verlag.

Clark, K. L. and McCabe, F. G. (2003). Go! for multi-threaded deliberative agents. In *Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'03)*, pages 964–965, New York, NY, USA. ACM.

Cliffe, O. and Padget, J. (2002). A framework for checking agent interaction within institutions. In *Proceedings of the ECAI 2002 Workshop on Model Checking and Artificial Intelligence (MoChArt'02)*.

Damianou, N., Bandara, A. K., Sloman, M., and Lupu, E. C. (2002). A survey of policy specification approaches. Technical report, Department of Computing, Imperial College of Science Technology and Medicine, London, UK. Available at `http://www.doc.ic.ac.uk/~mss/Papers/PolicySurvey.pdf`.

Dignum, V., Meyer, J.-J., and Weigand, H. (2002). Towards an organizational model for agent societies using contracts. In Castelfranchi, C. and Johnson, W. L., editors, *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'02)*, pages 694–695, New York, NY, USA. ACM Press.

Dulay, N., Damianou, N., Lupu, E., and Sloman, M. (2002). A policy language for the management of distributed agents. In *Revised Papers and Invited Contributions from the Second International Workshop on Agent-Oriented Software Engineering II (AOSE '01)*, pages 84–100, London, UK. Springer-Verlag.

Edmund M. Clarke, J., Grumberg, O., and Peled, D. A. (1999). *Model Checking*. MIT Press, Cambridge, MA, USA.

Emerson, E. A. and Lei, C.-L. (1986). Efficient model checking in fragments of the propositional mu-calculus (extended abstract). In *Proceedings of the First IEEE Symposium on Logic in Computer Science*, pages 267–278, Los Alamitos, CA. IEEE Computer Society Press.

Esteva, M., Padget, J. A., and Sierra, C. (2002). Formalizing a language for institutions and norms. In Meyer, J.-J. C. and Tambe, M., editors, *Revised Papers from the 8th International Workshop on Intelligent Agents VIII (ATAL'01)*, volume 2333 of *Lecture Notes in Computer Science*, pages 348–366, London, UK. Springer-Verlag.

Esteva, M., Rodríguez-Aguilar, J. A., Sierra, C., Garcia, P., and Arcos, J. L. (2001). On the formal specifications of electronic institutions. In Frank, D. and Sierra, C., editors, *Agent Mediated Electronic Commerce, The European AgentLink Perspective*, volume 1991 of *Lecture Notes in Artificial Intelligence*, pages 126–147, London, UK. Springer-Verlag.

Giordano, L., Martelli, A., and Schwind, C. (2003). Specifying and verifying systems of communicating agents in a temporal action logic. In Cappelli, A. and Turini, F., editors, *Proceddings of the 8th Congress of the Italian Association for Artificial Intelligence: Advances in Artificial Intelligence (AI*AI'03)*, volume 2829 of *Lecture Notes in Computer Science*, pages 262–274. Springer.

Giordano, L., Martelli, A., and Schwind, C. (2004). Verifying communicating agents by model checking in a temporal action logic. In Alferes, J. J. and Leite, J. A., editors, *Proceedings of the 9th European Conference on Logics in Artificial Intelligence (JELIA'04)*, volume 3229 of *Lecture Notes in Computer Science*, pages 57–69. Springer.

Greaves, M., Holmback, H., and Bradshaw, J. (2000). What is a conversation policy? In Dignum, F. and Greaves, M., editors, *Issues in Agent Communication*, volume 1916 of *Lecture Notes in Computer Science*, pages 118–131, Heidelberg, Germany. Springer-Verlag.

Hartonas, T. (2003). A minimal calculus for situated multi-agent systems. In *Proceedings of International Conference on Advances in Infrastructure for Electronic Business, Education, Science, Medicine, and Mobile Technologies on the Internet (SSGRR'03)*.

Hassan, F., Robertson, D., and Walton, C. (2005). Addressing constraint failures in an agent interaction protocol. In *Proceedings of the 8th Pacific Rim International Workshop on Multi-Agents (PRIMA'05)*. Springer-Verlag.

Hayton, R., Bacon, J., , and Moody, K. (1998). Access control in an open distributed environment. In *Symposium on Security and Privacy*, pages 3–14, Oakland, CA. IEEE Computer Society Press.

He, H., Haas, H., and Orchard, D. (2004). Web services architecture usage scenarios. W3C working group note, World Wide Web Consortium. Available at `http://www.w3.org/TR/2004/NOTE-ws-arch-scenarios-20040211/`.

Hennessy, M. and Milner, R. (1980). On observing nondeterminism and concurrency. In *Proceedings of the 7th Colloquium on Automata, Languages and Programming*, pages 299–309, London, UK. Springer-Verlag.

Herzberg, A., Mass, Y., Michaeli, J., Ravid, Y., and Naor, D. (2000). Access control meets public key infrastructure, or: Assigning roles to strangers. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy (SP '00)*, page 2, Washington, DC, USA. IEEE Computer Society.

Hilpinen, R. (1971). *Deontic Logic: Introductory and Systematic Readings*. Reidel, Dordrecht.

Hoare, C. A. R. (1985). *Communicating Sequential Processes*. Prentice Hall, Upper Saddle River, NJ, USA.

Holzmann, G. J. (2003). *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley.

Horling, B. and Lesser, V. (2005). A survey of multi-agent organizational paradigms. *The Knowledge Engineering Review*, 19(4):281–316.

Huget, M.-P. (2002). Model checking agent UML protocol diagrams. In *Proceedings of the ECAI 2002 Workshop on Model Checking and Artificial Intelligence (MoChArt'02)*.

Huguet, M.-P., Esteva, M., Phelps, S., Sierra, C., and Wooldridge, M. (2002). Model checking electronic institutions. In *Proceedings of the ECAI 2002 Workshop on Model Checking and Artificial Intelligence (MoChArt'02)*, pages 51–58.

Huth, M., Jagadeesan, R., and Schmidt, D. A. (2001). Modal transition systems: A foundation for three-valued program analysis. In *Proceedings of the 10th European Symposium on Programming Languages and Systems (ESOP'01)*, pages 155–169, London, UK. Springer-Verlag.

Jajodia, S., Samarati, P., and Subrahmanian, V. S. (1997). A logical language for expressing authorizations. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy (SP '97)*, page 31, Washington, DC, USA. IEEE Computer Society.

Jiao, W. and Shi, Z. (1999). A dynamic architecture for multi-agent systems. In *Proceedings of the 31st International Conference on Technology of Object-Oriented Language and Systems (TOOLS'99)*, page 253, Washington, DC, USA. IEEE Computer Society.

Joseph, S., de Pinninck, A. P., Robertson, D., Sierra, C., and Walton, C. (2006). Interaction model language definition. Project Deliverable 1.1, OpenKnowledge. Available at http://www.cisa.informatics.ed.ac.uk/OK/Deliverables/D1.1.pdf/.

Kacprzak, M., Lomuscio, A., and Penczek, W. (2004). Verification of multiagent systems via unbounded model checking. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'04)*, pages 638–645, Washington, DC, USA. IEEE Computer Society.

Kagal, L., Finin, T., and Joshi, A. (2003). A policy language for a pervasive computing environment. In *IEEE 4th International Workshop on Policies for Distributed Systems and Networks*. IEEE Computer Society.

Kinny, D. (2002). The psi calculus: An algebraic agent language. In *Revised Papers from the 8th International Workshop on Intelligent Agents VIII (ATAL'01)*, pages 32–50, London, UK. Springer-Verlag.

Kozen, D. (1983). Results on the propositional mu-calculus. *Theoretical Computer Science*, 27:333–354.

Kripke, S. A. (1963). A semantical analysis of modal logic i: Normal modal propositional calculi. *Zeitschrift fur Mathematische Logik und Grundlagen der Mathematik*, 9:67–96.

Leucker, M., Somla, R., and Weber, M. (2003). Parallel model checking for LTL, CTL*, and L^2_μ. *Electronic Notes in Theoretical Computer Science*, 89(1).

Lomuscio, A. and Raimondi, F. (2006a). MCMAS: A model checker for multi-agent systems. In Hermanns, H. and Palsberg, J., editors, *Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'06), held as part of the Joint European Conferences on Theory and Practice of Software (ETAPS'06)*, volume 3920 of *Lecture Notes in Computer Science*, pages 450–454. Springer.

Lomuscio, A. and Raimondi, F. (2006b). Model checking knowledge, strategies, and games in multi-agent systems. In *Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'06)*, pages 161–168, New York, NY, USA. ACM.

Mally, E. (1926). *Grundgesetze des Sollens: Elemente der Logik des Willens*. Leuschner & Lubensky, Graz. English title: The Basic Laws of Ought: Elements of the Logic of Willing.

Marchese, M., Vaccari, L., Trecarichi, G., Osman, N., and McNeill, F. (2008). Interaction models to support peer coordination in crisis management. Project Deliverable 6.7, OpenKnowledge. Available at `http://www.cisa.informatics.ed.ac.uk/OK/Deliverables/D6.7.pdf/`. This deliverable has also been submitted to ISCRAM'08.

Mascardi, V., Demergasso, D., and Ancona, D. (2005). Languages for programming BDI-style agents: an overview. In Corradini, F., Paoli, F. D., Merelli, E., and Omicini, A., editors, *Proceedings of Dagli Oggetti agli Agenti, the 6th AI*IA/TABOO Joint Workshop "From Objects to Agents" (WOA'05)*, pages 9–15, Bologna, Italy. Pitagora Editrice.

Mateescu, R. (2003). A generic on-the-fly solver for alternation-free boolean equation systems. In Garavel, H. and Hatcliff, J., editors, *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03), held as part of the Joint European Conferences on Theory and Practice of Software (ETAPS'03)*, volume 2619 of *Lecture Notes in Computer Science*, pages 81–96. Springer.

Mateescu, R. and Sighireanu, M. (2003). Efficient on-the-fly model-checking for regular alternation-free mu-calculus. *Science of Computer Programming*, 46(3):255–281.

McGinnis, J. and Robertson, D. (2004). Realizing agent dialogues with distributed protocols. In *Developments in Agent Communication*, volume 3396 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag.

McMillan, K. L. (1993). *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell, MA, USA.

Meyer, J.-J. C. and Wieringa, R. J. (1993). Deontic logic: a concise overview. In Meyer, J.-J. C. and Wieringa, R. J., editors, *Deontic Logic in Computer Science: Normative System Specification*, pages 3–16. John Wiley and Sons Ltd., Chichester, UK, UK.

Milner, R. (1989). *Communication and Concurrency*. Prentice Hall, Hemel Hempstead, UK.

Milner, R. (1999). *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, New York, NY, USA.

Müller-Olm, M., Schmidt, D. A., and Steffen, B. (1999). Model-checking: A tutorial introduction. In *Proceedings of the 6th International Symposium on Static Analysis (SAS'99)*, pages 330–354, London, UK. Springer-Verlag.

Nicola, R. D. and Vaandrager, F. (1995). Three logics for branching bisimulation. *Journal of the Association of Computing Machinery*, 42(2):458–487.

Oasis (2003). Extensible access control markup language (XACML) version 1.1. Available at http://www.oasis-open.org/committees/xacml/repository/cs-xacml-specification-1.1.pdf.

Osman, N. (2003). Addressing constraint failures in agent dialogues. Master's thesis, School of Informatics, The University of Edinburgh, Edinburgh, UK.

Osman, N. (2007). A contextualised trust model for distributed open systems. In *Proceedings of the Trust in E-Systems and the Grid Workshop (Tegrid'07)*.

Osman, N. and Robertson, D. (2007). Dynamic verification of trust in distributed open systems. In Veloso, M. M., editor, *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI'07)*, pages 1440–1445.

Osman, N., Robertson, D., and Walton, C. (2006a). Dynamic model checking for multi-agent systems. In Baldoni, M. and Endriss, U., editors, *Preceedings of the 4th International Workshop on Declarative Agent Languages and Technologies IV (DALT'06)*, volume 4327 of *Lecture Notes in Computer Science*, pages 43–60. Springer.

Osman, N., Robertson, D., and Walton, C. (2006b). Run-time model checking of interaction and deontic models for multi-agent systems. In Nakashima, H., Wellman, M. P., Weiss, G., and Stone, P., editors, *Proceedings of the 5th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'06)*, pages 238–240. ACM.

Penczek, W. and Lomuscio, A. (2003). Verifying epistemic properties of multi-agent systems via bounded model checking. In *The Second International Joint Conference on Autonomous Agents & Multiagent Systems (AAMAS'03)*, pages 209–216. ACM.

Prior, A. N. (1955). Diodoran modalities. *Philosophical Quarterly*, 5:205–13.

Raimondi, F. and Lomuscio, A. (2007). Automatic verification of multi-agent systems by model checking via ordered binary decision diagrams. *Journal of Applied Logic*, 5(2):235–251.

Rajan, S. P., Shankar, N., and Srivas, M. K. (1995). An integration of model checking with automated proof checking. In *Proceedings of the 7th International Conference on Computer Aided Verification (CAV'05)*, pages 84–97, London, UK. Springer-Verlag.

Ramakrishna, Y. S., Ramakrishnan, C. R., Ramakrishnan, I. V., Smolka, S. A., Swift, T., and Warren, D. S. (1997). Efficient model checking using tabled resolution. In Grumberg, O., editor, *Proceedings of the 9th International Conference on Computer Aided Verification (CAV'97)*, volume 1254 of *Lecture Notes in Computer Science*, pages 143–154. Springer.

Ramakrishnan, C. R., Ramakrishnan, I. V., Smolka, S. A., Dong, Y., Du, X., Roychoudhury, A., and Venkatakrishnan, V. N. (2000). XMC: a logic-programming-based verification toolset. In Emerson, E. A. and Sistla, A. P., editors, *Proceedings of the 12th International Conference on Computer Aided Verification (CAV'00)*, volume 1855 of *Lecture Notes in Computer Science*, pages 576–580. Springer.

Ramchurn, S. D., Hunyh, D., and Jennings, N. R. (2004). Trust in multi-agent systems. *Knowledge Engineering Review*, 19(1):1–25.

Robertson, D. (2004a). A lightweight coordination calculus for agent systems. In Leite, J. A., Omicini, A., Torroni, P., and Yolum, P., editors, *Proceedings of the Second International Workshop on Declarative Agent Languages and Technologies (DALT'04)*, volume 3476 of *Lecture Notes in Computer Science*, pages 183–197. Springer.

Robertson, D. (2004b). A lightweight method for coordination of agent oriented web services. In *Proceedings of AAAI Spring Symposium on Semantic Web Services*.

Robertson, D. (2004c). Multi-agent coordination as distributed logic programming. In Demoen, B. and Lifschitz, V., editors, *Proceeding of the 20th International Conference on Logic Programming (ICLP'04)*, volume 3132 of *Lecture Notes in Computer Science*, pages 416–430. Springer.

Ross, A. (1941). *Imperatives and Logic*, volume 7. Theoria.

Sagonas, K. F., Swift, T., and Warren, D. S. (1994). XSB as an efficient deductive database engine. In Snodgrass, R. T. and Winslett, M., editors, *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data (SIGMOD'94)*, pages 442–453, New York, NY, USA. ACM Press.

Shoham, Y. and Tennenholtz, M. (1995). On social laws for artificial agent societies: Off-line design. *Artificial Intelligence*, 73(1-2):231–252.

Siebes, R., Dupplaw, D., Kotoulas, S., de Pinninck, A. P., van Harmelen, F., and Robertson, D. (2007). The openknowledge system: An interaction-centered approach to knowledge sharing. In Meersman, R. and Tari, Z., editors, *Proceedings of the 15th International Conference on Cooperative Information Systems (CoopIS)*, volume 4803 of *Lecture Notes in Computer Science*, pages 381–390. Springer.

Sloman, M. (1994). Policy driven management for distributed systems. *Journal of Network and Systems Management*, 2:333–360.

Stirling, C. (2001). *Modal and Temporal Properties of Processes*. Texts in Computer Science. Springer-Verlag.

Stirling, C. and Walker, D. (1991). Local model checking in the modal mu-calculus. *Theoretical Computer Science*, 89(1):161–177.

von Wright, G. H. (1951). Deontic logic. *Mind*, 60(237):1–15.

von Wright, G. H. (1971). A new system of deontic logic. *Danish Yearbook of Philosophy*, 1:173–182.

Walton, C. D. (2004). Model checking agent dialogues. In Leite, J. A., Omicini, A., Torroni, P., and Yolum, P., editors, *Proceedings of the Second International Workshop on Declarative Agent Languages and Technologies II (DALT '04)*, volume 3476 of *Lecture Notes in Computer Science*, pages 132–147. Springer-Verlag.

Wen, W. and Mizoguchi, F. (1999). Analysis and verification of multi-agent interaction protocols. In *Proceedings of the Third Annual Conference on Autonomous Agents (AGENTS'99)*, pages 372–373, New York, NY, USA. ACM.

Wooldridge, M. (2000). *Reasoning about Rational Agents*. The MIT Press, Cambridge, Massachussetts/London, England.

Wooldridge, M., Fisher, M., Huget, M.-P., and Parsons, S. (2002). Model checking multi-agent systems with MABLE. In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems (AAMAS '02)*, pages 952–959, New York, USA. ACM Press.

Wooldridge, M. J. (2001). *Introduction to Multiagent Systems*. John Wiley & Sons, Inc., New York, NY, USA.

Woźna, B., Lomuscio, A., and Penczek, W. (2005). Bounded model checking for knowledge and real time. In *Proceedings of the Fourth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'05)*, pages 165–172, New York, NY, USA. ACM.

Zhao, J., Cheng, J., and Ushijima, K. (1994). Literal dependence net and its use in concurrent logic programming environment. In *Proceedings of Workshop on Parallel Logic Programming*, pages 127–141.