Department of Computer Science
University College London
University of London

Model Checking Multi-Agent Systems

Franco Raimondi
f.raimondi@cs.ucl.ac.uk

Submitted in partial fulfilment of the requirements
for the degree of Doctor of Philosophy
at the University of London

June 2006

Abstract

A *multi-agent system* (MAS) is usually understood as a system composed of interacting *autonomous agents*. In this sense, MAS have been employed successfully as a modelling paradigm in a number of scenarios, especially in Computer Science. However, the process of modelling complex and heterogeneous systems is intrinsically prone to errors: for this reason, computer scientists are typically concerned with the issue of *verifying* that a system actually behaves as it is supposed to, especially when a system is complex.

Techniques have been developed to perform this task: *testing* is the most common technique, but in many circumstances a *formal proof* of correctness is needed. Techniques for *formal verification* include *theorem proving* and *model checking*. Model checking techniques, in particular, have been successfully employed in the formal verification of distributed systems, including hardware components, communication protocols, security protocols.

In contrast to traditional distributed systems, formal verification techniques for MAS are still in their infancy, due to the more complex nature of agents, their autonomy, and the richer language used in the specification of properties. This thesis aims at making a contribution in the *formal verification* of properties of MAS via model checking. In particular, the following points are addressed:

- Theoretical results about model checking methodologies for MAS, obtained by extending traditional methodologies based on Ordered Binary Decision Diagrams (OBDDs) for temporal logics to multi-modal logics for time, knowledge, correct behaviour, and strategies of agents. Complexity results for model checking these logics (and their symbolic representations).
- Development of a software tool (MCMAS) that permits the specification and verification of MAS described in the formalism of interpreted systems.
- Examples of application of MCMAS to various MAS scenarios (communication, anonymity, games, hardware diagnosability), including experimental results, and comparison with other tools available.

To my wife

Acknowledgements

I would like to thank first of all my supervisor Alessio Lomuscio, for his support, for his help, for his patience, and for his suggestions on all aspects of academic life, from writing papers to parking bicycles and scooters in all the different colleges we have been through.

A big thank you to the people who have been involved in my PhD career as second supervisors: Marek Sergot from Imperial College (where I started my PhD), David Clark and Tom Maibaum from King's College (where I passed my transfer Viva), and David Rosenblum from UCL (where I will submit this thesis).

Special thanks to Charles Pecheur who invited me to NASA Ames in California for three sunny months of model checking, and to Bozena Wozna for her precious comments and for preparing so many Italian coffees in our office.

I am very grateful to (in random order): Wojciech Penczek, Wojciech Jamroga, Mike Wooldridge, Ron van der Meyden for their comments and suggestions.

I have been extremely lucky to meet here in London Paul, Kelly, Roshan, Jungwon, and all the group of the 7^{th} floor of CS@UCL: thank you!

Paolo, thank you for continuously winning against me at Connect 4 and Tic-Tac-Toe: you showed me the meaning of a strategy. Claudio, thank you for letting me try the strategies with you.

Thank you to all the people who, in some way, helped me in London: Olli, Mari, Fabien, Leo, Dylan, Fabio, Sandra, Laura, Davide.

Thank you to all my family, for supporting all my choices.

No, I didn't forget you: thank you Anastasia for being so patient with me; now we can go on holiday.

This thesis was partially supported by EPSRC CASE grant CNA04/04

Contents

List of Figures

List of Tables

Chapter 1

Preliminaries

1.1 Introduction

1.1.1 What is a multi-agent system?

Agent.

- A person who provides a particular service (Oxford Concise Dictionary).
- - noun: a noun denoting a person or thing that performs the action of a verb (Oxford Concise Dictionary).
- A computer program that performs various actions continuously and autonomously on behalf of an individual or an organisation (Encyclopaedia Britannica).
- One who is authorised to act for or in the place of another, with delegated authority (Merriam-Webster Dictionary). Examples: estate agent, secret agent.
- In biology, (infectious) agents: a kind of virus (Encyclopaedia Britannica).
- In *The Matrix*, the term Agent is used for sentient programs that battle the humans fighting for freedom
(from `http://en.wikipedia.org/wiki/Agent`, September 2005).

Figure 1.1: Definition of *agent* from various sources.

As can be seen from Figure 1.1, there is little agreement on the actual meaning of the word *agent*. Thus, *what is an agent*? In this thesis, agents are investigated in the realm of computer science. It has been observed by many computer scientists that, albeit a definition able to gather general consensus is probably lacking even in this single discipline, typically the term *agent* denotes an entity enjoying some form of *autonomy* and other characteristics. Wooldridge and Jennings [Wooldridge and Jennings, 1995] identify the following properties of an agent:

- *Autonomy*: an agent is capable of operating without external intervention.
- *Social ability*: an agent is capable of interacting with other agents and/or with its environment.
- *Reactivity*: an agent is capable of responding to external changes.
- *Pro-activity*: an agent is capable of behaving accordingly to its goals.

Other stronger abilities are considered in [Wooldridge and Jennings, 1995]. In particular, *rationality* (defined as the ability of an agent to act consistently with its goals) is often assumed when reasoning about agents.

Examples of the use of agents in computer science include agents for automatic information retrieval from the Internet, for e-commerce, for electronic auctions, and the infamous Microsoft Office Assistant (which disappeared from standard installations as from Microsoft Office 2000). But agents are not limited to the software domain: the two NASA rovers *Spirit* and *Opportunity* [S. W. Squyres et al., 2004a, S. W. Squyres et al., 2004b] have been exploring Mars's surface for more than two years. The two rovers operate on different regions of the planet and *must* be able to act autonomously: indeed, communication between Earth and Mars takes at least 20 seconds, it is not always available due to the rotation of both planets, and it is very limited in bandwidth (12 Kbit). Thus, rovers are "instructed" at the beginning of each sol[1] to perform a certain number of "tasks" such as "move 5m towards that rock – operate the rock abrasion tool – operate the X-ray spectrometer". These tasks are executed *autonomously*, the results are returned to Earth for processing and new instructions are uploaded.
RoboCup [Kitano, 1998] is another example of "hardware" agents: RoboCup, now in its ninth edition, is a competition for teams of robotic agents playing football. Research in this subject introduced a number of innovations for autonomy, real-time reasoning, and collaborations of agents.

The reason for such a widespread use of the concept of *agent* in computer science is probably our natural attitude of ascribing certain *mental qualities* to complex systems, as noted by McCarthy in his seminal paper [McCarthy, 1979]. Indeed, as systems (both hardware and software) have grown in recent years, so did the application of the *agent paradigm* in a variety of different areas.

[1]A sol is a Martian day.

Similarly to molecular chemistry, agents are often investigated in “isolation”, as atoms are modelled as single entities. Typically, however, agents (and atoms) interact to form more complex structures. NASA’s Remote Agent (RA, [Muscettola et al., 1998]) is an example of a *multi-agent system* [Wooldridge, 2002] for space exploration. RA is a complex architecture, composed by a network of “small” agents, whose main component is an autonomous *planner* and *scheduler*, designed to operate spacecrafts with minimal human assistance. RA flew the experimental space craft DS1 between 17th and 21st of May 1999.

In this thesis the term *multi-agent system* (MAS) will denote a set of agents operating in some *environment* [Wooldridge, 2002]. In some formalisms, including the one employed in this thesis and the formalism of Fagin et al. [Fagin et al., 1995], the environment itself may be modelled as an agent, thus allowing for a uniform description of a system of agents.

Multi-agent systems are employed in the description of complex scenarios, which can be abstracted successfully by ascribing high level qualities to each agent in the system, and by assuming that agents communicate and interact (possibly, *in a rational way*) with the other agents. As noted above, Remote Agent [Muscettola et al., 1998] is an example of a multi-agent system; many other research areas employ multi-agent systems, for instance specification of communication and security protocols, distributed planning [Cox et al., 2005], hardware diagnosability and recoverability, strategic games, etc. Some examples from these domains are analysed in detail in this thesis.

The last fifteen years have seen a growing number of publications and conferences addressing issues related to agents and multi agent systems. How should an agent be formalised? How should a system of agents be formalised? How should one reason about typical agents’ stances, such as agents’ “knowledge”, beliefs, desires, etc.? How should one express properties of a single agent, and of a system of agents? This thesis takes a logic-based approach for formalising agents; in particular, intentional attitudes of agents, in the sense of [Dennett, 1987], are represented by means of *modal operators* and are interpreted using the standard Kripke semantics. Details of this approach and further discussion are presented below.

For the purposes of this thesis, only *computationally grounded* [Wooldridge, 2000a] theories for MAS will be considered. The notion of *computationally grounded (logical) theory of agency* has been introduced in [Wooldridge, 2000a, Wooldridge and Lomuscio, 2000]: intuitively, a logical theory for multi-agent systems is computationally grounded if the class of (Kripke) models in which modalities are interpreted corresponds to the set of possible *computations* of the multi-agent systems, meaning that modalities can be interpreted directly on the set of possible computations.

It will be clear soon that *computationally grounded theories of agency* are essential for the *formal verification* of multi-agent systems.

1.1.2 Definition of the problem: verification of MAS

Imagine a situation in which ESA (the European Space Agency) has sent a rover to Venus for scientific exploration. The first day of scientific exploration proceeds smoothly and at the end of the first (Venusian) day the rover is ready to start its stand-by procedure for the night. At sunset, a controller is in charge of sending a "switch-off" message to all the scientific instruments, and of moving to a "stand-by" state, waiting for the sun to rise. The stand-by state is entered when no more power is required by the on-board instruments. At sunrise, as soon as the intensity of light is sufficient, the controller resumes power and sends messages to resume scientific activity. Unfortunately, the controller was designed for switching off scientific instruments only, and the panoramic camera was not included as a "scientific" instrument. Thus, at sunset the controller senses that an instrument (the camera) is still alive, it does not move to a waiting state and, consequently, the next day it does not send any message: the scientific instruments remain off. Scientific equipment includes the antenna for communication with Earth: the system is deadlocked and the mission fails on day two.

As technology allows for systems to grow bigger, *verification*, intended as the process of *verifying that a system satisfies its design requirements*, has to play a central role in any development process to avoid unwanted behaviours. *Verification* is even more crucial in multi-agent systems, which are intrinsically more complex than "traditional" distributed systems: by definition, MAS are employed to capture high level properties of large, autonomous systems. Moreover, agents in MAS are often *highly* autonomous and out of direct human control, as in the Venus rover scenario above. Hence, in such scenarios, in-depth verification can save time and money, and improve security.

Historically, verification has played a prominent role in computer science, and particularly in software engineering and hardware design, for nearly three decades; for instance, the idea of *formal verification via model checking* (defined below) appears already in [Clarke and Emerson, 1981] and in [Quielle and Sifakis, 1981]).

Verification encompasses a number of different techniques. *Testing* is the most common verification technique. Verification is performed by running a number of *test cases* and by checking that the required properties hold in all tested runs. Many techniques are available for testing [Beizer, 1990], for instance top-down testing, thread testing, syntax-testing, etc. One of the main problems of testing is to devise a sensible set of test cases: in the example of the Venus rover, testing might not find the deadlock, unless test cases included a multi-day test.

This thesis will not be concerned with the problem of *testing multi-agent systems*; instead, the problem of *formal verification* for multi-agent systems will be investigated. *Formal* *verification* is a class of logic-based techniques, which include *theorem proving* and *model checking*. In particular, model checking is an automatic technique that has been proven effective in a number of instances (details of this are introduced in Section 2.2).

The aim of this research work is to apply model checking techniques to multi-agent systems; more in detail, the objectives of this thesis are:

- **To investigate model checking techniques for the verification of multi-agent systems**: this task comprises the development of algorithms for model checking, and the analysis of the complexity of the problem of model checking for MAS.
- **To develop a model checker for MAS**: this task consists in the development of a software tool for model checking multi-agent systems.
- **To apply the tool for the verification of typical MAS examples**: in this task, examples from the multi-agents systems literature are verified with the tool developed in the previous task.

Correspondingly, the outcomes of this research work presented in this thesis are:

- Theoretical results about model checking MAS (algorithms and complexity results).
- MCMAS (Model Checking for Multi-Agent Systems), a tool for the verification of multi-agent systems.
- Experimental results obtained by running MCMAS on a set of examples.

These outcomes are presented according to the structure presented in Section 1.1.4.

1.1.3 Applications

Automatic tools for model checking have been employed successfully in the formal verification of various scenarios. Originally, the aim of model checking was the verification of hardware circuits [Burch et al., 1992, McMillan, 1993]; indeed, even today, this remains one of the most common applications of model checking in industry [Biere et al., 1999b]). Applications of "traditional" model checking for temporal logics, however, comprise the verification of other scenarios, including:

- Communication and security protocols: the model checker SPIN [Holzmann, 1997] is a model checker specifically designed for the verification of protocols. Model checking of security protocols has been investigated by many authors, starting from the mid 90's [Marrero et al., 1997]; more references are provided in Section 2.3.

- Software: traditionally, software programs have been verified using testing techniques. In recent years various suggestions have been put forward for the *automatic* verification of software. Java PathFinder [Brat et al., 2000], a tool for the automatic verification of Java programs using model checking, is an example of these efforts.
- Diagnosability: diagnosability is the ability of some component in a system to *diagnose* the state of some other component. For instance, the ability of a controller in a plant to detect faults is an example of diagnosability. In critical applications it is essential that controllers are *always* able to diagnose the state of certain components. It has been shown [Cimatti et al., 2003] that diagnosability can be verified by means of model checking.

In general, model checking enables the formal verification of a variety of *specification patterns* [Dwyer et al., 1998] in distributed systems. Traditionally, these patterns are expressed using *temporal* logic formulae; for instance, *liveness* and *safety* are two well known examples of temporal patterns. Many other patterns are investigated in [Dwyer et al., 1998], which allow for a formal representation of a number of requirements.

This thesis investigates model checking for multi-agent systems: applying model checking techniques to logic-based MAS formalisms introduces a number of benefits with respect to "traditional" model checking, and allows for new applications. Benefits and new applications include:

1. **Direct verification** (i.e., without translation into existing model checkers). The need for MAS verification grows in parallel with the use of the multi-agent paradigm for modelling scenarios. Nevertheless, translating complex systems formalised as a MAS into a formalism suitable for the "traditional" model checkers that are available may not be straightforward, and it is prone to errors. Model checking MAS, instead, allows for the *direct* verification of typical MAS scenarios.
2. **Richer expressivity**. Some requirements are more naturally expressed using intentional stances, such as knowledge. For instance, in communication protocols it is natural to reason about "knowledge" of certain messages. In this sense, model checking MAS allows for the verification of requirements that may not be expressed easily as temporal patterns.
3. **Improved efficiency**. Model checking MAS can improve the efficiency of verification even for traditional model checking. This is the case, for instance, with diagnosability. Section 6.4 introduces a non-temporal characterisation of diagnosability using agents that can reduce the size of the model being verified.

Chapter 6 motivates in further details the points above by exploring various MAS examples.

1.1.4 Structure of this thesis

This work presents theoretical results about model checking multi-agent systems, the implementation of a model checker for MAS, and applications of the model checker to various examples. The overall structure of the thesis is depicted in Figure 1.2. Specifically:

- Chapter 1 provides a motivational introduction.
- Chapter 2 summarises some background material on modal logics, multi-agent system theories, and model checking. This material enables the introduction of some technical details in Section 2.3, which contains a literature review of model checking in multi-agent systems.
- Chapter 3 presents an OBDD-based methodology for model checking multi-agent systems described in the formalism of interpreted systems. Model checking algorithms are presented for the verification of various modalities.
- Chapter 4 presents complexity results about model checking multi-agent systems, both for "explicit" model checking and for "symbolic" model checking.
- Chapter 5 describes the language ISPL (Interpreted Systems Programming Language, a language for describing multi-agent systems and their requirements), and introduces the implementation of the model checker MCMAS (Model Checking for Multi-Agent Systems).
- Chapter 6 analyses various applications of MCMAS to communication and security protocols, to strategic games, and to diagnosability and recoverability. Experimental results are presented and, where possible, they are compared to other model checkers.

Chapter 7 assesses the results obtained, presents open issues and sketches possible extensions of this work.

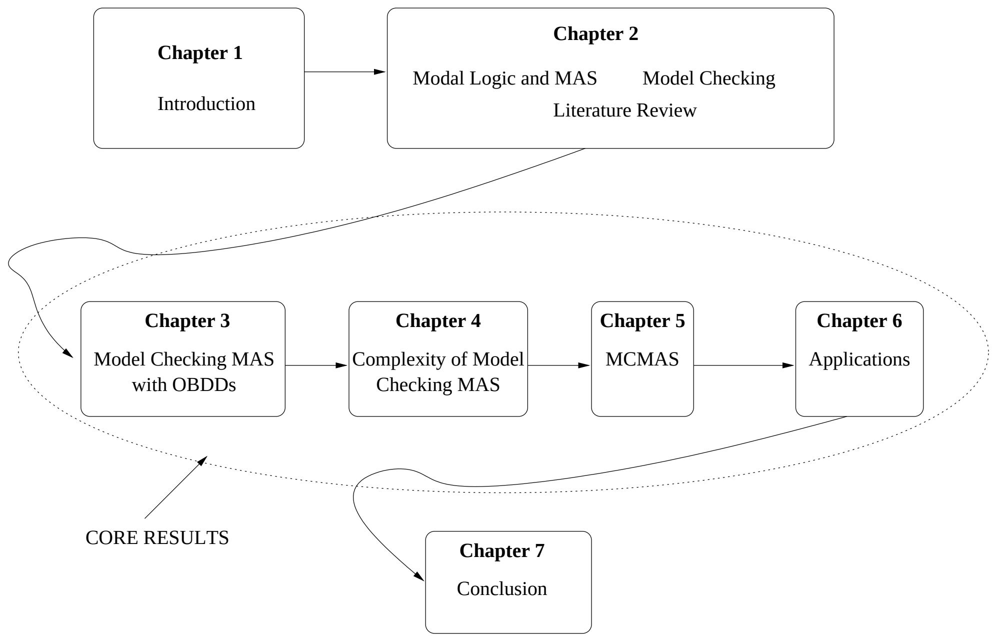

Figure 1.2: Structure of the thesis.

Publication note: All the results presented in this thesis are the outcome of the author's own research, except where explicitly stated. The majority of the results have also been published in the proceedings of various international conferences and workshops.

In particular, part of Section 2.1.7.3 appears in [Raimondi and Lomuscio, 2005c, Lomuscio and Raimondi, 2006c].

The material in Chapter 3 appeared, in a shorter version, in [Raimondi and Lomuscio, 2004c, Raimondi and Lomuscio, 2004b, Raimondi and Lomuscio, 2005c, Raimondi and Lomuscio, 2005a, Lomuscio and Raimondi, 2006b].

The complexity results appearing in Chapter 4 have been presented in [Raimondi and Lomuscio, 2005b, Lomuscio and Raimondi, 2006a].

The basic implementation of MCMAS has been presented in [Raimondi and Lomuscio, 2004d]; extensions of MCMAS appear in [Raimondi and Lomuscio, 2004a, Raimondi and Lomuscio, 2005c, Raimondi and Lomuscio, 2005a, Lomuscio and Raimondi, 2006b]. Chapter 5 presents this material in a uniform way.

The examples and experimental results appearing in Chapter 6 have been presented in [Raimondi and Lomuscio, 2004c, Raimondi and Lomuscio, 2004b, Raimondi and Lomuscio, 2005a, Raimondi et al., 2005]. The examples of diagnosability in Section 6.4 have been investigated initially by Charles Pecheur and Franco Raimondi at NASA Ames Research Center.

Chapter 2

Background literature

2.1 Modal logics and multi-agent systems

The model checking techniques presented in Chapter 3 rely on the logic-based characterisation of multi-agent systems. This section introduces the relevant background of *modal logic* that shall be used in the remainder of the thesis. The material presented below summarises standard results appearing in [Chellas, 1980, Goldblatt, 1992, Blackburn et al., 2001, Gabbay et al., 2003].

Following the modal logic introduction, this section introduces complexity theory, summarises various logic-based MAS theories, and it presents the formalism of *interpreted systems* [Fagin et al., 1995].

2.1.1 Syntax and axiomatic systems

Let P be a countable set of atomic formulae, usually denoted by $p, q, \ldots$. The language $\mathcal{L}$ of *propositional modal logic* is defined by the set of well-formed formulae $\varphi \in \mathcal{L}$:

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid \Box\varphi.$$

Other operators are introduced in a standard way. In particular, $\Diamond\varphi = \neg\Box\neg\varphi$, $\varphi \wedge \psi = \neg(\neg\varphi \vee \neg\psi)$, $\bot = p \wedge \neg p$, $\top = \neg\bot$, and $p \implies q = \neg(p \wedge \neg q)$. Possible readings of the formula $\Box\varphi$ are "It is necessarily true that φ", "It will always be true that φ", "It ought to be that φ", "It is known that φ". In the following sections other symbols may be used for the modal operator $\Box$, including K (to be read "it is known that") and O (to be read "it ought to be that").

A *schema* is a set of formulae all having the same syntactic form. For example, the schema

D:	$\Box A \implies \Diamond A$
T:	$\Box A \implies A$
4:	$\Box A \implies \Box\Box A$
5:	$\neg\Box A \implies \Box\neg\Box A$

Table 2.1: Some common names for axioms.

Name	**Axioms**
S4	KT4
S5	KT5

Table 2.2: Some common names for logics.

$\Box A \implies A$ is the set of formulae:

$$\{\Box B \implies B : B \in \mathcal{L}\}.$$

A *logic* is a set $\mathbf{L} \subseteq \mathcal{L}$ such that

- $\mathbf{L}$ includes all tautologies of propositional logic.
- $\mathbf{L}$ is closed under *Modus Ponens*, i.e., if $A, A \implies B \in \mathbf{L}$, then $B \in \mathbf{L}$.

Members of $\mathbf{L}$ are called *theorems* and it is usually written $L \vdash \varphi$ when $\varphi \in \mathbf{L}$.

A logic is *normal* if:

- it contains the schema
$$\mathbf{K} : \Box(A \implies B) \implies (\Box A \implies \Box B);$$
- it is closed under necessitation, i.e.,
$$\text{if } L \vdash A, \text{then } L \vdash \Box A.$$

The smallest normal logic is denoted[1] by $\mathbf{K}$. Following standard conventions, $\mathbf{KX_1} \ldots \mathbf{X_n}$ denotes the smallest normal logic containing the schemata $X_1 \ldots X_n$; these schemata are also called the *axioms* of the logic. Table 2.1 lists the names of some common axioms. Table 2.2 lists the names of some common logics.

Notice that a logic $\mathbf{L}$ is identified by a set of *axioms* and by a set of *inference rules*. For instance, the logic $\mathbf{K}$ is identified by the axiom $\Box(A \implies B) \implies (\Box A \implies \Box B)$, by all propositional tautologies, and by the rules modus ponens and necessitation.

[1] Notice that the same symbol is used to denote a schema and a logic. It should be clear from the context which is the intended meaning.

Such a set of axioms together with the set of rules, is usually denoted with the name of *Hilbert-style inference system*, or with the name of *axiomatic system*. In this sense, $L \vdash \varphi$ is usually read as "φ is derivable from the axioms of **L**", i.e., φ can be derived from the axioms using appropriate inference rules of the logic **L**.

2.1.2 Kripke semantics

Formulae of propositional logic are interpreted by assigning a value *true* ($\top$) or *false* ($\bot$) to atomic formulae (also denoted with the term *Boolean variables*), by means of an evaluation function $V : P \rightarrow \{\top, \bot\}$. In contrast, the interpretation of modal formulae requires more complex structures, known as *Kripke models*[2].

Given a set of Boolean variables P, a *(Kripke) model* is a tuple $M = (W, R, V)$, where W is a set of *possible worlds*, $R \subseteq W \times W$ is a binary relation (the *accessibility relation*), and $V : W \rightarrow 2^P$ is an evaluation function assigning sets of Boolean variables to possible worlds (intuitively, this is the set of variables true at a possible world). Notice that, equivalently, V could be defined as a *relation* $V \subseteq W \times P$.

It is usually written $M, w \models \varphi$ when a modal formula φ is true (or *holds*, or it is *satisfied*) at world w in a model M. $\models$ is defined inductively as follows:

$$\begin{array}{lll}
M, w \models p & \text{iff} & p \in V(w); \\
M, w \models \neg\varphi & \text{iff} & M, w \not\models \varphi; \\
M, w \models (\varphi \vee \psi) & \text{iff} & M, w \models \varphi \text{ or } M, w \models \psi; \\
M, w \models \Box\varphi & \text{iff} & \text{for all } w' \in W,\ wRw' \text{ implies } M, w' \models \varphi.
\end{array}$$

Intuitively, $\Box\varphi$ is true at a world w in a model M if φ is true at all worlds w' that are accessible via R from w.

A formula φ is true in a model M, denoted by $M \models \varphi$, if $M, w \models \varphi$ for all $w \in W$ (some authors say that, in this case, φ is valid in M).

A *frame* F is a pair $F = (W, R)$, where W is a set of worlds and $R \subseteq W \times W$ is a binary relation. Thus, a model can be seen as a pair $M = (F, V)$, where V is an evaluation function as above; in this case, the model M is said to be *based* on F. A formula is *valid in a frame* F, denoted with $F \models \varphi$, if $M \models \varphi$ for all possible models M based on F.

[2]In fact, other semantics are possible for modal formulae. In this thesis, the semantics for temporal and strategy operators is introduced in Section 2.1.4, and the semantics of *interpreted systems* is introduced in Section 2.1.7.1. Other semantics are possible, e.g., *modal algebras* [Gabbay et al., 2003].

Logic	**Class of frames**
K	All frames
KD	Serial frames
KT	Reflexive frames
S4 = **KT4**	Transitive and reflexive frames
S5 = **KT5**	Transitive, reflexive and symmetric frames

Table 2.3: Classes of frames.

2.1.3 Completeness and correspondence results

Let $\mathcal{C}$ be a class of frames, for instance the class of all frames with a finite number of worlds, or the class of all frames in which the accessibility relation is serial (see below). A formula φ is valid in a class of frames $\mathcal{C}$, denoted with $\mathcal{C} \models \varphi$, if $F \models \varphi$ for all frames $F \in \mathcal{C}$.

A logic **L** (in the sense of Section 2.1.1) is *sound* with respect to a class of frames $\mathcal{C}$ if, for every formula φ, $\mathbf{L} \vdash \varphi$ implies $\mathcal{C} \models \varphi$. A logic **L** is *complete* with respect to a class of frames $\mathcal{C}$ if $\mathcal{C} \models \varphi$ implies $\mathbf{L} \vdash \varphi$. A logic **L** is *determined by* $\mathcal{C}$ if **L** is sound and complete with respect to $\mathcal{C}$. Equivalently, it is also said that a logic **L** *corresponds* to a class of frames $\mathcal{C}$.

Kripke semantics are particularly attractive because many modal logics correspond to simple classes of frames, defined by imposing particular requirements to the accessibility relation R. A relation $R \subseteq W \times W$ is

- *Reflexive* if, for all $w \in W$, wRw.
- *Symmetric* if, for all $w, w' \in W$, wRw' implies $w'Rw$.
- *Transitive* if, for all $w, w', w'' \in W$, wRw' and $w'Rw''$ imply wRw''.
- *Serial* if, for all $w \in W$, there exists $w' \in W$ such that wRw'.

A relation R is an *equivalence* relation if it is reflexive, symmetric, and transitive. A frame is *reflexive* (resp.: symmetric, transitive, serial) if its accessibility relation is reflexive (resp.: symmetric, transitive, serial). Some well-known correspondence results are presented in Table 2.3 (more details and formal proofs can be found in [Chellas, 1980, Goldblatt, 1992]).

2.1.4 Extended Kripke semantics

The interpretation of the $\Box$ operator is not necessarily limited to the evaluation of a single-step accessibility relation. This is the case, for example, with modal operators to reason about time, and with modal operators to reason about strategies. This section introduces various semantics for these operators, and analyses their correspondence with Kripke semantics.

2.1.4.1 The temporal logic CTL

Given a countable set P of atomic formulae, the language $\mathcal{L}_{\mathbf{CTL}}$ of *Computational Tree Logic* (**CTL**, [Clarke and Emerson, 1981, Emerson, 1990]) is defined by

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid EX\varphi \mid EG\varphi \mid E[\varphi U \varphi].$$

In this definition, $p \in P$ is an atomic formula; $EX\varphi$ is read "there exists a path such that at the next state φ holds"; $EG\varphi$ is read "there exists a path such that φ holds globally along the path"; $E[\varphi U \psi]$ is read "there exists a path such that φ holds until ψ holds". Notice that **CTL** operators are composed of a pair of symbols: the first symbol is a quantifier over paths (E), while the second symbol expresses some constraint over paths. Also, notice that EU is a *binary* operator, sometimes written as $EU(\varphi, \psi)$.

Traditionally, the syntax of **CTL** includes the following operators as well: $EF\varphi, AX\varphi, AG\varphi, A[\varphi U \psi], AF\varphi$. These are read, respectively: "there exists a path such that φ holds at some future point"; "for all paths, in the next state φ holds"; "for all paths, φ holds globally"; "for all paths, φ holds until ψ holds"; "for all paths, φ holds at some point in the future". These additional **CTL** operators can be used to ease the specification process of various requirements but they are in fact definable in terms of the (minimal) set of **CTL** operators EX, EG, EU (see below).

The semantics of **CTL** is given in terms of *transition systems*: a transition system $T = (S, R_t, V)$ is a tuple in which S is a set of states, $R_t \subseteq S \times S$ is a *transition relation*, and $V : S \to 2^P$ is an evaluation function. The transition relation R_t models *temporal* transitions between states: given two states s and s' of S, sR_ts' means that s' is an immediate successor of s. It is usually assumed that every state has a successor, i.e., the transition relation R_t is serial[3]. A *path* π *in* T is an infinite sequence of states $\pi = (s_0, s_1, \dots)$ such that $s_iR_ts_{i+1}$ for all $i \geq 0$; the i-th state in the path is denoted by $\pi(i)$. **CTL** formulae are interpreted at a state s in a transition system T as follows:

[3]In this thesis, all transition systems are assumed to be serial; non-serial transition systems are analysed in [Pucella, 2005].

$T, s \models p$	iff	$p \in V(s)$;
$T, s \models \neg\varphi$	iff	$T, s \not\models \varphi$;
$T, s \models \varphi_1 \vee \varphi_2$	iff	$T, s \models \varphi_1$ or $T, s \models \varphi_2$;
$T, s \models EX\varphi$	iff	there exists a path π such that $\pi(0) = s$ and $T, \pi(1) \models \varphi$;
$T, s \models EG\varphi$	iff	there exists a path π such that $\pi(0) = s$ and $T, \pi(i) \models \varphi$ for all $i \geq 0$;
$T, s \models E[\varphi U \psi]$	iff	there exists a path π and a $k \geq 0$ such that $\pi(0) = s$ and $T, \pi(k) \models \psi$ and $T, \pi(i) \models \varphi$ for all $0 \leq i < k$.

$T \models \varphi$ denotes that the formula φ holds in all states $s \in S$ (notice that some authors include a set of *initial* states $I \subseteq S$ in the definition of a transition system, and they write $T \models \varphi$ if $T, s \models \varphi$ for all states $s \in I$).

Given the semantics above, the following equivalences hold for **CTL** [Huth and Ryan, 2004]:

$$\begin{aligned} EF\varphi &\equiv E[\top U \varphi]; \\ AX\varphi &\equiv \neg EX\neg\varphi; \\ AG\varphi &\equiv \neg EF\neg\varphi; \\ A[\varphi U \psi] &\equiv \neg(E[\neg\psi U(\neg\varphi \wedge \neg\psi)] \vee EG\neg\psi); \\ AF\varphi &\equiv A[\top U \varphi] \equiv \neg EG\neg\varphi. \end{aligned}$$

These equivalences show that all the **CTL** operators can be expressed using EX, EG, and EU only.

It is worth noticing that a transition system *is* a Kripke model. The difference between **CTL** semantics and traditional Kripke semantics lies in how the Kripke model is used: in Kripke semantics, formulae are interpreted directly on the model, while **CTL** formulae are interpreted on the possible *computations* arising from the model. Figure 2.1 shows a transition system without evaluation function (or, equivalently, a Kripke frame) on the left-hand side, and the initial branching structure of the corresponding computations on the right-hand side.

It is possible to provide an axiomatic system for **CTL** and, using non-standard techniques, it is possible to prove that **CTL** is sound and complete with respect to the class of serial frames (this result was established in [Emerson and Halpern, 1985]). A detailed discussion of this matter is beyond the scope of this thesis; more details can be found in [Emerson and Halpern, 1985, Goldblatt, 1992].

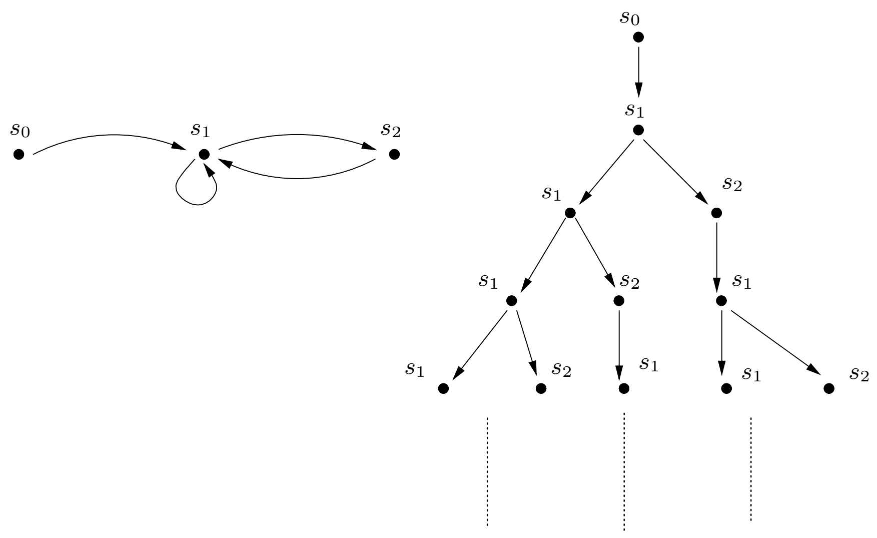

Figure 2.1: A Kripke model (left) and corresponding computations (right).

2.1.4.2 Other temporal logics

This thesis is concerned mainly with branching time structures *à la* **CTL**, but other temporal semantics are sometimes referenced. This section summarises briefly the temporal logics **LTL**, and μ-calculus.

LTL: In contrast to **CTL**, *Linear Temporal Logic* (**LTL**, [Pnueli, 1981]) is a logic to reason about *linear* sequences of states. The language $\mathcal{L}_{\mathbf{LTL}}$ of **LTL** is defined in terms of a set of atomic propositions P, as follows:

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid X\varphi \mid \varphi U \psi \mid G\varphi.$$

Traditionally, the unary operator F is included in the syntax, even if it can be derived from G:

$$F\varphi \equiv \neg G \neg \varphi.$$

LTL formulae are interpreted in a transition system $T = (S, R_t, V, I)$, where S, R_t, and V are defined as in **CTL**, and $I \subseteq S$ is a set of initial states. Similarly to **CTL**, a *path* π is an infinite sequence of states $\pi = (s_0, s_1, \dots)$ such that $s_i R_t s_{i+1}$ for all $i \geq 0$. The "tail" of a path starting from a state s_i is denoted by π_i, i.e., $\pi_i = (s_i, s_{i+1}, \dots)$. Satisfaction of a formula φ with respect to a path π in a transition system T is defined inductively as follows:

$$\begin{array}{lll}
T,\pi \models p & \text{iff} & p \in V(\pi(0)); \\
T,\pi \models \neg\varphi & \text{iff} & T,\pi \not\models \varphi; \\
T,\pi \models \varphi_1 \vee \varphi_2 & \text{iff} & T,\pi \models \varphi_1 \text{ or } T,\pi \models \varphi_2; \\
T,\pi \models X\varphi & \text{iff} & T,\pi_1 \models \varphi; \\
T,\pi \models (\varphi U \psi) & \text{iff} & \text{there exists a } k \text{ such that } \pi_k \models \psi \text{ and } \pi_i \models \varphi \text{ for all } 0 \leq i < k; \\
T,\pi \models G\varphi & \text{iff} & \pi_k \models \varphi \text{ for all } k \geq 0.
\end{array}$$

It is written $T \models \varphi$ when a formula φ holds in all paths starting from an initial state.

μ-calculus: propositional μ-calculus [Kozen, 1983] is a modal logic extended with operators for least and greatest fix-points of formulae. Given a set P of atomic formulae and a set $\mathcal{V}$ of variables, μ-calculus formulae are defined as follows[4]:

$$\varphi ::= p \mid v \mid \neg p \mid \varphi \vee \varphi \mid \Box\varphi \mid \Diamond\varphi \mid \mu y.\varphi(y) \mid \nu y.\varphi(y).$$

In the definition above, $p \in P$ is an atomic formula, $v \in \mathcal{V}$ is a variable, the unary operators $\Box$ and $\Diamond$ are the standard modal operators, μ and ν are the least and greatest fix-point operators. The expression $\mu y.\varphi(y)$ denotes the least fix-point of $\varphi(y)$, where y is a free variable appearing in φ.

Formulae of μ-calculus are interpreted in a transition system $T = (S, R, V, Val)$, where $V : P \rightarrow 2^S$ is an evaluation function which assigns sets of states to atomic formulae, and $Val : V \rightarrow 2^S$ is an evaluation function from variables to sets of states. Given a transition system T and a μ-calculus formula φ, the set of states in which φ holds, denoted by $[\![\varphi]\!]$, is defined inductively as follows[5]:

$$\begin{array}{rcl}
[\![p]\!] & = & V(p); \\
[\![v]\!] & = & Val(v); \\
[\![\neg p]\!] & = & S \backslash V(p); \\
[\![\varphi \vee \psi]\!] & = & [\![\varphi]\!] \cup [\![\psi]\!]; \\
[\![\Box\varphi]\!] & = & \{s \in S : \text{for all } s' \in S \text{ such that } sRs',\ s' \in [\![\varphi]\!]\}; \\
[\![\Diamond\varphi]\!] & = & \{s \in S : \text{there exists } s' \in S \text{ such that } sRs' \text{ and } s' \in [\![\varphi]\!]\}; \\
[\![\mu y.\varphi(y)]\!] & = & \bigcap\{Q \subseteq S : [\![\varphi]\!]_{Val[y \leftarrow Q]} \subseteq Q\}; \\
[\![\nu y.\varphi(y)]\!] & = & \bigcup\{Q \subseteq S : Q \subseteq [\![\varphi]\!]_{Val[y \leftarrow Q]}\}.
\end{array}$$

In the definition above, $Val[y \leftarrow Q]$ is an evaluation function such that $Val[y \leftarrow Q](y) = Q$, and $Val[y \leftarrow Q](z) = Val(z)$ if $z \neq y$.

Propositional μ-calculus strictly subsumes **CTL**, in the sense that all **CTL** formulae can be recasted in terms of fix-points operations and any **CTL** model corresponds to a model for μ-calculus. In Section 2.2, the model checking algorithm for **CTL** relies on the

[4]For technical reasons, in this syntax negation is allowed only for atomic proposition. This restriction does not affect expressivity; more details can be found in [Kozen, 1983].

[5]Notice that $[\![\cdot]\!]$ depends on T, and thus it should be written $[\![\cdot]\!]_T$. The subscript will be omitted when the transition system is clear from the context.

characterisation of **CTL** operators in terms of fix-point.

2.1.4.3 ATL

Alternating-time Temporal Logic (**ATL**) is a logic introduced in [Alur et al., 1997, Alur et al., 2002] to reason about *strategies* in multi-player games. Given a set of atomic formulae P, the language $\mathcal{L}_{\mathbf{ATL}}$ of **ATL** is defined as follows:

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid \langle\langle\Gamma\rangle\rangle X\varphi \mid \langle\langle\Gamma\rangle\rangle G\varphi \mid \langle\langle\Gamma\rangle\rangle[\varphi U\psi].$$

In the previous expression, Γ is a group of *players*. The formula $\langle\langle\Gamma\rangle\rangle X\varphi$ is read as "group Γ can enforce a next state in which φ holds" or, equivalently, "group Γ has a strategy to enforce Γ in the next state". Similarly, $\langle\langle\Gamma\rangle\rangle G\varphi$ is read as "group Γ has a strategy to enforce a sequence of states in which φ holds globally", and $\langle\langle\Gamma\rangle\rangle[\varphi U\psi]$ is read as "group Γ can enforce a sequence of states in which ψ eventually holds, and φ holds until then". As in the case of **CTL**, the operator F can be used as an abbreviation for $\langle\langle\Gamma\rangle\rangle[\top U\varphi]$.

ATL formulae are interpreted in *concurrent game structures*. A concurrent game structure is a tuple $C = (k, S, \delta, d, V)$, where:

- k is a natural number, the number of players.
- S is a set of states[6].
- δ is a transition function (see below).
- $d : \{1, \ldots, k\} \times S \to I\!N$ is a function assigning a natural number to a player number and a state. Intuitively, this is the number of moves available to a player in a given state.
- $V : S \to 2^P$ is an evaluation function from states to sets of propositions.

Three kinds of concurrent game structures are identified in [Alur et al., 2002], based on the properties of the transition function δ.

- In **turn-based synchronous** game structure "only a single player has a choice of moves" at every time step [Alur et al., 2002]. Formally, for every state $s \in S$, $d(i, s) = 1$ for all players but one. In this case, the function $\delta : S \times I\!N \to S$ assigns a "next" state to a "current" state and a natural number, which represents the move chosen by the moving player.

[6]This set is required to be finite in [Alur et al., 2002].

- In **Moore synchronous** game structures all players evolve simultaneously. The state space S is the Cartesian product of "local" state spaces $S_1, \dots, S_k$, one for each player, i.e., $S = S_1 \times \dots S_k$. A transition function $\delta_i : S \times I\!\!N \rightarrow S_i$, $i = \{1, \dots, k\}$ is defined for every player, assigning a "local" state to a state and to a move identifier for player i. The "global" transition function is defined by $\delta(s, i_1, \dots, i_k) = (\delta_1(q, i_1), \dots, \delta_k(q, i_k))$. In the previous expression, the natural numbers $i_j, (j \in \{1, \dots, k\})$ are the identifier of the moves chosen by each player in a given state, such that $i_j < d_j(s)$ for all $j \in \{1, \dots, k\}$ and $s \in S$.

- In **turn-based asynchronous** games structures, a *scheduler* selects a player to perform a move, and the evolution proceeds similarly to turn-based synchronous systems. The scheduler is modelled using one of the players, usually player k. This thesis is not concerned with turn-based asynchronous systems: more details about this approach can be found in [Alur et al., 2002].

A *strategy* for an agent i is a function f_i assigning a natural number to a non-empty sequence of states $\lambda \in S^+$, with the constraint that if s is the last state of λ, then $f_i(\lambda) \leq d(i, s)$. Intuitively, a strategy determines the moves of a player at any given state, based on the player's history. By following a strategy f_i a player may *enforce* a certain set of computations. Given a state $s \in S$, a set $\Gamma \subseteq \{1, \dots, k\}$ of players, and a set of strategies $F_\Gamma = \{f_i : i \in \Gamma\}$, the set $out(s, F_\Gamma)$ is the set of computations that players in Γ can *enforce*. Based on these definitions, the semantics for **ATL** is defined in [Alur et al., 2002] as follows:

$C, s \models p$	iff	$p \in V(s)$;
$C, s \models \neg\varphi$	iff	$s \not\models \varphi$;
$C, s \models \varphi_1 \vee \varphi_2$	iff	$C, s \models \varphi_1$ or $C, s \models \varphi_2$;
$C, s \models \langle\langle\Gamma\rangle\rangle X\varphi$	iff	there exists a set of strategies F_Γ such that, for all computations $\pi \in out(s, F_\Gamma)$ it is the case that $C, \pi(1) \models \varphi$;
$C, s \models \langle\langle\Gamma\rangle\rangle G\varphi$	iff	there exists a set of strategies F_Γ such that, for all computations $\pi \in out(s, F_\Gamma)$ and for all $j \geq 0$, it is the case that $C, \pi(j) \models \varphi$;
$C, s \models \langle\langle\Gamma\rangle\rangle[\varphi U \psi]$	iff	there exists a set of strategies F_Γ such that, for all computations $\pi \in out(s, F_\Gamma)$ there exists a $j \geq 0$ such that $C, \pi(j) \models \psi$ and, for all $0 \leq k < j$, it is the case that $C, \pi(k) \models \varphi$.

Notice that it is possible to see **ATL** as an extension of the logic **CTL** (see [Alur et al., 2002] for details). Also, see [Goranko and Jamroga, 2004] for various comparisons on semantics for **ATL**.

2.1.5 Many-dimensional modal logics

The logics described in Section 2.1.2 include a single modal operator $\Box$, which may be used to reason about a single stance of a single agent. Richer formalisms are needed when reasoning about multi-agent systems: this section describes *many-dimensional* modal logics, which may provide a formal account of a system of agents and their stances, as illustrated in Section 2.1.7.

Formally, given a natural number $n \in I\!N$, the language $\mathcal{L}_n$ of n-dimensional modal logic is defined over a set of atomic propositions P by the following rule:

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid \Box_1\varphi \mid \ \dots \ \mid \Box_n\varphi.$$

Similarly to the mono-modal case, other operators are introduced in the standard way; in particular, $\Diamond_i\varphi = \neg\Box_i\neg\varphi$.

The axiomatic characterisation of n-dimensional modal logic is analogous to the mono-modal one. An n-dimensional *normal* modal logic $\mathbf{L}$ is a set $\mathbf{L} \subseteq \mathcal{L}_n$ such that:

- $\mathbf{L}$ includes all tautologies of propositional logic.
- $\mathbf{L}$ is closed under *Modus Ponens*, i.e., if $A, A \implies B \in \mathbf{L}$, then $B \in \mathbf{L}$.
- $\mathbf{L}$ contains the n schemata

$$\mathbf{K_i} : \Box_i(A \implies B) \implies (\Box_i A \implies \Box_i B) \ \ (i \in \{1, \dots, n\}).$$

- $\mathbf{L}$ is closed under necessitation, i.e.,

$$\text{if } L \vdash A, \text{ then, for all } i, \ L \vdash \Box_i A.$$

The smallest n-dimensional normal logic is denoted by $\mathbf{K}_n$.

Kripke models can provide a semantics for n-dimensional modal logic, too: a Kripke model for an n-dimensional modal logic is a tuple $M = (W, R_1, \dots, R_n, V)$, where W is a set of possible worlds, $R_i \subseteq W \times W$ ($i \in \{1, \dots, n\}$) are n accessibility relations, and $V : W \to 2^P$ is an evaluation function. Given a Kripke model M, a world w, and a formula φ, satisfaction in a given state is defined as follows:

$M, w \models p$	iff	$w \in V(p)$;
$M, w \models \neg\varphi$	iff	$M, w \not\models \varphi$;
$M, w \models (\varphi \vee \psi)$	iff	$M, w \models \varphi$ or $M, w \models \psi$;
$M, w \models \Box_i\varphi$	iff	for all $w' \in W$, wR_iw' implies $M, w' \models \varphi$.

A frame for an n-modal logic is a tuple $F = (W, R_1, \dots, R_n)$. All the remaining conventions are analogous to the mono-modal case.

2.1.5.1 Combining logics

Multi-modal logics can be constructed by *combining* "smaller" logics in various ways: see [Gabbay et al., 2003] and references therein for the numerous techniques available. For the purposes of this thesis, only the *fusion* (or independent join) of two logics is summarised below.

The fusion of two logics is denoted by $\mathbf{L}_1 \otimes \mathbf{L}_2$. Given two logics $\mathbf{L}_1$ and $\mathbf{L}_2$, their languages $\mathcal{L}_1$ and $\mathcal{L}_2$, and their corresponding axiomatic systems $\mathcal{H}_1$ and $\mathcal{H}_2$, the logic $\mathbf{L}_1 \otimes \mathbf{L}_2$ is the smallest logic with the following characteristics:

- The language $\mathcal{L}_{\mathbf{L}_1 \otimes \mathbf{L}_2}$ is the union of $\mathcal{L}_{\mathbf{L}_1}$ and $\mathcal{L}_{\mathbf{L}_2}$ (it is assumed that $\mathcal{L}_{\mathbf{L}_1}$ and $\mathcal{L}_{\mathbf{L}_2}$ have disjoint sets of modal operators).
- The logic $\mathbf{L}_1 \otimes \mathbf{L}_2$ is axiomatised by the set of axioms $\mathcal{H}_1 \cup \mathcal{H}_2$. Notice that this implies that no "interaction" axiom is required, i.e., there is no axiom involving mixed modalities.

If $\mathbf{L}_1$ and $\mathbf{L}_2$ are interpreted in Kripke frames $F_1 = (W, R_1^1, \ldots, R_n^1)$ and $F_2 = (W, R_1^2, \ldots, R_m^2)$ the semantics for $\mathbf{L}_1 \otimes \mathbf{L}_2$ can be defined in the Kripke frame $F = (W, R_1^1, \ldots, R_n^1, R_1^2, \ldots, R_m^2)$ obtained by the "fusion" of the two frames F_1 and F_2.

Fusions of logics are particularly important because the operation of fusion preserves various property of the original logics. For instance, soundness and completeness are preserved, as well as decidability. In the case of multi-agent systems the operation of fusion allows to extend, in certain circumstances, to a system of agents the results obtained for a single agent.

An example of a logic obtained by fusion is the n-dimensional normal logic $\mathbf{K}_n$, which is the fusion of n copies of the logic $\mathbf{K}$, i.e., $\mathbf{K}_n = \mathbf{K} \otimes \cdots \otimes \mathbf{K}$. Another example of fusion is the logic **CTLK**, which is the fusion of the logic **CTL** with the logic **S5** [Fagin et al., 1995, Penczek and Lomuscio, 2003]: typically, the modal operator of **S5** formalises epistemic concepts. Thus, **CTLK** is a logic used to reason about knowledge and time. Other examples of fusions are provided in Section 2.1.7.

2.1.6 Complexity

This section fixes the notation to reason about the *complexity of decision problems* and it is based on material from [Papadimitriou, 1994]. A decision problem is a problem which requires an answer of the form "yes" or "no". The REACHABILITY problem is an example of a decision problem; given a graph $G = (V, E)$ (where V is a set of vertices and E is a set of edges), and two vertices $v_1, v_2 \in V$, REACHABILITY is the problem of establishing whether or not there is a path from v_1 to v_2[7].

Turing machines offer a uniform framework to reason about the complexity of the algorithms employed in decision problems. A Turing machine operates on a string of symbols (the *tape*) by moving a cursor on the string and by reading/writing/overwriting symbols on the tape at the cursor position. Formally, a Turing machine is a tuple $M = (S, \Sigma, \delta, s)$, where:

- S is a finite set of states;
- Σ is a finite set of symbols, disjoint from S, called the *alphabet* of M. The set Σ includes the special symbols $\sqcup$ and $\triangleright$, denoting a blank symbol and the "first" symbol;
- $\delta : S \times \Sigma \to (S \cup \{\text{yes,no},h\} \times \Sigma \times \{\leftarrow, \rightarrow, -\}$ is a transition function. The states $\{\text{yes,no},h\}$ are special *halting* states of M, and the symbols $\{\leftarrow, \rightarrow, -\}$ denote cursor directions. The function δ is the *program* of the machine;
- $s \in S$ is the initial state of M.

If, given an input string $x \in (\Sigma \backslash \sqcup)^*$, a machine M halts in any of the halting states $\{\text{yes,no}\}$, then it is said that the machine has *halted.* If a machine halts in a "yes" state, then it is said that the machine *accepts* the input x and, by convention, it is written $M(x) = \text{yes}$. If a machine halts in a "no" state, then it is said that the machine *rejects* the input x and it is written $M(x) = \text{no}$. If the machine halts in the "h" state, the output $M(x)$ of M is defined to be the string on the tape at the moment of halting. If a machine does not halt on input x it is written $M(x) = \nearrow$.

A language $L \subseteq \Sigma^*$ is *decided* by a Turing machine M if, for all strings $x \in L$, $M(x) = \text{yes}$ and, for all strings $y \notin L$, $M(x) = \text{no}$. If a language L is decided by some Turing machine, then L is said to be *recursive.*

A language $L \subseteq \Sigma^*$ is *accepted* by a Turing machine M if, for all strings $x \in L$, $M(x) = \text{yes}$ (notice that M is not required to halt when $x \notin L$). If a language L is accepted by some Turing machine, then L is said to be *recursively enumerable.*

Turing machines can be generalised to *multi-string* Turing machines: a k-string Turing machine (where $k \geq 1$ is an integer) is a tuple $M = (S, \Sigma, \delta, s)$, where S and Σ are as above, and the transition function δ takes into account the k strings of M. Formally,

[7] Notice that, in some classical examples, a problem is not presented as a decision problem. This is the case, for instance, with the "travelling salesman problem" (TSP). Such problems, however, can be presented in a decisional form. As an example, the decisional version of TSP is obtained by providing a bound B on the length of the tour of the salesman, and by asking whether or not there exists a tour of length B at most.

$\delta : S \times \Sigma^k \rightarrow (S \cup \{\text{yes},\text{no},h\} \times (\Sigma \times \{\leftarrow, \rightarrow, -\})^k$. Intuitively, δ prescribes the next symbol and the next movement of the cursor for each string. At the start of each run all strings start with the symbol $\triangleright$, and the first string contains the input. The output of a k-string Turing machine is stored in the last string. A configuration of a k-string Turing machine is a $(2k + 1)$ tuple $(s, \sigma_1, \sigma'_1, \ldots, \sigma_k, \sigma'_k)$, defined as for single tape Turing machines.

The *time required to halt* by a (multi-string) Turing machine M on input x is defined as the *number of steps* from the initial state to the halting state. If $M(x) = \nearrow$, then the time is $+\infty$. A machine M *operates in time* $f(n)$ where $f(n)$ is a function $f : I\!N \rightarrow I\!N$ if, for any string $x \in \Sigma^*$, the time required by M on input x is at most $f(|x|)$. A language L *belongs to the complexity class* TIME($f(n)$) if L *is decided by a multi-string Turing machine operating in time* $f(n)$. Thus, a time complexity class is the set of languages that can be decided within a certain time bound.

A k-string Turing machine *with input and output* is a standard k-string Turing machine with the restriction that the first string (the input string) is a read-only string, and the last string (the output string) is a write-only string. Given a k-string Turing machine M with input and output, suppose that M halts in the configuration $(s, \sigma_1, \sigma'_1, \ldots, \sigma_k, \sigma'_k)$ on input x. The *space required by* M *on input* x is defined as $\sum_{i=2}^{k-1} |\sigma_i \sigma'_i|$, i.e., the space required is the sum of the lengths of all the strings *excluding* the input and the output string.

A language L *belongs to the complexity class* SPACE(f(n)) if L is decided by a k-string Turing machine with input and output operating in space $f(n)$.

A *non-deterministic* Turing machine is a tuple $M = (S, \Sigma, \Delta, s)$, where S, Σ and s are as in standard Turing machines, and Δ is a transition *relation*: $\Delta \subseteq S \times \Sigma \times (S \cup \{\text{yes},\text{no},h\} \times \Sigma \times \{\leftarrow, \rightarrow, -\}$. Notice that, for each configuration, there may be *more than one possible "next" configuration.* Non-deterministic Turing machines can be generalised to multi-string non-deterministic Turing machines, using a method similar to that for standard Turing machines.

Non-deterministic Turing machines differ from deterministic machines with respect to the definition of complexity classes. A non-deterministic Turing machine M is said to *decide* language L if, given $x \in L$, $M(x) = \text{yes}$ *for some possible computation of* M. Notice that M is not required to accept x in all possible computations.

A non-deterministic Turing machine M decides a language L in time $f(n)$ if (i) M decides L and (ii) M does not have computation paths longer than $f(n)$ (where n is the size of the input). The set of languages decided by a non-deterministic Turing machine within time $f(n)$ is denoted by NTIME(f(n)). The complexity class NSPACE(f(n)) is defined analogously to SPACE(f(n)). Table 2.4 lists the definition of some commonly used complexity classes and their names.

Name	**Definition**
L	$\bigcup$SPACE($log(n)$)
NL	$\bigcup$NSPACE($log(n)$)
P	$\bigcup$TIME(n^k)
NP	$\bigcup$NTIME(n^k)
PSPACE	$\bigcup$SPACE(n^k)
NPSPACE	$\bigcup$NSPACE(n^k)
EXP	$\bigcup$TIME(2^{n^k})

Table 2.4: Some complexity classes and their definitions.

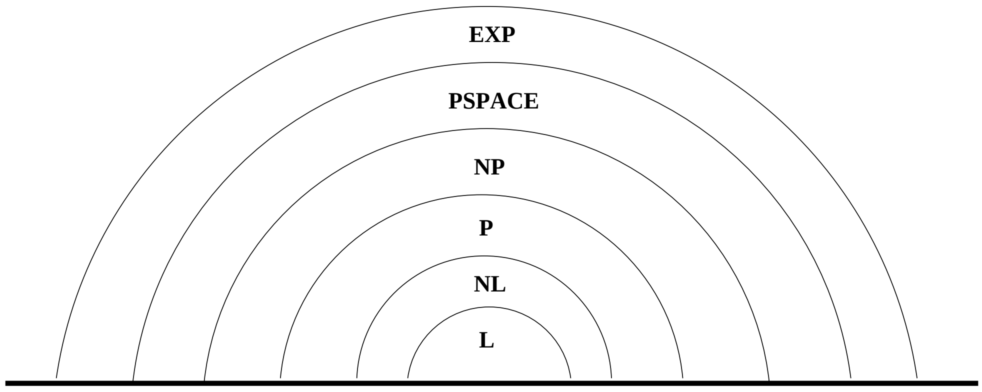

Figure 2.2: Complexity classes (from [Papadimitriou, 1994].)

A problem P_1 is *reducible* to a problem P_2 if there exists a transformation T from strings to strings, converting any input x for P_1 to an input for P_2, denoted by $T(x)$ and such that $P_1(x) = P_2(T(x))$. It is required that the transformation T is computable by a Turing machine belonging to the complexity class $\mathbf{L}$[8]. If a problem P_1 is reducible to a problem P_2, then P_2 is said to be *as hard as* P_1. Given a complexity class $\mathbf{C}$ and a problem $P \in \mathbf{C}$[9], P is said to be $\mathbf{C}$-*complete* if any problem $P' \in \mathbf{C}$ can be reduced to P. It is possible to establish a hierarchy for complexity classes, comparing both time and space classes. Figure 2.2 from [Papadimitriou, 1994] depicts graphically the following sequence of inclusions:

$$\mathbf{L} \subseteq \mathbf{NL} \subseteq \mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{PSPACE} \subseteq \mathbf{EXP}.$$

It is known that **L** is a proper subset of **PSPACE**, and that **P** is a proper subset of **EXP**. However, it is an open question which of the remaining inclusions is proper.

Given a language L, the *complement* of L is the language $\bar{L} = \Sigma^* \backslash L$. Given a complexity class **C**, the *complement* of **C**, denoted by **co-C**, is the set of languages defined by **co-C** $= \{\bar{L} : L \in \mathbf{C}\}$. Notice that, for any deterministic complexity class **C**, **co-C** =

[8]This implies that the machine computing T operates in polynomial time – see below.

[9]Notice that Turing machines can be seen as algorithms to solve *problems*. Thus, a problem P is in a complexity class **C** if the Turing machine implementing an algorithm for P is in **C**.

C. Important results about complexity classes are summarised in the following theorems from [Papadimitriou, 1994]. These theorems will be employed in Section 4.2 to prove complexity bounds for the complexity of model checking multi-agent systems.

Theorem 2.1.1. (Savitch's Theorem, [Papadimitriou, 1994], p.149) *The problem of* REACHABILITY *belongs to the complexity class* $\text{SPACE}(\log^2(n))$.

Theorem 2.1.2. ([Papadimitriou, 1994], p.150) *The following inclusion holds:* $\text{NSPACE}(f(n)) \subseteq \text{SPACE}(f^2(n))$.
As a corollary, notice that **NPSPACE** = **PSPACE**.

Theorem 2.1.3. ([Papadimitriou, 1994], p.153) *The following equivalence holds:* $\text{NSPACE}(f(n)) = \text{co} - \text{NSPACE}(f(n))$.

2.1.6.1 The complexity of modal logics

The machinery presented above may be used in the definition of the *complexity of a logic.* Traditionally, the complexity of a logic is defined as the complexity of the *satisfiability* problem for that logic. Given a formula φ, satisfiability is the problem of establishing whether or not there exists a model M and a world w of M such that $M, w \models \varphi$. Notice that a complexity result for this problem gives an immediate result for the problem of *validity* of a formula φ, as it is defined in Section 2.1.1: indeed, a formula φ is valid iff $\neg\varphi$ is *not* satisfiable.

The following theorems summarise complexity results for various logics:

Theorem 2.1.4. ([Blackburn et al., 2001]) *Every normal modal logic extending* **S4.3** *has an NP-complete satisfiability problem. Every normal logic between* **K** *and* **S4.3** *has a PSPACE-hard satisfiability problem*[10].

Theorem 2.1.5. ([Emerson and Halpern, 1985, Sistla and Clarke, 1985]) *The satisfiability problem for* **CTL** *is* **EXP***-complete. The satisfiability problem for* **LTL** *is* **PSPACE***-complete.*

Theorem 2.1.6. ([van Drimmelen, 2003]) *The satisfiability problem for* **ATL** *is* **EXP***-complete.*

2.1.7 MAS theories

Many different formalisms are available for reasoning about multi-agent systems using logics; detailed reviews can be found in [Wooldridge and Jennings, 1995, Hoek and Wooldridge, 2003b]. This section provides a brief introduction to some formalisms, while

[10]**S4.3** is the logic obtained by adding the axiom $\Box(\Box p \implies q) \vee \Box(\Box q \implies p)$ to the logic **S4**.

Section 2.1.7.1 introduces the details of the framework of *interpreted systems* [Fagin et al., 1995].

Cohen and Levesque's intention logic: The key assumption of Cohen and Levesque is that intelligent agents must achieve a *rational balance* between beliefs, goals, and intentions ([Cohen and Levesque, 1990], p.214). To this end, they introduce a first order multi-modal logic with four primary operators: BEL, GOAL, HAPPENS and DONE ([Cohen and Levesque, 1990], p.222). The semantics of BEL and GOAL is the usual Kripke semantics; the accessibility relation for BEL is Euclidean, transitive and serial; the accessibility relation for GOAL is serial. Moreover, the GOAL relation is a subset of the BEL relation. Worlds in the formalism are an infinite sequence of events.

Besides the temporal operators HAPPENS and DONE, there are other constructs similar to dynamic logic [Harel, 1984], such as ";" to denote a sequence of events and "?" to denote a test action.

The standard temporal operators $\Box$ ("always") and $\Diamond$ ("at some time") are defined as abbreviations:

$$\Diamond\varphi = \exists x(\text{HAPPENS } x; \varphi?); \quad \Box\varphi = \neg\Diamond\neg\varphi.$$

Other constructs are derived from the basic operators; the most important is *persistent* goal[11]:

$$\begin{aligned}(\text{P-GOAL } i\ p) \quad = \quad & (\text{GOAL } i\ (\text{LATER } p)) \wedge \\ & (\text{BEL } i\ \neg p) \wedge \\ & [\text{BEFORE}((\text{BEL } i\ p) \vee (\text{BEL } i\ \Box\neg p)) \\ & \neg(\text{GOAL } i\ (\text{LATER } p))];\end{aligned}$$

which means that an agent i has p as a persistent goal if: i has a goal that p becomes true at some point in the future, and i believes that p is currently false, and i drops his goal only if i believes that the goal has been satisfied, or i believes that the goal will never be satisfied.

Intentions to act are defined as follows[12]:

$$\begin{aligned}(\text{INTEND } i\ \alpha) \quad = \quad & (\text{P-GOAL } i \\ & [\text{DONE } i\ (\text{BEL } i\ (\text{HAPPENS } \alpha))?; \alpha]).\end{aligned}$$

Notice that an agent drops an intention of doing an action only if the agent believes that the action has been performed, or the agent believes that the action cannot be performed.

[11] The definition of LATER and BEFORE is straightforward and can be found in the original paper.

[12] A similar definition for "intending that something becomes true" can be found in [Cohen and Levesque, 1990].

Condition	Axiom
$\mathcal{B} \subseteq_{sup} \mathcal{D} \subseteq_{sup} \mathcal{I}$	(INTEND i $E(\varphi)$) $\implies$ (DES i $E(\varphi)$)(BEL i $E(\varphi)$)
$\mathcal{B} \subseteq_{sub} \mathcal{D} \subseteq_{sub} \mathcal{I}$	(INTEND i $A(\varphi)$) $\implies$ (DES i $A(\varphi)$)(BEL i $A(\varphi)$)
$\mathcal{B} \subseteq \mathcal{D} \subseteq \mathcal{I}$	(INTEND i φ) $\implies$ (DES i φ)(BEL i φ)
$\mathcal{B} \cap \mathcal{D} \neq \emptyset$	(BEL i φ) $\implies$ $\neg$(DES i $\neg\varphi$)
$\mathcal{D} \cap \mathcal{I} \neq \emptyset$	(DES i φ) $\implies$ $\neg$(INTEND i $\neg\varphi$)
$\mathcal{B} \cap \mathcal{I} \neq \emptyset$	(BEL i φ) $\implies$ $\neg$(INTEND i $\neg\varphi$)

Table 2.5: Interaction conditions and corresponding axioms for BDI logics.

Rao and Georgeff's BDI logic: (This presentation follows the lines of [Hoek and Wooldridge, 2003b]). Rao and Georgeff propose a family of BDI logics (BDI stands for Beliefs, Desires, Intentions) based on the branching time temporal logic **CTL**. Their logics include the modal operators BEL, DES and INTEND for expressing beliefs, desires and intentions. Beliefs correspond to information that an agent has about the world. Desires correspond to states of affairs that an agent would like to achieve. Intentions correspond to desires that an agent is *committed* to achieve.

The semantics of BDI modalities is based on the standard Kripke semantics. However, each world is itself a Kripke structure for **CTL** logic. Hence, a world is a structure $w =< T, R >$ where T is a non-empty set of time points and R is a branching time relation on T. A *situation* is a pair $< w, t >$ composed of a world and a time point. The accessibility relations $\mathcal{B}, \mathcal{D}, \mathcal{I}$ for BEL, DES and INTEND are defined on situations. The logics proposed by Rao and Georgeff differ on the the interactions between modalities. Interaction between relations correspond to axioms in the logic. For example, if $\mathcal{D} \subseteq \mathcal{I}$, then for every agent i, INTEND i $\varphi \implies$ DES i φ.

But worlds are themselves structures, so one can also reason about interactions on the structure of worlds. If w and w' are worlds, $w \sqsubseteq w'$ means that w has the same structure as w', but fewer paths. Consider now two accessibility relation R and R'. R is a *structural subset* of R', denoted by $R \subseteq_{sub} R'$, if for every R-accessible world w, there is an R'-accessible world w' such that $w \sqsubseteq w'$. Similarly, R is a *structural superset* of R', denoted by $R \subseteq_{sup} R'$, if $w' \sqsubseteq w$.

Various BDI logical systems can be obtained from the interactions between relations. Examples are reported in Table 2.5.

Benerecetti, Giunchiglia and Serafini's MATL: Multi-Agent Temporal Logic [Benerecetti et al., 1998] is the composition of the temporal logic **CTL** and the logic **HML** (Hierarchical Meta-Logic) to represent beliefs, desires and intentions.

HML is defined as follows. Let I be a set of agents, and $O = \{B, D, I\}$ be a set of symbols, one for each attitude. Let $OI^* = (O \times I)^*$, i.e., each $\alpha \in OI^*$ is a string representing a possible nesting of attitudes. Each $\alpha \in OI^*$ is called a *view*, including the empty string

ϵ representing the view of an "external observer". An agent "is a tree rooted in the view that the external observer has of it" (notice that the view that an agent has of another agent can be different from the agent itself). A logical language L_α is associated to each view α. Each language is used to express what is true in the representation corresponding to α. It is imposed that $O_i\varphi$ is a formula of L_α iff φ is a formula of $L_{O_i\alpha}$.

The semantics of $\{L_\alpha\}_{\alpha\in OI^*}$ is given by means of the concept of *tree*. A tree is a subset of the set of possible interpretations of a language L_α, denoted by M_α. Namely, each interpretation is denoted by $t_\alpha \in M_\alpha$, and a tree is a set $\{t_\alpha\}_{\alpha\in OI^*}$. A *compatibility relation* T is a set of trees. A tree satisfies a formula at a view iff the formula is satisfied by all the elements that the tree associates to the view.

A *Hierarchical Meta-Structure* (HM Structure) is a set of trees T on L_α, closed under containment, such that there is a $t \in T$ with $t_\epsilon \neq \emptyset$; if t_α satisfies $O_i\varphi$, then $t_{O_i\alpha}$ satisfies φ, and if for all $t' \in T$, $t'_\alpha \in t_\alpha$ implies that $t_{\alpha O_i}$ satisfies φ , then t_α satisfies $O_i\varphi$.

MATL structures (i.e., models) are a particular kind of HM structures: each language L_α is a **CTL** language. This allows for the interpretation of formulae of a language that includes BDI and temporal (**CTL**) operators.

Wooldridge's LORA: LORA [Wooldridge, 2000b] can be viewed as an extension of the temporal logic **CTL**. LORA has four main components: a classical first-order component, a BDI component, a temporal component and an action component (in a dynamic logic style).

The BDI component is similar to the Rao and Georgeff's formalism presented above. The state of an agent is defined by its beliefs, desires and intentions, whose semantics is given via standard Kripke semantics, and worlds are themselves branching time structures. LORA also contains terms to reason about *groups* of agents.

The semantics of LORA is defined by means of models. A model for LORA is a structure

$$M =< T, R, W, D, Act, Agt, \mathcal{B}, \mathcal{D}, \mathcal{I}, C, \Phi >$$

where T is the set of all time points, $R \subseteq T \times T$ is a branching time relation over T, W is a set of worlds over T (see above); $D =< D_{Ag}, D_{Ac}, D_{Gr}, D_U >$ is a domain, $Act : R \to D_{Ac}$ associates an action with every relation in R, $Agt : D_{Ac} \to D_{Agt}$ associates an agent with every action, $\mathcal{B}, \mathcal{D}, \mathcal{I}$ are the accessibility relations, C is an interpretation function for constants and Φ is an interpretation function for predicates. In the definition of D, $D_{Ag} = \{1, \dots, n\}$ is a set of agents, $D_{Ac} = \{\alpha, \alpha', \dots\}$ is a set of actions, D_{Gr} is a set of non-empty subsets of D_{Ag}, i.e., groups of agents, D_U is a set of other individuals. LORA models provide the semantics for state and path formulae. Details of LORA's syntax and the interpretation of LORA's formulae can be found in [Wooldridge, 2000b].

2.1.7.1 Interpreted systems

The formalism of *interpreted systems* was introduced in [Fagin et al., 1995] to model a system of agents and to reason about the agents' epistemic and temporal properties. In this formalism, each agent is modelled using a set of *local states*, a set of *actions*, a *protocol*, and an *evolution function*.

- The set of local states for an agent i is denoted by the symbol L_i. Elements of L_i capture the "private" information of an agent and, at any given time, local states represent the state in which an agent is (e.g. `ready` and `busy` may be elements of L_i). Contrary to [Fagin et al., 1995], it is assumed that the set L_i is finite (this is required by the model checking algorithms).
- The set of actions for an agent i is denoted by the symbol Act_i. Elements of Act_i represent the possible actions that an agent is allowed to perform. Differently from local states, actions are "public". Similarly to local states, here the set Act_i is assumed to be finite.
- The protocol for an agent i is denoted by the symbol P_i. The protocol is a "rule" establishing which actions may be performed in each local state. The protocol P_i is modelled by a function $P_i : L_i \to 2^{Act_i}$, assigning a set of actions to a local state. Intuitively, this set corresponds to the actions that are enabled in a given local state. Notice that this definition may enable more than one action to be performed for a given local state. When more than one action is enabled, it is assumed that an agent selects *non-deterministically* which action to perform.
- The evolution function for agent i is denoted by the symbol t_i (notice: [Fagin et al., 1995] define a single evolution function t for all the agents, see discussion below). The evolution function determines how local states "evolve", based on the agent's local state, on other agents' actions, and on the local state of a special agent used to model the environment (see below). The evolution function is modelled by a function $t_i : L_i \times L_E \times Act_1 \times \cdots \times Act_n \times Act_E \to L_i$, where n is the number of agents in the system.

A special agent E is used to model the environment in which the agents operate. Similarly to the other agents, E is modelled using a set of local states L_E, a set of actions Act_E, a protocol P_E, and an evolution function t_E. As noticed above, local states for E are "public": all the remaining agents may "peek" at L_E to determine their temporal evolution.

For all agents including the environment, the sets L_i and Act_i are assumed to be non-empty, and the number $n \in \mathbb{N}$ of agents is assumed to be finite. For convenience, the symbol Act denotes the Cartesian product of the agents' actions, i.e., $Act = Act_1 \times \cdots \times Act_n \times Act_E$.

An element $\alpha \in Act$ is a tuple of actions (one for each agent) and is referred to as a *joint action.* The Cartesian product of the agents' local states is denoted by S, i.e., $S = L_i \times \cdots \times L_n \times L_E$. An element $g \in S$ is called a *global state*; given a global state g, the symbol $l_i(g)$ denotes the local state of agent i in the global state g. It is assumed that, in every state, agents evolve *simultaneously* (notice that this requirement is similar to the definition of Moore synchronous game structures given in Section 2.1.4.3).

The definition of a single evolution function $t : S \times Act \rightarrow S$ presented in [Fagin et al., 1995] differs slightly from the definition of $n + 1$ evolution functions presented here. The two definitions are, in fact, equivalent: $t(g, a) = g'$ iff, for all $i \in \{1, \dots, n\}, t_i(l_i(g), a) = l_i(g')$ and $t_E(l_E(g), a) = l_E(g')$ (the decomposition from a single t to $n + 1$ "local" transition functions is guaranteed to be possible by the assumptions on t). As it will be clear in Section 5.2, the definition of an evolution function for each agent helps to keep the description of the system compact.

Given a set of *initial global states* $I \subseteq S$, the protocols and the evolution functions generate a set of *reachable global states* $G \subseteq S$, obtained by all the possible runs of the system. A set of atomic propositions P and an *evaluation relation* $V \subseteq P \times S$ are introduced to complete the description of an interpreted system. Formally, given a set of n agents $\{1, \dots, n\}$, an interpreted system is a tuple:

$$IS = \left\langle (L_i, Act_i, P_i, t_i)_{i \in \{1,\dots,n\}}, (L_E, Act_E, P_E, t_E), I, V \right\rangle.$$

It has been shown in [Fagin et al., 1995] that interpreted systems can provide a semantics to reason about time and epistemic properties, by means the following language:

$$\varphi \ ::= \ p \mid \neg\varphi \mid \varphi \vee \varphi \mid EX\varphi \mid EG\varphi \mid E[\varphi U \psi] \mid \ K_i\varphi \mid E_\Gamma\varphi \mid C_\Gamma\varphi \mid D_\Gamma\varphi.$$

In this grammar, $p \in P$ is an atomic proposition, and the operators EX, EG, and EU are the standard **CTL** operators; the remaining **CTL** operators EF, AX, AG, AU, AF can be derived in the standard way presented above. The formula $K_i\varphi$ ($i \in \{1, \dots, n\}$) is read as "agent i knows φ". The symbol Γ denotes a group of agents. The formula $E_\Gamma\varphi$ is read as "everybody in group Γ knows φ"; the formula $C_\Gamma\varphi$ is read as "φ is *common knowledge* in group Γ" (intuitively, common knowledge of φ in a group of agents denotes the fact that everyone knows φ, and everyone knows that everybody else knows φ); the formula $D_\Gamma\varphi$ is read as "φ is *distributed* knowledge in group Γ" (intuitively, distributed knowledge in a group of agents is the knowledge obtained by "sharing" all agents' knowledge).

Given an interpreted system IS, it is possible to associate a Kripke model $M_{IS} = (W, R_t, \sim_1, \dots, \sim_n, V)$ to IS; the model M_{IS} can be used to interpret formulae of the grammar above. The model M_{IS} is obtained as follows:

- The set of possible worlds W is the set G of reachable global states (this is to avoid the epistemic accessibility of states which cannot reached using the temporal relation).

- The temporal relation $R_t \subseteq W \times W$ relating two worlds (i.e., two global states) is defined by the temporal transition t_i. Two worlds w and w' are such that wR_tw' iff there exists a joint action $a \in Act$ such that $t(g,a) = g'$, where t is the transition relation of IS obtained by the composition of the functions t_i, $i \in \{1,\dots,n\}$ and t_E.

- The epistemic accessibility relations $\sim_i \subseteq W \times W$ are defined by imposing the equality of the local components of the global states. Two worlds $w, w' \in W$ are such that $w \sim_i w'$ iff $l_i(w) = l_i(w')$ (i.e., two worlds w and w' are related via the epistemic relation $\sim_i$ when the local states of agent i in global states w and w' are the same [Fagin et al., 1995]).

- The evaluation relation $V \subseteq AP \times W$ is the evaluation relation of IS.

Similarly to the definitions of Section 2.1.4, let $\pi = (w_0, w_1, \dots)$ be an infinite sequence of worlds such that, for all i, $w_iR_tw_{i+1}$, and let $\pi(i)$ denote the i-th world in the sequence (the temporal relation is assumed to be serial and thus all computation paths are infinite). Let $R^E_\Gamma \subseteq W \times W$ denote the relation obtained by taking the union of the epistemic relations for the agents in Γ, i.e., $R^E_\Gamma = \bigcup_{i\in\Gamma} \sim_i$. Let R^D_Γ denote the intersection of the epistemic relations for the agents in Γ, i.e., $R^D_\Gamma = \bigcap_{i\in\Gamma} \sim_i$. Let R^C_Γ denote the transitive closure of R^E_Γ. It is written $M_{IS}, w \models \varphi$ when a formula φ is true at a world w in the Kripke model M_{IS}, associated with an interpreted system IS. Satisfaction is defined inductively as follows:

$$
\begin{array}{lll}
M_{IS}, w \models p & \text{iff} & (p, w) \in V, \\
M_{IS}, w \models \neg\varphi & \text{iff} & M_{IS}, w \not\models \varphi, \\
M_{IS}, w \models \varphi_1 \vee \varphi_2 & \text{iff} & M_{IS}, w \models \varphi_1 \text{ or } M_{IS}, w \models \varphi_2, \\
M_{IS}, w \models EX\varphi & \text{iff} & \text{there exists a path } \pi \text{ such that } \pi(0) = w, \\
 & & \text{and } M_{IS}, \pi(1) \models \varphi, \\
M_{IS}, w \models EG\varphi & \text{iff} & \text{there exists a path } \pi \text{ such that } \pi(0) = w, \\
 & & \text{and } M_{IS}, \pi(i) \models \varphi \text{ for all } i \geq 0, \\
M_{IS}, w \models E[\varphi U \psi] & \text{iff} & \text{there exists a path } \pi \text{ such that } \pi(0) = w, \text{ and there exists} \\
 & & k \geq 0 \text{ such that } M_{IS}, \pi(k) \models \psi, \text{ and } M_{IS}, \pi(j) \models \varphi \\
 & & \text{for all } 0 \leq j < k, \\
M_{IS}, w \models K_i\varphi & \text{iff} & \text{for all } w' \in W,\ w \sim_i w' \text{ implies } M_{IS}, w' \models \varphi, \\
M_{IS}, w \models E_\Gamma\varphi & \text{iff} & \text{for all } w' \in W,\ wR^E_\Gamma w' \text{ implies } M_{IS}, w' \models \varphi, \\
M_{IS}, w \models C_\Gamma\varphi & \text{iff} & \text{for all } w' \in W,\ wR^C_\Gamma w' \text{ implies } M_{IS}, w' \models \varphi, \\
M_{IS}, w \models D_\Gamma\varphi & \text{iff} & \text{for all } w' \in W,\ wR^D_\Gamma w' \text{ implies } M_{IS}, w' \models \varphi.
\end{array}
$$

Similarly to standard Kripke models, a formula φ is *true in a model*, written $M_{IS} \models \varphi$, if $M_{IS}, w \models \varphi$ for all $w \in W$.

A formula φ *is true in an interpreted system IS*, denoted by $IS \models \varphi$, iff it is true in the associated Kripke model ([Fagin et al., 1995], p. 111). In the remainder of this thesis, **CTLK** will denote the logic including the temporal operators of **CTL** and the epistemic operators K_i, while $\mathbf{CTLK}^{D,C}$ will denote the logic **CTLK** with distributed and common knowledge. The complexity of the satisfiability problem for **CTLK** and for $\mathbf{CTLK}^{D,C}$ has been investigated in [Meyden and Wong, 2003], where it has been proven to be **EXP**-complete for both logics.

2.1.7.2 Deontic interpreted systems

Interpreted systems have been extended in [Lomuscio and Sergot, 2003] to include the notion of *correct behaviour*. This is done by partitioning the set of local states L_i into two sets: a non-empty set G_i of allowed (or correct, or "green") states, and a set R_i of disallowed (or faulty, or "red") states, such that $L_i = G_i \cup R_i$, and $G_i \cap R_i = \emptyset$. Given a set of agents $\{1, \ldots, n\}$ and a set of atomic propositions P, a *deontic interpreted system*[13] is a tuple

$$DIS = \left\langle (G_i, R_i, Act_i, P_i, t_i)_{i \in \{1,\ldots,n\}}, (G_E, R_E, Act_E, P_E, t_E), I, V \right\rangle.$$

Two new logical operators are introduced in [Lomuscio and Sergot, 2003]:

- The operator $O_i\varphi$ expresses the fact that, *under all the correct alternatives for agent i, φ holds.*
- The operator $\hat{K}_i^{\Gamma}$, where $\Gamma \subseteq \{1, \ldots, n\}$ is a group of agents, expresses the knowledge that agent i has *on the assumption that all agents in Γ are functioning correctly.*

With slight abuse of notation, it is usually written $\hat{K}_i^j$ when Γ is a singleton $\Gamma = \{j\}$.

Though temporal operators are not considered in [Lomuscio and Sergot, 2003], they can be included in the syntax of formulae evaluated in deontic interpreted systems [Raimondi and Lomuscio, 2004a]. The syntax is defined as follows:

$$\begin{aligned}\varphi \;::=\;& p \mid \neg\varphi \mid \varphi \vee \varphi \mid EX\varphi \mid EG\varphi \mid E[\varphi U \psi] \mid \;K_i\varphi \mid E_\Gamma\varphi \mid C_\Gamma\varphi \mid D_\Gamma\varphi \\ & O_i\varphi \mid \;\hat{K}_i^{\Gamma}\,\varphi.\end{aligned}$$

Formulae are interpreted in deontic interpreted systems DIS by associating a Kripke model $M_{DIS} = (W, R_t, \sim_1, \ldots, \sim_n, R_1^O, \ldots, R_n^O, V)$ to DIS. The definition of the relations $R_i^O \subseteq W \times W$ ($i \in \{1, \ldots, n\}$) is based on the set G_i for agent i: two

[13]Notice that the term "deontic" is used in [Lomuscio and Sergot, 2003] without any reference to *obligations* of agents, but with the aim of reasoning exclusively about *correct functioning* behaviour.

worlds $w, w' \in W$ are such that $wR_i^O w'$ iff $l_i(w') \in G_i$ (notice that R_i^O *does not* depend on the world w appearing on the left hand side of the relation). The definition of all the remaining symbols is analogous to the definitions presented in Section 2.1.7.1, as well as the definition of the semantics for formulae, except for:

$M_{DIS}, w \models O_i\varphi$ iff for all $w' \in W$, $wR_i^O w'$ implies $M_{DIS}, w' \models \varphi$;

$M_{DIS}, w \models \hat{K}_i^{\Gamma} \varphi$ iff for all $w' \in W$ and for all $j \in \Gamma$, $w \sim_i w'$ and $wR_j^O w'$ implies $M_{DIS}, w' \models \varphi$.

Similarly to Section 2.1.7.1, a formula φ *is true in a deontic interpreted system DIS* iff it is true in the associated Kripke model. In the remainder of this thesis, the logic which includes temporal, epistemic, and correct behaviour operators is denoted by $\mathbf{CTLKD}^{D,C}$.

2.1.7.3 Reasoning about actions in interpreted systems

The idea of reasoning about knowledge and actions goes back to [Moore, 1990], and it has been investigated actively in recent years. Actions are treated explicitly in interpreted systems, but the logic languages introduced in [Fagin et al., 1995] do not include operators to reason about actions, neither in a dynamic logic style [Harel, 1984], nor *à la* **ATL**. This section presents a possible approach to the analysis of actions and strategies in interpreted systems.

A framework to reason about knowledge and actions in multi-agent systems has been investigated in [Hoek and Wooldridge, 2003a], where the logic **ATEL** (Alternating-time Temporal Epistemic Logic) is introduced. The language of **ATEL** is the fusion of the languages of **ATL** and $\mathbf{CTLK}^{D,C}$; **ATEL** formulae are interpreted in *alternating epistemic transition systems* (ATES). An ATES is a tuple

$$(\Pi, \Sigma, Q, \sim_1, \ldots, \sim_n, \pi, \delta)$$

such that:

- Π is a set of atomic propositions;
- $\Sigma = \{1, \ldots, n\}$ is a set of agents;
- Q is a finite set of states;
- $\sim_i \subseteq Q \times Q$ ($i \in \{1, \ldots, n\}$) are epistemic accessibility relations (one for each agent);
- $\pi : Q \to 2^{\Pi}$ is an evaluation function;
- $\delta : Q \times \Sigma \to 2^{2^Q}$ is an evolution function.

The definition of satisfiability for **ATEL** is obtained by taking the union of the rules for **ATL** with the standard rules for epistemic operators.

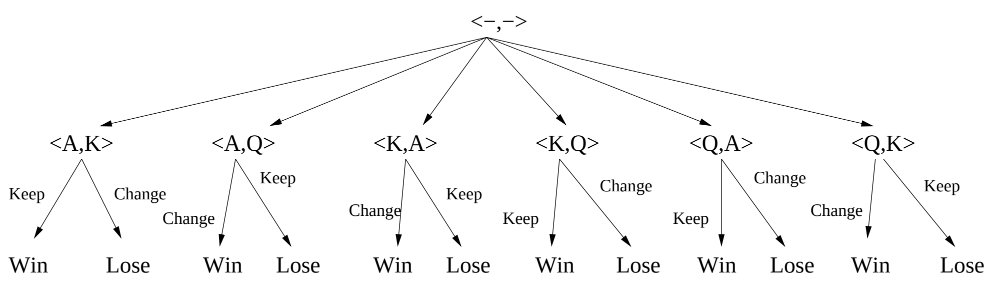

Figure 2.3: A simple card game for **ATEL**.

Intuitively, ATES provide a "coarser grain" semantics than interpreted systems (i.e., ATES can be *embedded* [Goranko and Jamroga, 2004] in interpreted systems): thus, **ATEL** formulae may be evaluated in interpreted systems, and **ATEL** can be seen as the fusion of the two logics **ATL** and $\mathbf{CTLK}^{D,C}$, without *interaction axioms* between knowledge and strategic operators. However, it has been argued [Jamroga, 2004a, Jonker, 2003, Jamroga, 2004b, Jamroga and van der Hoek, 2004] that the interpretation of **ATL** operators in **ATEL** might not correspond entirely to the original spirit of ATL [Alur et al., 2002]. The following example from [Jonker, 2003, Jamroga, 2004b] illustrates this. An agent (the player) plays a simple card game against another agent (the environment). There are just three cards in the deck: Ace ("A"), King ("K)", and Queen ("Q"). "A" wins over "K", "K" wins over "Q", and "Q" wins over "A". In the initial state no cards are distributed. In the first step, the environment gives a card to the player and takes a card for itself. In the second step, the player can either keep its card, or change it. The game is depicted in Figure 2.3.

The formula $\langle\langle \text{player} \rangle\rangle F(\text{win})$, expressing that the player has a strategy to reach a state in which win holds, is true in the initial state of this model. Indeed, the player may *guess* an action to bring about a winning state, but it is clearly not the case that the player can *always enforce* a win: the player cannot distinguish between the "global" states `<A,K>` and `<A,Q>`, and thus cannot *always* choose the right action (either to keep or to change). As originally remarked in [Moore, 1990], some form of *dependence* must be taken in account for actions and knowledge. However, it has been shown in [Agotnes, 2005] that *there is no interaction axiom* that can be added to **ATEL** to express with **ATL** operators what an agent can enforce.

Various solutions have been put forward to express **ATL** operators in a semantics based on MAS [Jonker, 2003, Jamroga, 2004b, Jamroga and van der Hoek, 2004]. Instead of exploring new logics, this thesis employs the language of **ATEL** extended with operators for correct behaviour, and presents three classes of interpreted systems in which formulae this language can be interpreted[14]. This logic will be denoted by $\mathbf{CTLKD} - \mathbf{A}^{D,C}$; formally,

[14]This approach is similar to the one proposed in [Fagin et al., 1995] for the interpretation of the same

the language of $\mathbf{CTLKD} - \mathbf{A}^{D,C}$ is defined by:

$$\begin{aligned}\varphi \ ::= \ & p \mid \neg\varphi \mid \varphi \vee \varphi \mid EX\varphi \mid EG\varphi \mid E[\varphi U \psi] \mid \ K_i\varphi \mid E_\Gamma\varphi \mid C_\Gamma\varphi \mid D_\Gamma\varphi \\ & O_i\varphi \mid \ \hat{K}_i^\Gamma\, \varphi \mid \langle\langle\Gamma\rangle\rangle X\varphi \mid \langle\langle\Gamma\rangle\rangle G\varphi \mid \langle\langle\Gamma\rangle\rangle[\varphi U \psi].\end{aligned}$$

In this language, it is assumed that Γ is a set of agents, and the derived temporal operators F and A are obtained in the standard way[15].

$\mathbf{CTLKD} - \mathbf{A}^{D,C}$ formulae are evaluated in a deontic interpreted system

$$DIS = \Big\langle (G_i, R_i, Act_i, P_i, t_i)_{i \in \{1,\ldots,n\}}, (G_E, R_E, Act_E, P_E, t_E), I, V \Big\rangle$$

by associating an "enriched" model Me_{DIS} to DIS. In particular,

$$Me_{DIS} = (W, R_t, \sim_1, \ldots, \sim_n, R_1^O, \ldots, R_n^O, t, V).$$

The "enriched" model Me_{DIS} differs from the model M_{DIS} presented in Section 2.1.7.2 in that the evolution function $t : G \times Act \rightarrow G$ is carried over from DIS to Me_{DIS}. Indeed, the evolution function t is employed in the evaluation of **ATL** operators, as follows. Let $\Sigma = \{1, \ldots, n\}$ denote the set of agents and let $\mathrm{pre}_\Gamma(\varphi)$ be the set of states defined by $\mathrm{pre}_\Gamma(\varphi) = \{w \in W | \exists a \in Act_\Gamma$ s.t. $\forall a' \in Act_{\Sigma\setminus\Gamma}$ all temporal transitions labelled with the $< a, a' >$ lead to a state w' s.t. $Me_{DIS}, w' \models \varphi$.

$Me_{DIS}, w \models \langle\langle\Gamma\rangle\rangle X\varphi$	iff	$w \in \mathrm{pre}_\Gamma(\varphi)$
$Me_{DIS}, w \models \langle\langle\Gamma\rangle\rangle G\varphi$	iff	$Me_{DIS}, w \models \varphi$ and for all paths π from w and, for all states w_i, w_{i+1} of π, $Me_{DIS}, w_{i+1} \models phi$ and $w_i \in \mathrm{pre}_\Gamma(\varphi)$
$Me_{DIS}, w \models \langle\langle\Gamma\rangle\rangle[\varphi U \psi]$	iff	for all temporal paths π starting from w, the agents in Γ may perform joint actions along the paths s.t. eventually ψ will hold and φ holds along the paths until then.

Notice that the semantics presented above corresponds to the memoryless, imperfect information semantics of **ATL**, because actions for agent i depend on the current local state only via the protocol P_i. A "partial" memory semantics could be defined in this formalism by adding a vector to the local states of an agent, containing the list of previously visited

knowledge operator in various classes on interpreted systems (synchronous, asynchronous, perfect recall, etc.).

[15]Notice that **CTL** operators may, in fact, be derived from **ATL** operators. Here the two classes of operators are kept separated, to underline the difference between *temporal* reasoning with **CTL**, and *strategic* reasoning with **ATL**.

[15]A joint action a for a group of agents Γ is a tuple belonging to the set Act_Γ, where Act_Γ is the Cartesian product $Act_\Gamma = \prod_{i \in \Gamma} a_i$. Given two joint actions $a \in Act_\Gamma$ and $a' \in Act_{\Sigma\setminus\Gamma}$, $< a, a' > \in Act$ is the joint action obtained by the concatenation of a and a' (with the appropriate reordering of terms, if needed).

[15]A joint action a for a group of agents Γ is a tuple belonging to the set Act_Γ, where Act_Γ is the Cartesian product $Act_\Gamma = \prod_{i \in \Gamma} a_i$. Given two joint actions $a \in Act_\Gamma$ and $a' \in Act_{\Sigma\setminus\Gamma}$, $< a, a' > \in Act$ is the joint action obtained by the concatenation of a and a' (with the appropriate reordering of terms, if needed).

local states. This option is denoted by the term *bounded recall*.

This thesis distinguishes three classes of deontic interpreted systems to interpret $\mathbf{CTLKD} - \mathbf{A}^{D,C}$ formulae:

1. **Non-deterministic deontic interpreted systems**: this is the most general class of interpreted systems, defined in Section 2.1.7.2. In this class, **ATL** operators express what agents *may bring about*, maybe by guessing moves, and not what agents may *enforce*. Nevertheless, such interpretation may be useful in certain circumstances.

2. **Deterministic deontic interpreted systems**: the general definition of deontic interpreted systems enables agents to run non-deterministic protocols, i.e., the same agent may non-deterministically perform different actions in the same local state. To avoid random actions, it is possible to focus on the subclass of deontic interpreted systems, whose protocols are deterministic, i.e., protocols in which only one action is associated to a given local state: $P_i : L_i \rightarrow Act_i$. A deontic interpreted system is said to be *deterministic* iff the protocol of each agent is *deterministic* (a deterministic protocol associates a unique action to each local state). Notice that, in deterministic interpreted systems, agents perform the same action in *epistemically equivalent states*.

3. **Γ-uniform deontic interpreted systems**: in many circumstances the class of deterministic interpreted systems is too restrictive to be used in the specification of MAS scenarios. In these circumstances it is useful to reason about non-deterministic interpreted systems that are at least consistent in their selection of actions in a given local state. For instance, in the example of Figure 2.3, it is reasonable to assume that, if the player decides to keep its card in state `<A,K>`, then it should do so in state `<A,Q>` as well. By extending the concepts of [Jonker, 2003, Jamroga and van der Hoek, 2004], an agent is defined to be *uniform* if the agent performs the same action in epistemically equivalent global states. A group of agents $\Gamma \subseteq \Sigma$ is *uniform* if every agent in the group is uniform. A deontic interpreted system is Γ-*uniform* if all agents in Γ are uniform, whereas agents in $\Sigma \backslash \Gamma$ and the environment may choose their actions freely, according to their protocol. A Γ-uniform deontic interpreted system DIS_Γ is said to be *compatible* with a given non-deterministic deontic interpreted system DIS if:

   - The group of agents Γ is uniform in DIS_Γ;
   - The protocols for agents in Γ of DIS_Γ are a *restriction* of the protocols of DIS (in the sense that only one action is enabled in DIS_Γ, and the action is one of the enabled actions in DIS);
   - All the remaining parameters remain the same.

Notice that, for a given non-deterministic deontic interpreted system, there may be several Γ-uniform deontic interpreted systems *compatible* with it, but at most $\prod_{i\in\Gamma} |Act_i|^{|L_i|}$. The set of Γ-uniform deontic interpreted systems compatible with a given deontic interpreted system *DIS* is denoted by $\{DIS\}_\Gamma$. A formula φ is *true in the class of* Γ-uniform deontic interpreted systems compatible with a given *DIS*, and it is written $DIS \models_\Gamma \varphi$, if φ is true in *at least one* of the models associated with the deontic interpreted systems in $\{DIS\}_\Gamma$. Application examples of Γ-uniform deontic interpreted systems are presented in Section 6.3.

2.1.7.4 Why interpreted systems?

Interpreted systems and their extensions to reason about time, knowledge, correct behaviour, and actions *à la* **ATL**) offer a suitable formalism for modelling multi-agent systems. This thesis builds on this formalism for various reasons:

- Interpreted systems are *computationally grounded* [Wooldridge, 2000a]: the semantics of interpreted systems maps directly to *runs* of a system, and *vice-versa.* Indeed, the description of a scenario in terms of runs of interpreted systems (using local states, protocols, etc.) immediately provides a logic model to evaluate formulae (i.e., specifications).
- Differently from other formalisms, epistemic properties are not *ascribed* to agents by means of sets of propositions; instead, epistemic properties are based on the *equivalence* of local states.
- The concept of "local states" offers a flexible abstraction for the agents. Local states can be "singletons", corresponding to a very high level description of the agents. But local states are allowed to have a more complex structure: for instance, local states could be arrays of variables, or a combination of singletons and arrays, thereby allowing for a "fine grained" description of agents.
- Interpreted systems are easily extensible: the original work of [Fagin et al., 1995] includes temporal and epistemic operators only, but in the previous sections it has been shown that various extensions are possible to include other modalities.

Some issues remain open when formalising MAS using interpreted systems, e.g., the lack of modalities to reason about typical agents' stances such as desires and intentions. Nevertheless, as shown in the examples of Chapter 6, interpreted systems are a useful abstraction, as "frictionless" is in Physics.

2.2 Model checking

2.2.1 Problem definition

Model checking is the problem of establishing whether or not a given formula φ is true in a given model M. Notice that, unlike the problem of satisfiability of a formula, model checking has two input parameters: the formula φ and the model M.

Historically, techniques to perform model checking have received little attention in modal logic. Apart from the definition of satisfiability in a model (denoted in the previous Section by $M \models \varphi$), no references to model checking techniques appear in [Chellas, 1980, Goldblatt, 1992, Hughes and Cresswell, 1996, Blackburn et al., 2001, Gabbay et al., 2003]. Nevertheless, model checking techniques are prominent in the area of *formal* verification. Using model checking, the problem of verifying that a generic system S complies with a given specification P is reduced to the problem of verifying that a logical formula φ_P (representing the specification P) is satisfied in a model M_S (representing the generic system S). No particular requirements are usually imposed on S, even though, in most cases, S is required to be finite. Traditionally, systems have been represented by means of *temporal* models, i.e., **LTL**, **CTL**, or **CTL*** models, and specifications have been encoded using the language of one of these logics. Indeed, the expressivity of temporal logics is suitable for the description of many requirements. A *pattern system* for temporal logics has been investigated in [Dwyer et al., 1998], where a *pattern* is defined as the "description of a commonly occurring requirement"; for instance, safety and liveness are two commonly recurring specification patterns, but many others can be defined (see [Dwyer et al., 2006] for an ongoing project collecting temporal specification patterns).

The expressivity of temporal logics, combined with the possibility of abstracting generic systems by means of logical models of temporal logics, has lead to the development of tools and techniques for the verification of many scenarios, from hardware circuits, to communication protocols, and software (see [Clarke et al., 1999] and references therein). The process of representing a generic system S with a (temporal) model M_S, however, suffers from the so called *state explosion problem*: the number of states in the model M_S grows exponentially with the number of variables, or parallel components, constituting the system S. For instance, modelling explicitly a simple piece of software containing 20 variables of type `byte`, would require a model with $256^{20} \approx 1.5 \cdot 10^{48}$ states. Thus, one of the main challenges of model checking is the development of efficient techniques to tackle the state explosion problem. The next sections present various solutions from the literature.

2.2.2 Model checking techniques

2.2.2.1 Fix-point characterisation of CTL and the labelling algorithm

This section introduces the *labelling algorithm* for model checking **CTL**. This algorithm is based on the fix-point characterisation of some **CTL** operators.

Let Q be a set; an operator $\tau : 2^Q \rightarrow 2^Q$ is said to be *monotonic* if, given two sets $X, Y \subseteq Q$, $X \subseteq Y$ implies $\tau(X) \subseteq \tau(Y)$. It is possible to prove [Tarski, 1955] that a monotonic operator τ has a greatest and a least fix-point; these are denoted by $\nu Z.\tau(Z)$ and $\mu Z.\tau(Z)$, respectively. Let $\tau^i(X)$ be defined by $\tau^0(X) = X$, and $\tau^{i+1}(X) = \tau(\tau^i(X))$. If Q is finite and τ is monotonic, then there exist integer numbers n, m such that $\nu Z.\tau(Z) = \cap_i \tau^n(Q)$ and $\mu Z.\tau(Z) = \cup_i \tau^n(\emptyset)$.

The monotonicity properties above are used in conjunction with the following equivalences [Huth and Ryan, 2004] for the purpose of **CTL** model checking:

$$EG\varphi \equiv \varphi \wedge EXEG\varphi; \tag{2.1}$$

$$E[\varphi U \psi] \equiv \psi \vee (\varphi \wedge EXE[\varphi U \psi]). \tag{2.2}$$

Let φ be a **CTL** formula, let $M = (S, R, V)$ be a **CTL** model, and let $[\![\varphi]\!]_M \subseteq S$ denote the set of states of M in which φ holds[16]. The equivalences above imply the following:

$$[\![EG\varphi]\!] \equiv [\![\varphi]\!] \cap [\![EXEG\varphi]\!]; \tag{2.3}$$

$$[\![E[\varphi U \psi]]\!] \equiv [\![\psi]\!] \cup ([\![\varphi]\!] \cap [\![EXE[\varphi U \psi]]\!]). \tag{2.4}$$

Following [Huth and Ryan, 2004], let $\text{pre}_\exists(X)$ denote a procedure that, given a set $X \subseteq S$, computes the set of states $Y \subseteq S$ from which a transition is enabled to a state in X, i.e.:

$$Y = \text{pre}_\exists(X) = \{s \in S | \exists s'.(s' \in X \text{ and } sRs')\}.$$

Using this procedure, it is possible to rewrite equations 2.3 and 2.4 as follows:

$$[\![EG\varphi]\!] \equiv [\![\varphi]\!] \cap \text{pre}_\exists([\![EG\varphi]\!]); \tag{2.5}$$

$$[\![E[\varphi U \psi]]\!] \equiv [\![\psi]\!] \cup ([\![\varphi]\!] \cap \text{pre}_\exists([\![E[\varphi U \psi]]\!])). \tag{2.6}$$

Let $\tau_{EG,\varphi} : 2^S \rightarrow 2^S$ be the operator defined by $\tau_{EG}(X) = [\![\varphi]\!] \cap \text{pre}_\exists(X)$, and let $\tau_{EU,\varphi,\psi} : 2^S \rightarrow 2^S$ be defined by $\tau_{EU,\varphi,\psi}(X) = [\![\psi]\!] \cup ([\![\varphi]\!] \cap \text{pre}_\exists(X))$. Equations 2.5 and 2.6 imply that $[\![EG\varphi]\!]$ is the fix-point of the operator $\tau_{EG,\varphi}$, while $[\![E[\varphi U \psi]]\!]$ is the fix-point of $\tau_{EU,\varphi,\psi}$. It is possible to prove that the operators $\tau_{EG,\varphi}$ and $\tau_{EU,\varphi,\psi}$ are monotonic,

[16]The subscript M will be omitted when it is clear from the context.

```
$\textsc{mc}(\varphi, M)$ {
  $\varphi$ is an atomic formula: return $V(\varphi)$;
  $\varphi$ is $\neg\varphi_1$: return $S \setminus \textsc{mc}(\varphi_1, M)$;
  $\varphi$ is $\varphi_1 \vee \varphi_2$: return $\textsc{mc}(\varphi_1, M) \cup \textsc{mc}(\varphi_2, M)$;
  $\varphi$ is $EX\varphi_1$: return $\textsc{mc}_{EX}(\varphi_1, M)$;
  $\varphi$ is $EG\varphi_1$: return $\textsc{mc}_{EG}(\varphi_1, M)$;
  $\varphi$ is $E[\varphi_1 U \varphi_2]$: return $\textsc{mc}_{EU}(\varphi_1, \varphi_2, M)$;
}
```

Figure 2.4: The labelling algorithm, from [Huth and Ryan, 2004].

```
$\textsc{mc}_{EX}(\varphi, M)$ {
  $X = \textsc{mc}(\varphi, M)$;
  $Y = \text{pre}_\exists(X)$;
  return $Y$;
}
```

Figure 2.5: The support procedure $\textsc{mc}_{EX}(\varphi, M)$, from [Huth and Ryan, 2004].

and that $[\![EG\varphi]\!]$ is the greatest fix-point of $\tau_{EG,\varphi}$, while $[\![E[\varphi U \psi]]\!]$ is the least fix-point of $\tau_{EU,\varphi,\psi}$ [Clarke et al., 1999, Huth and Ryan, 2004]. Thus, there exist finite natural numbers n and m such that $[\![EG\varphi]\!] = \tau^n_{EG,\varphi}(S)$ and $[\![E[\varphi U \psi]]\!] = \tau^m_{EU,\varphi,\psi}(\emptyset)$.

The characterisation of the operators EG and EU using fix-points permits the definition of algorithm for model checking **CTL** formulae, denoted with MC. The algorithm MC takes a formula φ and a **CTL** model M as input, and operates by labelling with the string φ all the states of M in which φ holds; equivalently, it can be said that the algorithm $\textsc{mc}(\varphi, M)$ computes the set $[\![\varphi]\!]$.

```
$\textsc{mc}_{EG}(\varphi, M)$ {
  $X = \textsc{mc}(\varphi, M)$;
  $Y = S$;
  $Z = \emptyset$;
  while ( $Z != Y$ ) {
    $Z = Y$;
    $Y = X \cap \text{pre}_\exists(Y)$;
  }
  return $Y$;
}
```

Figure 2.6: The support procedure $\textsc{mc}_{EG}(\varphi, M)$, from [Huth and Ryan, 2004].

```
$\textsc{mc}_{EU}(\varphi_1, \varphi_2, M)$ {
  $X = \textsc{mc}(\varphi_1, M)$;
  $Y = \textsc{mc}(\varphi_2, M)$;
  $Z = \emptyset$;
  $W = S$;
  while ( $Z != W$ ) {
    $W = Z$;
    $Z = Y \cup (X \cap \mathrm{pre}_{\exists}(Z))$;
  }
  return $Z$;
}
```

Figure 2.7: The support procedure $\textsc{mc}_{EU}(\varphi_1, \varphi_2, M)$, from [Huth and Ryan, 2004].

Figure 2.4 presents the labelling algorithm to compute $[\![\varphi]\!]$, from [Huth and Ryan, 2004]. The additional procedures $\textsc{mc}_{EX}(\varphi_1, M)$, $\textsc{mc}_{EG}(\varphi_1, M)$, and $\textsc{mc}_{EU}(\varphi_1, \varphi_2, M)$ implementing the fix-point characterisation presented above are presented in Figures 2.5–2.7.

2.2.2.2 Ordered Binary Decision Diagrams and symbolic model checking

This section introduces Ordered Binary Decision Diagrams (OBDDs), and presents how the problem of model checking a **CTL** formula in a model M can be reduced to the problem of comparing two OBDDs.

A *Boolean variable* x is a variable whose value is either 0 or 1. A *Boolean function* of n Boolean variables is a function $f : \{0,1\}^n \rightarrow \{0,1\}$. *Boolean formulae* can be seen as Boolean functions. For instance, the Boolean formula $x_1 \wedge (x_2 \vee x_3)$ can be seen as the Boolean function $f(x_1, x_2, x_3) = x_1 \wedge (x_2 \vee x_3)$.

A rooted, directed graph G can be associated to every Boolean function $f(x_1, \ldots, x_n)$ by imposing an ordering on the variables $x_1, \ldots, x_n$, and by reducing the graph (in the sense explained below) [Bryant, 1986]. The graph G is called the *Ordered Binary Decision Diagrams* of f. For instance, the reduced graph associated with the Boolean function $f(x_1, x_2, x_3) = x_1 \wedge (x_2 \vee x_3)$ is depicted in Figure 2.8 (b), by "simplifying" the graph depicted in Figure 2.8 (a). Formally, a graph is reduced by iteratively eliminating the vertices which are the root of two isomorphic subgraphs, and by merging isomorphic subgraphs. A graph is said to be *reduced* if it contains no isomorphic subgraphs and no vertices v and v' such that the sub-graphs rooted at v and v' are isomorphic. Notice that every vertex, except the final leaves, has two children. In the remainder, it is assumed that the left child of a vertex corresponds to the choice of the value 0 (i.e., *false*) for the variable preceding it, while the right child correspond to the choice of the value 1 (i.e.,

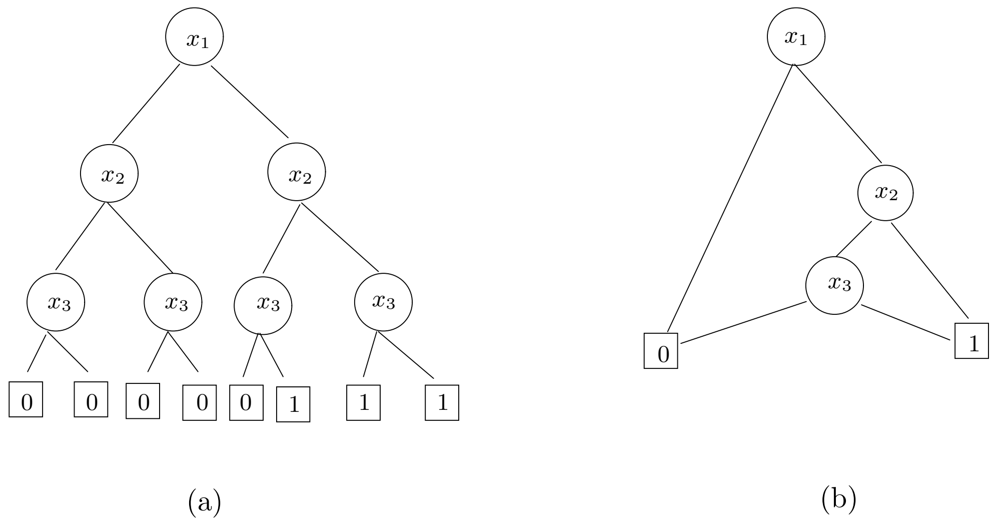

Figure 2.8: OBDD example for $f = x_1 \wedge (x_2 \vee x_3)$.

true). Thus, the leftmost path of Figure 2.8 (a) corresponds to an assignment of 0 to all variables and, consequently, to the value 0 to the expression $f(x_1, x_2, x_3) = x_1 \wedge (x_2 \vee x_3)$.

It is shown in [Bryant, 1986] that, given a fixed ordering of the Boolean variables $x_1, \ldots, x_n$, the reduced graph of any Boolean function $f : \{0,1\}^n \rightarrow \{0,1\}$ is unique (i.e., OBDDs are a *canonical* representation for Boolean functions).

Boolean operators can be applied to Boolean functions; for instance the disjunction operator $\vee$ can be applied to two Boolean functions f_1 and f_2 to obtain a third Boolean function $f_3 = f_1 \vee f_2$. Boolean functions can be composed, too: given two Boolean functions f and g, the composition of f and g is defined by $f_{x_i=g} = f(x_1, \ldots, x_{i-1}, g(x_1, \ldots, x_n), x_{i+1}, \ldots, x_n)$. These operators are denoted with `apply(`f, g`,<operator>)`, and with `compose((`f, g, x_i`)`.

The operation of *Boolean quantification* is particularly important for the purposes of model checking. Formally, given a Boolean function $f(x_1, \ldots, x_n)$, the operation $\exists x_i . f(x_1, \ldots, x_n)$ is defined as the application of the disjunction operator to the composition of f with a constant function, i.e., $\exists x_i . f(x_1, \ldots, x_n) = f_{x_i=0}(x_1, \ldots, x_n) \vee f_{x_i=1}(x_1, \ldots, x_n)$. The definition of Boolean quantification can be extended to the quantification over a set of variables $\bar{x} = (x_1, \ldots, x_n)$ (see [Clarke et al., 1999] for more details). Boolean quantification of a Boolean function f can be implemented for the OBDD representing f; the complexity of this operation, together with the complexity of other operations on OBDDs, is presented in Section 2.2.5.

Ordered binary decision diagrams have been particularly successful in Computer Science because they offer, on average, a much more compact representation of Boolean functions

State	Boolean vector	Boolean formula
s_1	$(1,1)$	$x_1 \wedge x_2$
s_2	$(1,0)$	$x_1 \wedge \neg x_2$
s_3	$(0,1)$	$\neg x_1 \wedge x_2$

Table 2.6: Boolean encoding for the states of $S = \{s_1, s_2, s_3\}$ ($N = \lceil log_2(3) \rceil = 2$).

with respect to other canonical forms, e.g. conjunctive/disjunctive normal forms. The application of OBDDs techniques to model checking for **CTL** has been investigated from the beginning of the 1990s by various authors, [Burch et al., 1992, McMillan, 1993]. Intuitively, given a **CTL** formula φ and a **CTL** model $M = (S, R, V)$, the idea of model checking using OBDDs is to associate an OBDD to the formula φ, and an OBDD to the set of states S. By comparing the two OBDDs it is possible to establish whether or not $M \models \varphi$[17]. The details of this technique are presented below.

Encoding sets of states. The key idea of model checking using OBDDs is to represent states (and set of states) as Boolean formulae which, in turn, can be encoded as OBDDs. Let S be the set of states of a **CTL** model $M = (S, R, V)$ (notice: it is assumed that the set of states of M is finite), and let $N = \lceil log_2|S| \rceil$. Each element $s \in S$ is associated with a vector of Boolean variables $\bar{x} = (x_1, \dots, x_N)$, i.e., each element of s is associated with a tuple of $\{0,1\}^N$. Each tuple $\bar{x} = (x_1, \dots, x_N)$ is then identified with a Boolean formula, represented by a conjunction of literals, i.e., a conjunction of variables or their negation[18]. It is assumed that the value 0 in a tuple corresponds to a negation. An example of Boolean encoding for the set $S = \{s_1, s_2, s_3\}$ is given in Table 2.6.

Sets of states are encoded by taking the disjunction of the Boolean formulae encoding the single states. For instance, the set of states $\{s_1, s_3\}$ from the example in Figure 2.6 is encoded by the Boolean formula $f = (x_1 \wedge x_2) \vee (\neg x_1 \wedge x_2)$.

Encoding the transition relation. Given a model $M = (S, R, V)$, and given an encoding of the set of states S using N Boolean variables $(x_1, \dots, x_N)$, the transition relation $R \subseteq S \times S$ may be encoded as a Boolean function. To this end, a new set of "primed" variables $(x'_1, \dots, x'_N)$ is introduced to encode the relation between two states $s, s' \in S$. In particular, if sRs' holds, then s is encoded using the non-primed variables, s' is encoded using the primed variables, and the transition step sRs' is expressed as a Boolean formula by taking the conjunction of the encoding for s and s'. The whole relation $R \subseteq S \times S$ is

[17]This technique is traditionally identified with the term *symbolic model checking*. More precisely, [McMillan, 1993] defines symbolic model checking as a technique that "avoids building a state graph by using Boolean formulas to represent sets and relations". Some authors [Schnoebelen, 2003] use the term "symbolic model checking" in a more general sense to denote any technique in which the model is not given "explicitly", but by means of some "compact" representation (Boolean functions being one possible choice). To avoid confusion, this thesis employs the term "symbolic model checking" in the stricter sense, to denote model checking techniques based on Boolean functions.

[18]By slight abuse of notation, the same symbols $x_i (i \in \{1, \dots, N\})$ are used to denote Boolean variables in a vector, and atomic propositions in logical formulae.

encoded as a Boolean formula by taking the disjunction of all the transition steps.

As an example, let $R = \{(s_1, s_2), (s_2, s_3), (s_3, s_1)\}$ be a transition relation for the states of the example in Figure 2.6. This transition relation is encoded by the following Boolean formula f_R:

$$f_R(x_1, x_2, x'_1, x'_2) = [(x_1 \wedge x_2) \wedge (x'_1 \wedge \neg x'_2)] \vee [(x_1 \wedge \neg x_2) \wedge (\neg x'_1 \wedge x'_2)] \vee [(\neg x_1 \wedge x_2) \wedge (x'_1 \wedge x'_2)].$$

The labelling algorithm and Boolean formulae. The algorithm presented in Figure 2.4 returns the set of states satisfying a formula φ in a given model $M = (S, R, V)$. The algorithm operates recursively on the structure of φ and builds the set of states $[\![\varphi]\!]$ using the following operations on sets: union, intersection, complementation, existential quantification. When sets of states are encoded using Boolean formulae, all these operations on sets may be translated into operations on Boolean formulae:

- the union of two sets corresponds to the disjunction of the Boolean formulae encoding the two sets;
- the intersection of two sets corresponds to the conjunction of the Boolean formulae encoding the two sets;
- the complementation of a set P with respect to a given set Q (i.e., $P \backslash Q$) is the conjunction of the Boolean formula encoding Q with the negation of the Boolean formula encoding P;
- the existential quantification of an element x in a set P is the (quantified) Boolean formula $\exists \bar{v_x}.f_P$, where $\bar{v_x}$ are the Boolean variables required to encode x, and f_P is the Boolean formula encoding P.

In the basic case (i.e., when φ is an atomic proposition) the algorithm returns a set of states: by encoding this set of states as a Boolean formula, the algorithm of Figure 2.4 can operate entirely on the Boolean representation of a model $M = (S, R, V)$ to return a Boolean formula encoding the set of states $[\![\varphi]\!]$.

The labelling algorithm and model checking using OBDDs. All the Boolean formulae mentioned in the previous step can be represented using OBDDs. Thus, the algorithm of Figure 2.4 provides a methodology to build the OBDD corresponding to the set of states $[\![\varphi]\!]$ in which a formula φ holds for a given model M. The problem of model checking is reduced in this way to the problem of comparing the OBDDs for $[\![\varphi]\!]$ and for M. As OBDDs offer a canonical representation for Boolean formulae, this last step is limited to the verification that the two OBDDs are equal. The proof of the correctness of this approach can be found in [Clarke et al., 1999, Huth and Ryan, 2004].

Notes

- The process of translating the problem of model checking into the comparison of two OBDDs may seem to increase the complexity of model checking. However, as it will become clear in Section 2.2.3, the models are not built explicitly in model checking tools; instead, the OBDDs representing the various parameters in the models are obtained incrementally from a dedicated programming language, thereby permitting the verification of models whose size would be intractable.
- The problem of verifying that a formula φ holds in a given model M is defined by some authors with the term *global* model checking, as opposed to the problem of *local* model checking, which is the problem of establishing whether or not a formula is true at a given state in a given model. The algorithm presented in Figure 2.4 can be employed for local model checking as well: indeed, it is sufficient to check that the state in which a formula φ has to be verified is included in the set $[\![\varphi]\!]$.
- As mentioned in Section 2.1.4, in certain cases a **CTL** model includes a set of *initial states*: $M = (S, R, V, I)$. The evolution of the system is described by the transition function R, and it may happen that not all states of S are reachable. In this case, formulae need to be evaluated in the set of *reachable* states only, and complementation must be limited to the set of reachable states. Reachable states can be encoded as an OBDD (Section 3.3 explores this issue in more detail).

2.2.2.3 SAT-based translations

Other techniques exist to perform model checking for temporal logics. This section introduces techniques that reduce the problem of model checking to a problem of satisfiability for a Boolean formula (SAT). Efficient procedures for Boolean satisfiability have been investigated and implemented since the 1960s; thus the reduction of the problem of model checking to a Boolean satisfiability problem may benefit from the advances in this area.

Bounded model checking for LTL. The idea of reducing the problem of model checking for **LTL** to a satisfiability problem was introduced in [Biere et al., 1999a], and it is based on the concept of *bounded semantics* for **LTL** models. Intuitively, given an **LTL** model $M = (S, R, V, I)$, an **LTL** formula φ and a (finite) integer k, the expression $M \models_k \varphi$ is read as "formula φ holds in M along a path of length k (starting from the set of initial states)". It can be proven that $M \models \varphi$ iff there exists a finite integer k such that $M \models_k \varphi$. The problem $M \models_k \varphi$ is reduced to propositional satisfiability, as follows. A propositional formula $[M, \varphi]_k$ is built such that $[M, \varphi]_k$ is satisfiable iff φ holds along some path π of length k starting from the set of initial states. To construct $[M, \varphi]_k$, a propositional formula $[M]_k$ if defined first to enforce valid paths of length k. Then, the **LTL** formula φ is translated into a propositional formula $[\varphi]_k$, and $[M, \varphi]_k$ is obtained as the conjunction of $[M]_k$ and $[\varphi]_k$: $[M, \varphi]_k = [M]_k \wedge [\varphi]_k$. The Boolean formula $[M, \varphi]_k$ is satisfiable iff $M \models_k \varphi$. A detailed presentation of this approach can be found in [Biere et al., 1999a, Clarke et al., 1999].

Bounded model checking for CTL. A bounded semantics for the universal fragment of **CTL** was introduced in [Penczek et al., 2002]. The universal fragment of **CTL**, denoted with **ACTL**, restricts negation to atomic formulae only, and permits universally quantified temporal operators only. Given a **CTL** model $M = (S, R, V, I)$ and an **ACTL** formula φ, bounded model checking for **CTL** is similar, in spirit, to bounded model checking for **LTL**, in that a Boolean formula $[M, \varphi]_k$ is built as a conjunction of two Boolean formulae $[M]_k$ and $[\varphi]_k$. The technical machinery involved in **ACTL** model checking, however, is substantially different from **LTL** due to the branching structure of **CTL** models (notice that the bounded semantics for **ACTL** depends on a set of initial states).

Unbounded model checking. It has been shown [Biere et al., 1999a] that bounded model checking techniques can identify false formulae in a much quicker way than OBDD-based techniques, when counter-examples can be found for small values of the bound k. When such value is high, or when formulae are true in a model, however, there is a significant decrease in the performance of SAT-based techniques. A possible solution for this issue has been presented in [McMillan, 2002], introducing *unbounded model checking*. Intuitively, unbounded model checking techniques are based on algorithms similar to the algorithm presented in Figure 2.4: Boolean formulae representing sets of states are computed but, instead of representing Boolean formulae using OBDDs and comparing OBDDs, the problem of model checking is translated into a satisfiability problem for Boolean formulae (typically represented in conjunctive normal form). Model checkers implementing unbounded model checking are currently being developed, but no experimental results are available yet.

2.2.2.4 Automata-based techniques

A different approach for model checking the logic **LTL** was proposed in the 1980s by [Vardi and Wolper, 1986] using *automata*. An automaton is a tuple $A = (\Sigma, Q, Q^0, \delta, F)$ where:

- Σ is a finite alphabet;
- Q is a finite set of states, and $Q^0 \subseteq Q$ is a set of initial states;
- $\delta \subseteq Q \times \Sigma \times Q$ is a transition relation;
- $F \subseteq Q$ is a set of final (or accepting) states.

An automaton can be represented as a graph; for instance, Figure 2.9 depicts the automaton characterised by $\Sigma = \{a, b\}$, $Q = \{0, 1\}$, $Q^0 = \{0\}$ (this is indicated with an incoming

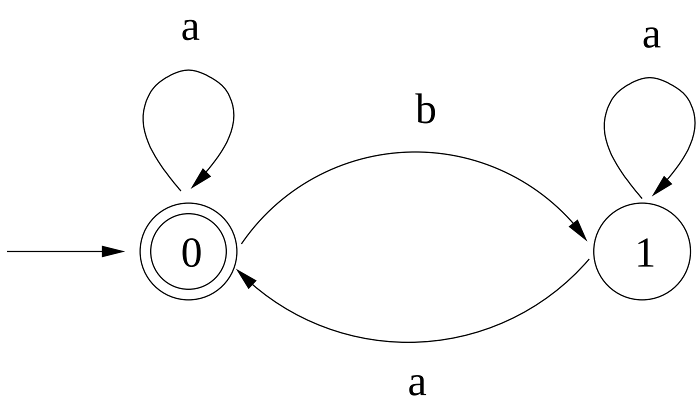

Figure 2.9: Automaton example.

arrow in the graph), $F = \{0\}$ (this is indicated with a double circle in the graph), and $\delta = \{(0, a, 0), (0, b, 1), (1, a, 1), (1, a, 0)\}$.

Let $\sigma \in \Sigma^*$ be a string; an automaton A *accepts* σ iff there exists a sequence of states of $(q_0, q_1, \ldots, q_{|\sigma|}) \in Q^*$ such that $q_0 \in Q^0$, $q_{|\sigma|} \in F$, and for all states in the sequence there exist transitions $(q_i, \alpha, q_{i+1}) \in \delta$ such that α is the i-th symbol in σ (such sequences are usually called *runs* of A). The set of strings accepted by A is called the *language accepted* by A and it is denoted with $\mathcal{L}(A)$. For instance, the language accepted by the automaton in Figure 2.9 includes the strings λ (the empty string), *aaaba*, *aabaaaba*, etc.

Automata over infinite words accept strings of infinite length from Σ^ω; their definition is similar to the definition of automata over finite words presented above, the only difference being the acceptance condition. Different kind of acceptance conditions can be defined, and these correspond to different kind of automata. Büchi automata are defined as follows. Let $\sigma \in \Sigma^\omega$ be an infinite string, and let $i(\sigma)$ be the set of states appearing infinitely often in a run of A accepting the string σ. Given a set F of accepting states, a Büchi automaton $A = (\Sigma, Q, Q^0, \delta, F)$ accepts σ iff $i(\sigma) \cap F \neq \emptyset$ (i.e., at least an accepting state from F appears infinitely often in the run). Notice that, in automata over infinite words, Q and Σ are *finite* sets.

Model checking and automata (From [Clarke et al., 1999]). A model $M = (S, R, V, I)$ for the logic **LTL** can be translated into an automaton $A_M = (\Sigma, Q, Q^0, \delta, F)$ as follows:

- $\Sigma = 2^{AP}$, where AP is a set of the atomic propositions appearing in V;
- $Q = S \cup \{\iota\}$, where ι is a special state of the automaton;
- $Q^0 = I \cup \{\iota\}$;
- $F = S \cup \{\iota\}$, i.e., all the states are accepting;

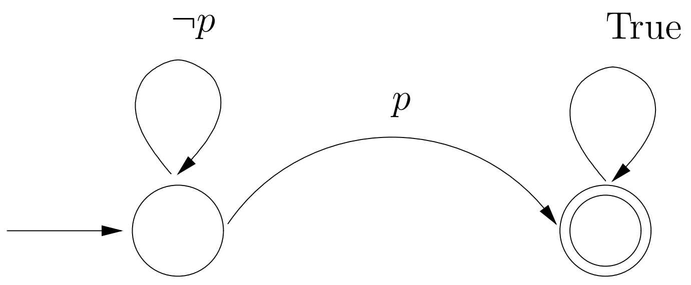

Figure 2.10: Automaton for the formula Fp.

```
MC($\varphi, M$) {
  Build the automaton $A_M$;
  Build the automaton accepting $\mathcal{L}(\bar{A}_\varphi)$;
  Build the automaton accepting $\mathcal{L}(A_M) \cap \mathcal{L}(\bar{A}_\varphi)$;
  Check emptiness of this latter automaton;
  Return YES if the automaton is empty;
}
```

Figure 2.11: The automata-based model checking algorithm for LTL.

- $(s, \alpha, s') \in \delta$ iff $(s, s') \in R$ and $\alpha = V(s')$. Moreover, $(\iota, \alpha, s) \in \delta$ iff $\alpha = V(s')$ and $s \in I$.

Notice that the permitted executions of the automaton A_M correspond to the possible runs in M.

Given a **LTL** formula φ, an automaton A_φ can be associated to φ such that $\mathcal{L}(A_\varphi)$ includes all the "allowed behaviours" of the system. The details of the construction of A_φ for a generic formula φ are beyond the scope of this summary and can be found, for instance, in [Clarke et al., 1999]. As an example, the automaton corresponding to the **LTL** formula Fp is represented in Figure 2.10.

Given an **LTL** model M and a formula φ of the same logic, $M \models \varphi$ iff $\mathcal{L}(A_M) \subseteq \mathcal{L}(A_\varphi)$, i.e., iff the language accepted by the automaton representing the model M is included in the language "allowed" by the automaton representing the formula φ. The inclusion condition can be rewritten in an equivalent way as $\mathcal{L}(A_M) \cap \mathcal{L}(\bar{A}_\varphi) = \emptyset$, where $\mathcal{L}(\bar{A}_\varphi)$ is the complement of the language $\mathcal{L}(A_\varphi)$ with respect to the set Σ^ω. There exist constructive procedures (see [Clarke et al., 1999] and references therein) to build the automata corresponding to the complement of a Büchi automaton and to the intersection of two Büchi automata. These procedures allow for the definition of a model checking algorithm based on automata; its high-level description is provided in Figure 2.11.

Various optimisation techniques can be employed to improve the efficiency of the algorithm of Figure 2.11. For instance, *on-the-fly model checking* is an optimisation technique which avoids building the automaton A_M in the first step of the algorithm. Instead, using this technique the automaton for $\mathcal{L}(\bar{A}_\varphi)$ is constructed first, and then the automaton accepting $\mathcal{L}(A_M) \cap \mathcal{L}(\bar{A}_\varphi)$ is constructed iteratively by adding states of M when needed. This technique sometimes permits to find counter-examples of false formulae in an efficient way.

2.2.3 Model checking tools

The techniques presented in Section 2.2.2 have been implemented in a number of software tools, from the early 1990s. This section briefly summarises three mature tools and their programming and specification languages; other tools exist, and they are briefly summarised in Section 2.3. The tools reviewed below have been chosen because of their robustness, of their wide circulation and availability, and because of their relevance with material presented in later chapters.

2.2.3.1 SPIN

The model checker SPIN (Simple Promela INterpreter) is one of the most mature model checkers available: it was introduced in the 1980s at Bell Labs, it has been available to the general public since 1991, and it has been continually developed since then. A general introduction to the tool can be found in [Holzmann, 1997], while the theoretical foundations and a detailed user manual are presented in the book [Holzmann, 2003]. The main characteristics of SPIN are:

- It is a model checker for the temporal logic **LTL**.
- It is mainly aimed at verification of protocols and software. SPIN's programming language PROMELA (PROcess MEta LAnguage) reflects this intended use (see below).
- It implements an automata-based algorithm for model checking and various optimisation strategies, including on-the-fly model checking and partial order reduction[19].
- It provides a graphical user interface (Xspin) to the model checker and to an interactive simulator.

[19]This technique is based on the observation that execution traces arising from different orderings of interleaved concurrent processes are sometimes equivalent for the evaluation of a formula φ.

```
mtype = { ... };
chan = { ... };
<type> = { ... };

proctype SampleProcess1(<args>) {
 ...
}

proctype SampleProcess1(<args>) {
 ...
}

init {
 main body
}
```

Figure 2.12: Structure of a PROMELA program.

The structure of a PROMELA program is represented in Figure 2.12. A program includes a declaration section for message types (**mtype**), for channel types (**chan**), and for global variables of various types (indicated collectively with the string <type> in the figure). Various processes may be defined in a PROMELA program using the keyword **proctype**; each process is defined by a name and by a list of accepted arguments. The behaviour of each process is defined in its *body* (not shown in the figure), and each process may include a list of local variables. Processes communicate using global variables and channels. Processes are initially created in the **init** section of the program, they execute concurrently, and they can be created by other processes.

The Xspin graphical interface acts as a “control centre” for the components of SPIN architecture (depicted in Figure 2.13). The user needs to provide a PROMELA program and an **LTL** formula φ; the **LTL** formula is then translated by the tool into an appropriate automaton. SPIN can be used as a simulator (to perform either random or interactive simulations), or as a model checker. In the latter case, a C program implementing the automata-based algorithm presented above is constructed and compiled, producing a binary executable that provides an answer to the model checking question.

2.2.3.2 MOCHA

MOCHA [Alur et al., 1998] is a model checker for the logic **ATL** presented in Section 2.1.4.3. Two versions of MOCHA are available: cMocha and jMocha, but only the former allows for model checking **ATL** formulae (the latter being more simulation-oriented).

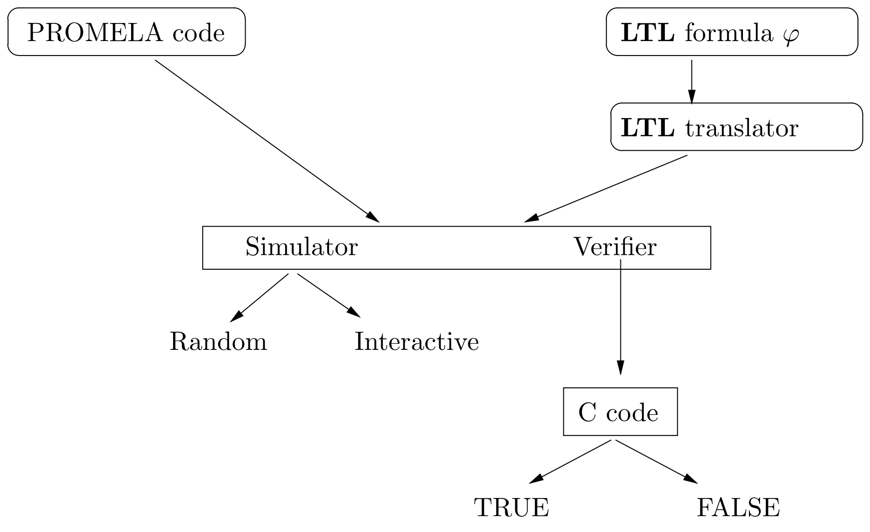

Figure 2.13: Structure of SPIN.

Model checking is performed in MOCHA by extending the labelling algorithm presented in Figure 2.4 for **CTL** to **ATL** formulae: indeed, it has been shown in [Alur et al., 1998, Alur et al., 2002] that **ATL** operators can be characterised using fix-points. Therefore, the techniques presented in Section 2.2.2.2 for the verification of **CTL** using OBDDs can be extended to the verification of **ATL** formulae. The details of this approach can be found in [Alur et al., 1998].

Systems are specified in MOCHA by using the dedicated language ReactiveModules. Each specification consists of one or more *modules*; each module is characterised by a set of input variables (called external variables), by a set of output variables (called interface variables), and by a set of local variables. In each module, the initial value and the evolution of variables is controlled by a set of constructs called *atoms*. Instances of modules are created at end of the file, and they are composed in parallel. Excerpts from a ReactiveModules program are reported in Figure 2.14; the full syntax of the language is available from [Alur et al., 2006].

MOCHA provides a graphical user interface to input programs written in ReactiveModule, and **ATL** formulae. Interactive simulations and model checking may be performed either using the interface, or using command line instructions.

2.2.3.3 SMV and NuSMV

SMV (Symbolic Model Verifier, [McMillan, 1992]) and NuSMV [Cimatti et al., 2002] are two of the most widely cited model checkers. The SMV system was developed at the beginning of the 1990s to implement the OBDD-based symbolic model checking techniques

```
module SampleModule1
  private  v1: bool
  external  v2: bool
  interface  v3: bool
  atom controls v1 reads v1 awaits v3
    init [. . . ]
    update [. . . ]
  endatom
  [. . . ]
endmodule

module SampleModule2
  [. . . ]
endmodule

Main := [. . . ] ( SampleModule1 || SampleModule2 ) [. . . ]
```

Figure 2.14: Excerpts from a REACTIVEMODULE program.

for **CTL** presented in Section 2.2.2.2 using OBDDs.

NuSMV is a re-implementation of the SMV system; the tool is implemented in C language and it is available freely under an "open" license. NuSMV implements symbolic model checking techniques for **CTL** and bounded model checking techniques for **LTL**. NuSMV can operate either in "batch" mode or interactively using a text shell; in this case, simulations can be performed. NuSMV accepts parameters to optimise the size of OBDDs by means of various heuristic functions. OBDDs are manipulated using the CUDD library [Somenzi, 2005].

The input languages of SMV and NuSMV present minor differences. They both allow for a compact description of systems using modules, which may be composed to describe the evolution of states. The NuSMV program for a 3 bit counter is presented in Figure 2.15, as an example to introduce the syntax of the language. A NuSMV module is identified by a string (`counter_cell` in the figure), it may accept input parameters (`carry_in`), and it may include "local" variables (`value`). The initial value of the module and the evolution of the variables are defined in the section appearing under the **ASSIGN** keyword, using the constructs **init** and **next**. In particular, **next**`(value)` is read as "the next value of the variable `value` is obtained by taking the disjunction of the current value of it with the value of the variable `carry_in`, modulo 2". The keyword **DEFINE** is used to introduce a "derived" variable, i.e., a variable which is not part of the state space, but whose value may be derived from other variables. The behaviour of the system is described in the (mandatory) module `main`. In the example, three instances of the module `counter_cell` are created, imposing the constraints that `carry_out` of counter i is equal to `carry_in` of

```
MODULE counter_cell(carry_in)
 VAR value: boolean;
 ASSIGN
  init(value) := 0 ;
  next(value) := (value + carry_in) mod 2;
 DEFINE
  carry_out := value & carry_in;

MODULE main
 VAR
  bit0 := counter_cell(1);
  bit1 := counter_cell(bit0.carry_out);
  bit2 := counter_cell(bit1.carry_out);
```

Figure 2.15: An SMV program for a 3 bit counter (from [Cimatti et al., 2002]).

counter $i + 1$.

A possible execution of NuSMV in interactive mode to verify the 3 bit counter presented above is shown in Figure 2.16. In this example, NuSMV is run interactively with the option `-int`, and the file `counter.smv` is processed. At the command line of NuSMV (identified by the prompt `NuSMV` >), the command `go` launches a number of preliminary operations on the input file, including parsing the input text file and generating the OBDDs for the temporal relation. The command `check_spec -p "AX(bit0.value=0)"` launches the OBDD-based verification of the **CTL** formula `AX(bit0.value=0)`. As the formula is false, NuSMV outputs a counter-example.

2.2.4 Review of other temporal model checkers

Other tools are available to perform model checking and simulations. These tools include:

- **Model checkers for the verification of software**.
  - Blast (Berkley Lazy Abstraction Software verification Tool, [Henzinger et al., 2003]) is a model checker for the verification of C programs. It implements automata-based verification techniques with various optimisations for the verification of reachability properties.
  - CBMC is a model checker for ANSI-C programs implementing bounded model checking techniques [Clarke et al., 2004a]. Binary versions of the tool are available from [Clarke et al., 2006].

```
$ ./NuSMV -int counter.smv
NuSMV > go
NuSMV > check_spec -p "AX(bit0.value=0)"
-- specification AX bit0.value = 0 is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  bit0.value = 0
  bit1.value = 0
  bit2.value = 0
  bit0.carry_out = 0
  bit1.carry_out = 0
  bit2.carry_out = 0
-> State: 1.2 <-
  bit0.value = 1
  bit0.carry_out = 1
NuSMV >
```

Figure 2.16: A NuSMV session.

  - `Java PathFinder` (JPF) [Brat et al., 2000] is a model checker for Java bytecode programs, developed since 1999 at NASA Ames Research Center. JPF operates on the explicit representation of the state space using a number of optimisation techniques to detect deadlocks and unhandled exceptions.

- **Model checkers for real-time systems**. *Real-time* systems are systems extended with clocks to reason about the flow of time between events [Alur and Dill, 1994]. Typically, properties of real-time systems are given in **TCTL**, an extension of the logic **CTL** including time constraints in logical formulae. For instance, the **TCTL** formula $AF_{\leq 5}(\text{ready})$ is read as "in all future paths, the proposition ready will become true in less than 5 time units". More details about **TCTL** and model checking techniques for real-time systems can be found in [Alur and Dill, 1994, Penczek et al., 2004, Woźna and Zbrzezny, 2005] and in the presentations of the numerous tools available for real-time verification. These tools include UPPAAL [Bengtsson et al., 1998, Pettersson and Larsen., 2000], KRONOS [Yovine, 1997, Daws et al., 1995], Rabbit [Beyer et al., 2003], and VerICS [Nabialek et al., 2004].

- **Other model checkers**. Other general-purpose model checkers for temporal logic include:

  - the model checker VIS (Verification Interacting with Synthesis) [Brayton et al., 1996], which is presented as "a system for formal verification, synthesis, and simulation of finite state systems". VIS supports the verification of **CTL** and simulation using bounded model checking and OBDD-based techniques.
  - the model checker SAL (Symbolic Analysis Laboratory) [Moura et al., 2004] is developed by SRI International [Moura et al., 2006]. SAL implements OBDD-based and bounded model checking techniques.

2.2.5 Complexity results for model checking

Traditionally, given a formula φ and a model $M = (S, R, V)$, the complexity of temporal logics model checking is expressed as a function of the size of the model $|M|$ and of the size of the formula φ, where $|M|$ is defined as the sum of the number of states in S and the number of elements in R, while $|\varphi|$ is defined as the number of symbols appearing in φ.

CTL. The algorithm presented in Figure 2.4 provides an upper bound for the complexity of **CTL** model checking. Indeed, the algorithm always terminates after at most $|M| \cdot |\varphi|$ steps, and thus model checking for **CTL** is in **P** [Clarke et al., 1986]. A proof for the **P**-hardness of **CTL** model checking is presented in [Schnoebelen, 2003], which allows to conclude that the problem of model checking for **CTL** is **P**-complete.

Logic	Complexity
CTL [Clarke et al., 1986, Schnoebelen, 2003]	P-complete
LTL [Sistla and Clarke, 1985]	PSPACE-complete
CTL* [Clarke et al., 1986, Sistla and Clarke, 1985]	PSPACE-complete
μ-calculus [Kupferman et al., 2000]	$\in$ NP $\cap$ co-NP

Table 2.7: The complexity of model checking for some temporal logics.

LTL. The complexity of **LTL** model checking was investigated in [Sistla and Clarke, 1985]. Given a formula φ and a model M, the problem of model checking for **LTL** is reduced to the satisfiability problem for an **LTL** formula. Intuitively, an **LTL** formula φ_M is constructed in polynomial time to encode the valid runs of M, and it is shown that $M \models \varphi$ iff $\varphi_M \implies \varphi$ is a valid **LTL** formula. Therefore, Theorem 2.1.5 provides a **PSPACE** upper bound for **LTL** model checking. A corresponding lower bound is presented in [Sistla and Clarke, 1985], which allows to conclude that the problem of model checking for **LTL** is **PSPACE**-complete.

The complexity of model checking for **CTL**, for **LTL**, and for other temporal logics is presented in Table 2.7.

The results presented above, however, do not apply to the verification of programs written in one of the languages presented in the previous section (i.e., PROMELA, Reactive-Modules, and SMV). In these practical instances, states and relations of temporal models are not listed *explicitly*. Instead, a *compact description* is usually given. Thus, the complexity of model checking needs to be investigated in terms of the size of the formula and of the size of the program representing the model.

Among others, *concurrent programs* [Kupferman et al., 2000] offer a suitable framework to investigate the complexity of model checking when compact representations are used, because various languages can be reduced to concurrent programs. Formally, a program is a tuple $D = \langle AP, AC, S, \Delta, s^0, L\rangle$, where AP is a set of atomic propositions, AC is a set of actions, S is a set of states, $\Delta : S \times AC \to S$ is a transition function, s^0 is an initial state, and $L : S \to 2^{AP}$ is a valuation function. Given n programs $D_i = \langle AP_i, AC_i, S_i, \Delta_i, s^0_i, L_i\rangle$ ($i \in \{1, \ldots, n\}$), a concurrent program $D_C = \langle AP_C, AC_C, S_C, \Delta_C, s^0_C, L_C\rangle$ is defined as the parallel composition of the n programs D_i, as follows:

- $AP_C = \cup_{1 \leq i \leq n} AP_i$;
- $AC_C = \cup_{1 \leq i \leq n} AC_i$;
- $S_C = \prod_{1 \leq i \leq n} S_i$;
- $(s, a, s') \in \Delta_C$ iff
  - $\forall i.1 \leq i \leq n$, if $a \in AC_i$, then $(s[i], a, s'[i]) \in \Delta_i$, where $s[i]$ is the i-th compo-

nent of a state $s \in S$.

– if $a \notin AC_i$, then $s[i] = s'[i]$;

- $L_C(s) = \cup_i L_i(s[i])$.

(in the remainder, the subscript C is dropped when this is clear from the context).

CTL formulae can be interpreted in a (concurrent) program D by using the standard Kripke semantics for **CTL** formulae in a model $M = (S, R, V)$. For this, define the set of states S of M to be set of states S of D, the temporal relation R to be Δ, and the evaluation function V to be L (more details can be found in [Kupferman et al., 2000]). By slight abuse of notation, the term "Kripke models" is used occasionally below when referring to the programs D_i and to D.

The *program complexity* of model checking is defined as the complexity of model checking for a given, fixed formula. Program complexity results for some temporal logics are presented in Table 2.8.

Logic	Program complexity
CTL	**NLOGSPACE**-complete
CTL*	**NLOGSPACE**-complete
μ-calculus	**P**-complete

Table 2.8: Program complexity of model checking for some temporal logics, from [Kupferman et al., 2000].

By analysing the program complexity of model checking concurrent programs, the authors of [Kupferman et al., 2000] obtain complexity results for various temporal logics. Their results are based on the definition of various kind of automata, extending the concepts presented in Section 2.2.2.4. Table 2.9 presents their main results.

Logic	Program complexity	Overall complexity
CTL	**PSPACE**-complete	**PSPACE**-complete
CTL*	**PSPACE**-complete	**PSPACE**-complete
μ-calculus	**EXP**-complete	**EXP**

Table 2.9: Program complexity and complexity of model checking for some temporal logics using concurrent programs [Kupferman et al., 2000].

The complexity of model checking tools and techniques. The results presented above provide a lower bound for the problem of model checking temporal models specified using PROMELA, ReactiveModules, and SMV. Indeed, a generic concurrent program D can be encoded using any of the languages above. For instance, an SMV module can be associated to each program D_i appearing in D, and a similar approach can be employed for PROMELA and ReactiveModules. Upper bound results for model checking some programming languages for multi-agent systems are provided in Section 6.5.5, page 125.

Also, the worst case complexity of model checking tools can be analysed.

- SPIN implements the automata-based techniques presented in Section 2.2.2.4; their complexity is linear in the size of the model [Clarke et al., 1986]. However, the size of the model is exponential in the number of processes used in the PROMELA specification. Thus, although many optimisation techniques have been implemented in SPIN, in the worst case the complexity of model checking a PROMELA program using SPIN requires exponential time.
- SMV and MOCHA use (an extension of) the labelling algorithm presented in Figure 2.4, whose complexity is linear in the size of the model. As above, however, the model has a size exponential in the number of modules employed in the programming language. On average, OBDD-based techniques may reduce the space required by the labelling algorithm (and, thus, the time required to perform model checking), but it was shown [Bryant, 1991] that for certain problems OBDDs have an exponential size, irrespective of the chosen ordering of the variables. Some operations on OBDDs, such as reduction to canonical form and composition, require time linear in the size of the OBDDs; Boolean quantification, instead, may require exponential time. These results imply that in the worst case OBDD-based model checking may require a double-exponential time.
- NuSMV implements SAT-based techniques for **LTL**, as presented in Section 2.2.2.3. It has been shown in [Clarke et al., 2004b] that the Boolean formulae generated using SAT-based techniques may have a number of variables which is exponential in the size of the input program. Algorithms for Boolean satisfiability may require, in the worst case, an exponential time; thus, SAT-based techniques for model checking are, in the worst case, doubly exponential in the size of the input.

2.3 Model checking multi-agent systems: state of the art

This section analyses the literature concerned with model checking for multi-agent systems. Differently from other PhD theses, this section has been placed after the introduction of theoretical preliminaries, syntax, and notation. This choice is motivated by the technical content of the review.

Different approaches have been proposed for the problem of verification in multi-agent systems using model checking. This section summarises a selection from the recent literature, including both theoretical results and presentations of tools.

2.3.1 Theoretical investigations

1. A model checking procedure for multi-agent systems is presented in [Benerecetti et al., 1998]. Here agents are modelled using MultiAgent Temporal Logic (**MATL**).

   MATL is the fusion of the temporal logic **CTL** and the logic **HML** (Hierarchical MetaLogic) to represent beliefs, desires and intentions. HML is defined as follows. Let I be a set of agent, and $O = \{B, D, I\}$ be a set of symbols, representing the attitudes Belief, Desire, Intention. Let $OI^* = (O \times I)^*$: an element $\alpha \in OI^*$ is a sequence of attitude-agent pairs, and it represents a possible nesting of attitudes. Each $\alpha \in OI^*$ is called a *view*, including the empty string ϵ representing the view of an "external observer". An agent "is a tree rooted in the view that the external observer has of it" (notice that the view that an agent has of another agent can be different from the agent itself). A logical language L_α is associated to each view α. Each language is used to express what is true in the representation corresponding to α. It is imposed that $O_i\varphi$ is a formula of L_α iff φ is a formula of $L_{O_i\alpha}$. The semantics of $\{L_\alpha\}_{\alpha \in OI^*}$ is given by means of "*trees*". A tree is a subset M_α of the set of possible interpretation of a language L_α. Namely, each interpretation is denoted with $t_\alpha \in M_\alpha$, and a tree is a set $\{t_\alpha\}_{\alpha \in OI^*}$. A *compatibility relation* T is a set of trees. A tree satisfies a formula at a view iff the formula is satisfied by all the elements that the tree associates to the view. A *Hierarchical MetaStructure* (HM Structure) is a set of trees T on L_α closed under containment such that there is a $t \in T$ with $t_\epsilon \neq \emptyset$, if t_α satisfies $O_i\varphi$, then $t_{O_i\alpha}$ satisfies φ, and if for all $t' \in T$, $t'_\alpha \in t_\alpha$ implies that $t_{\alpha O_i}$ satisfies φ , then t_α satisfies $O_i\varphi$. **MATL** structures (i.e., models) are a particular kind of HM structures: each language L_α is a **CTL** language. This allows for the interpretation of formulae of a language that includes BDI and temporal (**CTL**) operators.

   The model checking algorithm for **MATL** is essentially an extension of the labelling algorithm for **CTL** taking into account the appropriate data structures for the manipulation of **MATL** views.

2. A methodology to model check knowledge and time is presented in [Hoek and Wooldridge, 2002]. This paper presents the logic $\mathbf{CKL}_n$, obtained from the fu-

sion of **LTL** with $\mathbf{S5}_n$ to reason about knowledge, with the addition of an operator to reason about common knowledge in a group of agents. Formulae of $\mathbf{CKL}_n$ are evaluated over the *runs* of a multi-agent system, expressed in the formalism presented in Section 2.1.7.1 but without explicit references to actions and protocols. The methodology introduced in [Hoek and Wooldridge, 2002] relies on the concept of *local propositions*. Given an interpreted system IS, let $\mathcal{R}$ denote the set of runs of IS; let v, w be two integer numbers, representing the time (i.e., the number of steps) elapsed from an initial state; a pair (r, v), where $r \in \mathcal{R}$ is a possible run, is called a *point* (notice that points have been denoted with the term *global states* in Section 2.1.7.1). A formula φ of $\mathbf{CKL}_n$ is said to be *propositional* if it involves no modal operators. A propositional formula φ is *local to Agent i* iff, for all points $(r, v), (r', w)$, if $(r, v) \sim_i (r', w)$, then $IS, (r, v) \models \varphi$ iff $IS, (r', w) \models \varphi$. [Hoek and Wooldridge, 2002] show that, using local propositions, the problem of $\mathbf{CKL}_n$ model checking can be reduced to the problem of **LTL** model checking. The key idea is to reduce $\mathbf{CKL}_n$ formulae involving the knowledge operator to a pure temporal formula. For instance, the formula $K_i\varphi$ is translated into the formula $G(\psi \implies \varphi)$, where ψ is an appropriate propositional formula local to agent i. A concrete scenario (the bit transmission problem, see Section 6.1) is verified in [Hoek and Wooldridge, 2002] using the programming language PROMELA and the model checker SPIN.

The main limitation of this approach is that the local propositions needed for the translation from $\mathbf{CKL}_n$ to **LTL** cannot be computed automatically, and must be provided by the user.

3. The problem of model checking knowledge and time is also explored in [Meyden and Shilov, 1999]. The paper considers the class of interpreted systems with *perfect recall*. Intuitively, in a system with perfect recall each agent keeps a complete record of all the (local) states he passes through. Formally, using the notion of points defined above, in interpreted systems with perfect recall the local state of an agent i at the point (r, v), denoted with $r_i(v)$, is a tuple $r_i(v) = (r_i(0), \ldots, r_i(v))$. Additionally, the concept of *synchronous systems* is often associated with that of perfect recall: an interpreted system is said to be *synchronous* if, for every agent i, if $(r, v) \sim (r', w)$, then $v = w$. [Meyden and Shilov, 1999] analyse the problem of model checking the logic $\mathbf{CKL}_n$ and some of its restrictions in synchronous and asynchronous interpreted systems with perfect recall. Let $\mathcal{L}_{X,U,K_1,\ldots,K_n,C}$ denote the full language of $\mathbf{CKL}_n$. Similarly, let $\mathcal{L}_{X,K_1,\ldots,K_n,C}$ be the restriction of the previous language without the "Until" operator, let $\mathcal{L}_{K_1,\ldots,K_n,C}$ be the restriction without temporal operators, and let $\mathcal{L}_{X,U,K_1,\ldots,K_n}$ denote the restriction without the operator for common knowledge. The model checking technique presented in [Meyden and Shilov, 1999] extends the automata-based model checking technique for **LTL** by employing a family of Büchi automata, which permits the verification of the epistemic operators. Complexity results for model checking various perfect recall semantics are obtained by analysing the automata-based technique; results are presented in Table 2.10.

Language	**Complexity**
$\mathcal{L}_{K_1,\ldots,K_n,C}$, synchronous	PSPACE-hard
$\mathcal{L}_{K_1,\ldots,K_n,C}$, asynchronous	undecidable
$\mathcal{L}_{X,K_1,\ldots,K_n,C}$, synchronous	PSPACE-complete
$\mathcal{L}_{X,U,K_1,\ldots,K_n}$, synchronous	non-elementary
$\mathcal{L}_{X,U,K_1,\ldots,K_n,C}$, synchronous	undecidable

Table 2.10: Complexity of model checking for some perfect recall semantics.

4. The ideas presented in the previous item have been extended further in [van der Meyden and Su, 2004], where an OBDD-based algorithm for the verification of synchronous interpreted systems with perfect recall is introduced. Their algorithm accepts the class of formulae of $\mathbf{CKL}_n$ whose structure is $X^k(K_i p)$ (where p is an atomic proposition and X^k denotes a concatenation of k temporal operators X of **LTL**). It is shown that the problem of model checking this class of formulae in synchronous interpreted systems can be reduced to the verification of the equivalence of Boolean formulae. Similarly to OBDD-based model checking for **CTL**, all the operations employed in the computation of the Boolean formulae can be carried out on their representation using OBDDs. This methodology is applied to the verification of the protocol of the dining cryptographers (see Section 6.2). Experimental results are presented for this example (see below the tool MCK for further progress along these lines). The main issue with this approach is its limitation to a restricted class of formulae and to a restricted subclass of interpreted systems (synchronous with perfect recall). Nevertheless, this technique may be more efficient than others when a scenario may be suitably formalised using synchronous interpreted systems with perfect recall. See Section 6.5 for further details and for a discussion about this example.

5. Bounded model checking techniques for **CTL** [Penczek et al., 2002] have been extended to the universal fragment of **CTLK** in [Penczek and Lomuscio, 2003]. In this work, universal **CTLK** formulae are evaluated in interpreted systems, as presented in Section 2.1.7.1. To this end, a *bounded* semantics for **CTLK** is defined using *k-models*. Intuitively, a k-model is a structure obtained by taking all the possible runs of length k in a given interpreted system. Let M_{IS}^k be the k model associated to a given interpreted system IS, and let M_{IS} be the standard model associated to IS. It is proven in [Penczek and Lomuscio, 2003] that for every **CTLK** formula φ, $M_{IS} \models \varphi$ iff $M_{IS}^k \models \varphi$ for some $k \leq |M|$. When φ is a formula of the universal fragment of **CTLK**, the problem of verifying whether or not $M_{IS}^k \models \varphi$ can be reduced to the problem of verifying the satisfiability of a Boolean formula $[M^{\varphi}]_k \wedge [\varphi]_k$ (similarly to bounded model checking for **LTL** and **CTL**, the Boolean formula is the conjunction of two Boolean formulae representing, respectively, the Boolean encoding of M_{IS}^k and of φ). Further works along this line include [Woźna et al., 2004] extending the technique to the verification of correct behaviour, and [Woźna et al.,

2005] which introduces a technique for the verification of knowledge in real-time systems. These approaches have been implemented in VerICS, a model checker for multi-agent systems (see below).

6. [Otterloo et al., 2003] and [Jamroga, 2004b] present model checking techniques for epistemic extensions of **ATL**. Their approach is similar, in spirit, to [Hoek and Wooldridge, 2002] in reducing the problem of model checking for an "extended" logic to a less expressive logic for which model checking tools and techniques exists. In particular, [Otterloo et al., 2003] introduce *Turn-Based Epistemic Systems* (TBES), an extension of Turn-Based Systems [Alur et al., 2002, Hoek and Wooldridge, 2003a] to evaluate formulae of the logic **ATEL** (see Section 2.1.7.3). It is shown that the problem of model checking an **ATEL** formula φ in a TBES $\mathcal{T}$ can be reduced to the problem of verifying an **ATL** formula ψ in a Turn-Based system $\mathcal{S}$, and a methodology is provided to compute the reduction. This technique provides a methodology that enables the application of the MOCHA model checker (see Section 2.2.3.2) to the verification of **ATEL**.

   The reduction technique presented in [Jamroga, 2004b] differs from the previous one in that it allows for the verification of the full language of **ATEL**, including the "Until" operator, and operators for group epistemic properties. As in the previous case, the problem of model checking an **ATEL** formula is reduced to the verification of an **ATL** one. However, no implementation or examples are provided.

   The main limitation of both approaches is the need of manual intervention in the reduction process from **ATEL** to **ATL**. While the reductions seem feasible for small examples, handling large scale examples manually may be impractical.

2.3.2 Model checking tools for multi-agent systems

1. A language for the specification of multi-agent systems, called MABLE, is proposed in [Wooldridge et al., 2002]. MABLE combines agents' distinctive notions (such as beliefs, desires, etc.) with traditional-style programming constructs. A MABLE program consists of a number of agents, each characterised by a unique identifier and by a "program body" describing the agent's behaviour. Standard constructs such as loops and if-then-else sequences can be used in the program body of an agent. In addition, the body of a MABLE agent may include constructs to describe the behaviour of an agent when the agent is "unsure" about the truth value of some logical expression; this is achieved by using constructs of the form `if` φ `then` P_1 `else` P_2 `unsure` P_3. MABLE agents are allowed to perform actions resulting in modification of the environment in which they live: this ability is captured by the MABLE instruction `do` α, where α is an element of the set of actions available to the agent (including actions for communication with other agents). The evolution of MABLE agents is defined by the parallel composition of programs appearing in

the body of each agent.

Each MABLE program may include a number of *claims* to encode the required properties to be verified; claims are expressed using a subset of the syntax of LORA (see Section 2.1.7). Verification of MABLE programs is performed using the model checker SPIN: the MABLE compiler generates PROMELA code for SPIN by associating a process (see Section 2.2.3) to each agent. LORA claims appearing in the MABLE code are reduced to **LTL** claims using a technique similar to the one presented in [Hoek and Wooldridge, 2002] using local propositions. To evaluate these “reduced” formulae, the PROMELA code is annotated with propositions corresponding to agents’ attitudes (beliefs, desires, etc.). Similarly to [Benerecetti et al., 1998], desires, and intentions are treated as nested data structures in the MABLE framework.

2. The approach of [Bordini et al., 2003b, Bordini et al., 2003a] is similar, in spirit, to the one presented in the previous point. Indeed, the problem of model checking a language for multi-agent systems is reduced to the verification of an **LTL** formula in a PROMELA model using SPIN. The language AgentSpeak(F) was introduced in [Bordini et al., 2003b] as a restriction of the language AgentSpeak(L); the latter is a language used to formalise a multi-agent system expressed using Rao and Georgeff’s BDI logic (see Section 2.1.7). AgentSpeak(F) is the restriction of AgentSpeak(L) to finite state systems, with further constraints on first order quantifications and other technical requirements. The main difference between MABLE and AgentSpeak(F) is that AgentSpeak(F) is a logic programming language (*à la* Prolog), while MABLE is an imperative programming language (*à la* C); we refer to [Bordini et al., 2003b] for further details.

   The translation from AgentSpeak(F) and BDI specifications (using Rao and Georgeff’s BDI logic) to a PROMELA program and an **LTL** specification is performed automatically by a translator called CASP (Checking AgentSpeak Programs, [Bordini et al., 2003a]). As in the case of MABLE, BDI modalities are evaluated using nested data structures.

3. MCK (Model Checking Knowledge) is a software tool developed by Ron van der Meyden et al. [Gammie and van der Meyden, 2004] implementing the model checking techniques presented in [Meyden and Shilov, 1999, van der Meyden and Su, 2004] (see above). MCK differs from the two tools listed above in that it is “self-contained”, i.e., it does not reduce the problem of model checking for multi-agent systems to a temporal-only problem to take advantage of existing model checkers. Instead, MCK uses OBDDs to represent models symbolically, and permits the verification of **CTL** and **LTL** formulae extended with epistemic operators for single agents and for groups of agents. MCK makes use of an input language to describe the *environment* in which agents interact, observation functions for the agents to define which part of

the environment each agent can observe[20], the agents' behaviour using actions, the set of initial states, fairness constraints, and formulae to be checked. The structure of the input language of MCK is summarised in Figure 2.17: shared variables describing the environment are declared first; the initial state of the system is described by a Boolean expression involving environment variables. Agents are declared using an identifier followed by the name of a protocol (see below) and by a list of variables of the environment that the agent is allowed to "observe". The protocols of the agents are defined after the declaration of the transitions for the variables of the environment and an optional **CTL** formula encoding fairness conditions. Each protocol defines the actions taken by each agent (e.g., writing some value in a shared variable). The syntax of the formulae to be checked depends on the <specification-type> chosen: using appropriate keywords it is possible to verify formulae under the assumption that:

(a) The local state of an agent is defined by the observable environment variables only;
(b) The local state of an agent is defined by the observable environment variables and by the value of the clock (this corresponds to a synchronous interpreted systems);
(c) The local state of an agent is defined by the observable environment variables, by the value of the clock, and by all the past observations (this corresponds to a synchronous interpreted system with perfect recall).

In the first case, either the full syntax of **CTLK**, or **LTL** extended with epistemic operators can be verified. In the second case, only formulae involving the "Next" operator (both for **CTL** and for **LTL**) and the knowledge of a single agent can be verified. In the third case, only the **LTL** "Next" operator and the knowledge of a single agent can be verified. MCK is written in Haskell and the available package includes an implementation of the protocol of the dining cryptographers (see Section 6.2).

4. VerICS [Nabialek et al., 2004] is a software tool for the verification of multi-agent systems. VerICS accepts various input formalisms, including a subset of the Estelle language (an ISO standard for the description of communicating processes), networks of timed automata, and timed Petri nets. Interpreted systems can be encoded using a network of un-timed automata by associating an automaton the each agent, and formulae of $\mathbf{CTLKD}^{D,C}$ can be interpreted in networks of automata. Thus, VerICS can be employed in the verification of interpreted systems.

   VerICS implements three different model checking techniques: bounded model checking, un-bounded model checking, and on-the-fly model checking using abstract mod-

[20]In this formalism the observation function plays a role similar to the local states of interpreted systems (see Section 2.1.7.1).

```
– Type declarations
type TypeName1 = { ... }
...

– Shared variables
varname1 : TypeName1

init_cond = [ Boolean expression with variables above ]

– Agent declarations
agent AgentName1 ”protocol1” ( ... env variables ... )
...
agent AgentNameN ”protocolN” ( ... env variables ... )

– Transitions
transitions
begin
  ...
end

fairness = [ CTL formula ]

– Formulae to be verified
<specification-type> = temporal-epistemic formula

– Protocol declarations
protocol ”protocol1” ( ... env variables ... )
begin
  ...
end
...
protocol ”protocolN” ( ... env variables ... )
begin
  ...
end
```

Figure 2.17: Structure of the MCK input language [Gammie and van der Meyden, 2004].

els (abstract models are obtained from the original model by combining into equivalence classes the set of states of the original model). These techniques are implemented as separate modules for VerICS and offer different model checking capabilities. The bounded model checking module permits the verification of the universal fragment of the logic $\mathbf{CTLKD}^{D,C}$; the un-bounded model checking module permits the verification of the full language of $\mathbf{CTLKD}^{D,C}$; the on-the-fly module permits the verification of reachability properties (but it cannot be used in the verification of epistemic modalities).

More details on VerICS and its applications to multi-agent systems can be found in the paper [Kacprzak et al., 2006]; Section 6.5 presents further details of VerICS and a comparison with MCMAS.

Chapter 3

Model checking multi-agent systems using OBDDs

This chapter presents possible solutions to the problem of model checking multi-agent systems represented in the formalism of interpreted systems. Section 3.1 defines the problem of model checking deontic interpreted systems, while Section 3.2 presents a possible solution using existing tools. A self-contained solution to the model checking problem is presented in Section 3.3, where algorithms are provided for the verification of formulae in interpreted systems and a translation is defined to enable the use of OBDDs in the verification process.

3.1 Problem definition

As defined in Section 2.1.7.1, an interpreted system is a tuple:

$$IS = \left\langle (L_i, Act_i, P_i, t_i)_{i \in \{1,\dots,A\}}, (L_E, Act_E, P_E, t_E), I, V \right\rangle$$

where A is the number of agents in the system. Each agent i is characterised by a set of local states (L_i), a set of actions (Act_i), a protocol ($P_i : L_i \rightarrow 2^{Act_i}$), and an evolution function ($t_i : L_i \times L_E \times Act \rightarrow L_i$, where $Act = Act_1 \times Act_n \times Act_E$). The agent E is a special agent (the environment) whose local states are "public" (this is reflected by the agents' evolution functions). As described in Chapter 6, agent E may be used to capture special requirements of the environment with and in which agents interact. The presence of E is not strictly necessary to formalise all the examples: in the remainder of this thesis, the environment will be treated as a standard agent, the subscript E will be dropped and, by convention, the number of agents will be denoted by n (which may or may not include E. Notice that, when an agent is present, each evolution function t_i also depends on L_E: this scenario should be clear from the context). The Cartesian product of the local states of the agents is denoted by $S = \bigtimes_{i=1}^{n} L_i$; elements of S are tuples of local states, called global states. Given $g \in S$, $l_i(g)$ denotes the i-th component of g, i.e., the local state of agent i in global state g. In the definition of IS above, the set $I \subseteq S$ is the set of initial global states. Starting from the set of initial states, the protocols and the evolution functions define a set of states that are reachable, denoted by G. Given a set of atomic propositions AP, the evaluation relation $V \subseteq S \times AP$ completes the definition of an interpreted system[1]. As presented in Section 2.1.7.2, interpreted systems can be extended to deontic interpreted systems by partitioning the set of local states L_i into two sets: a non-empty set G_i of allowed states, and a set R_i of disallowed states, such that $L_i = G_i \cup R_i$, and $G_i \cap R_i = \emptyset$.

Formulae of $\mathbf{CTLKD} - \mathbf{A}^{\{D,C\}}$ can be interpreted in a deontic interpreted system DIS by associating a Kripke model M_{DIS} to DIS (see Sections 2.1.7.1, 2.1.7.2, and 2.1.7.3).

Given a deontic interpreted system DIS and a $\mathbf{CTLKD} - \mathbf{A}^{\{D,C\}}$ formula φ, the aim of this chapter is to define effective procedures to establish whether or not

$$DIS \models \varphi.$$

3.2 Explicit verification using NuSMV and Akka

A possible solution to the model checking problem for interpreted system may be provided using existing tools. In particular, NuSMV can be used in conjunction with a model verifier (see below) to perform verification. Given an interpreted system IS, verification using these tools is performed as follows:

- An SMV program is defined, consisting of a single `main` module. A variable is introduced for each set of local states and for each set of actions; for instance, if $L_1 = \{l_1^1, l_1^2, l_1^3\}$, the SMV variable `L1` is defined such that `L1 : {l1-1, l1-2,l1-3};` (and similarly for actions). The SMV construct `next` is used to synchronise the value of the variables encoding actions with the value of the variables for local states, following the protocols P_i. For instance, let P_1 be a protocol such that $P_1(l_1^1) = \{a_1^1, a_1^2\}$ and $P_1(l_1^2) = \{a_1^3\}$. The corresponding SMV code is as follows:

```
next(Act1) = case
                L1 = l1-1 : {a1-1,a1-2};
                L1 = l1-2 : {a1-3};
```

[1]For the purposes of this thesis, the evaluation relation V needs to be defined for the set of reachable states only, i.e., $V \subseteq G \times AP$.

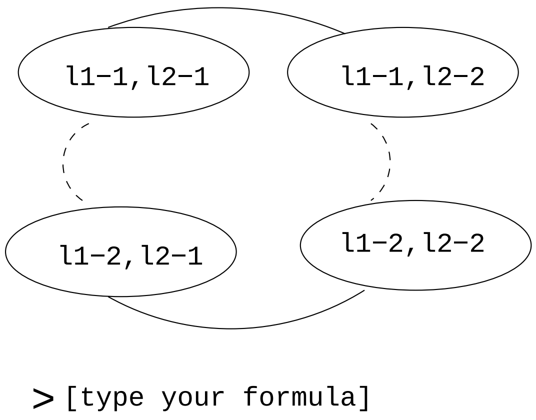

Figure 3.1: Akka screen-shot.

```
        esac;
```

In this way actions performed in a given local state are mapped in the “next” SMV state (this approach is usually denoted by the term “post-projection”). The evolution functions t_i are translated into appropriate SMV `next` constructs.

- The SMV code described in the previous point is used to compute the set of reachable states and actions. The source code of NuSMV can be modified to print explicitly this set in the form of a list of tuples (see [Lomuscio et al., 2003] for details), and this list can be parsed to obtain the list of reachable global states. A possible output is represented below:

```
      ⋮
 -----------
l1-1,l2-1,l3-1
 -----------
l1-1,l2-2,l3-1
 -----------
l1-2,l2-2,l3-2
 -----------
      ⋮
```

- Akka [Hendrik, 2006] is a Kripke model editor that supports the verification of multi-modal formulae in the edited models. A simple Akka screen-shot is depicted in Figure 3.1. The list of reachable states, appropriately parsed, is fed into Akka, where epistemic-only formulae can be verified.

Examples whose size is limited to a few dozens of reachable states can be checked using this methodology. For instance, the bit transmission problem with a faulty receiver has 32 reachable states (see Section 6.1) and it is verified in [Lomuscio et al., 2003]. In general, the methodology is particularly useful for small examples and for didactic purposes, because it is very quick to implement, the state space is clearly visible, and counterexamples for false formulae can be understood graphically.

The main limitation of this approach is the restriction of the verification to either epistemic-only formulae using Akka, or to temporal-only formulae using NuSMV. Additionally, the methodology requires manual intervention in the definition of the SMV code, and it is not fully symbolic: the set of reachable states is dealt with explicitly by Akka. Therefore, this methodology may suffer from scalability issues in large examples.

3.3 Symbolic model checking of interpreted systems using OBDDs

This section presents a self-contained methodology for the verification of temporal, epistemic, correctness, and strategic modalities in deontic interpreted systems using OBDDs. First, a deontic interpreted system is encoded using Boolean variables and Boolean formulae; then, a verification algorithm that can operate on this representation is built progressively for model checking temporal, epistemic, and correct behaviour operators, and operators for strategies.

3.3.1 Boolean encoding of deontic interpreted systems

Let DIS be a deontic interpreted system for a set of n agents $\{1, \ldots, n\}$ (this set may include a special agent modelling the environment, see Section 3.1):

$$DIS = \left\langle (G_i, R_i, Act_i, P_i, t_i)_{i \in \{1,\ldots,n\}}, I, V \right\rangle$$

Let $L_i = G_i \cup R_i$ denote the set of local states for agent i, let $S = \bigtimes_{i=1}^{n} L_i$, and let AP be a set of atomic propositions. In the remainder of this section the relation $V \subseteq S \times AP$ is represented by means of a function $V : AP \to 2^S$.

The number of Boolean variables required to encode the set of local states L_i for agent i is denoted by $nv(i)$, and it is computed by taking the logarithm of the number of local states in L_i, that is $nv(i) = \lceil log_2 |L_i| \rceil$. Thus, a global state g can be associated with a vector of Boolean variables $(x_1, \ldots, x_N)$, where $N = \sum_{i=1}^{n} nv(i)$:

$$g \rightarrow (\underbrace{x_1, \ldots, x_{nv(1)}}_{\text{variables for } L_1}, \ldots, \underbrace{x_j, \ldots, x_{j+nv(k)}}_{\text{variables for } L_k}, \ldots, x_N)$$

(where $j = \sum_{i<k} nv(i)$).

The number of Boolean variables required to encode the set of actions Act_i for agent i is denoted by $na(i)$, and it is computed by $na(i) = \lceil log_2 |Act_i| \rceil$. Similarly to global states, a joint action $\alpha \in Act = \bigtimes_{i=1}^{n} Act_i$ can be univocally identified by a vector of Boolean variables $(a_1, \ldots, a_M)$, where $M = \sum_{i=1}^{n} na(i)$.

In turn, every Boolean vector can be identified with a Boolean formula represented by a conjunction of literals, i.e., a conjunction of Boolean variables or their negation. For instance, the Boolean vector $(1, 0, 0, 1)$ is identified with the Boolean formula $f(x_1, x_2, x_3, x_4) = x_1 \wedge \neg x_2 \wedge \neg x_3 \wedge x_4$. Given a set of global states $Q = \{g_1, \ldots, g_k\}$, the set Q can be univocally identified with a Boolean formula f_Q: the formula f_Q is obtained by taking the disjunction of the Boolean formulae encoding each state $g_i \in Q$. For instance, if $Q = \{g_1, g_2\}$, $g_1 \rightarrow (1, 0, 0, 1)$, and $g_2 \rightarrow (1, 1, 0, 0)$, then $f_Q = (x_1 \wedge \neg x_2 \wedge \neg x_3 \wedge x_4) \vee (x_1 \wedge x_2 \wedge \neg x_3 \wedge \neg x_4)$. A similar technique can be applied to sets of joint actions, which can be expressed as disjunction of Boolean formulae.

The encoding of local states, global states, and actions by means of Boolean formulae permits the definition of Boolean formulae encoding agents' protocols. As defined in Section 2.1.7.1, a protocol P_i for agent i is a function $P_i : L_i \rightarrow 2^{Act_i}$. Let $bf(Q)$ be a function that, given a set Q of states or actions (either global or local), returns the appropriate Boolean function encoding the set. The Boolean formula f_{P_i} encoding the protocol for agent i is defined by:

$$f_{P_i} = \bigvee_{l_i \in L_i} \left[bf(\{l_i\}) \wedge \left(\bigoplus_{a \in P_i(l_i)} bf(\{a\}) \right) \right].$$

Intuitively, this formula restricts for each local state (encoded by $bf(\{l_i\})$ the set of actions that can be performed by agent i. The formula $\bigoplus_{a \in P_i(l_i)} bf(\{a\})$ is used to capture the requirement that a single action has to be performed in a given local state when $|P(l_i)| > 1$ for some $l_i \in L_i$. Formally, $\bigoplus_{a \in P_i(l_i)} bf(\{a\})$ imposes that only one of the Boolean formulae encoding the actions $a \in P_i(l_i)$ is true. If $|P_i(l_i)| = 2$, this is expressed using the exclusive-or (x-or) of two Boolean formulae. For $|P_i(l_i)| > 2$, the formula is a generalisation of the x-or operator. A "joint" protocol is obtained by taking the conjunction of the Boolean formula encoding the agents' protocols:

$$f_P(x_1, \dots, x_N, a_1, \dots, a_M) = \bigwedge_{i=1}^{n} f_{P_i}$$

The evolution functions $t_i : L_i \times L_E \times Act \to L_i$ are translated into Boolean functions by introducing a new set of "primed" Boolean variables $(x'_1, \dots, x'_N)$ to encode the successor of a state (either local or global). A generic pair $t_i(l_p, a_q) = l_r$ (with $l_p, l_r \in L_i$ and $a_q \in Act_i$) is translated into the Boolean function $bf(\{l_p\}) \wedge bf(\{a_q\}) \wedge bf'(\{l_r\})$, and the Boolean formula f_{t_i} corresponding to t_i is obtained by taking the disjunction of all the possible such pairs[2]. The Boolean formula f_t corresponding to the global evolution function t defined in Section 2.1.7.1, is defined as:

$$f_t(x_1, \dots, x_N, a_1, \dots, a_m, x'_1, \dots, x'_N) = \bigwedge_{i=1}^{n} f_{t_i}.$$

(where the arguments of t_i are the appropriate Boolean variables to encode L_i, as defined above).

The set I of initial global states can be translated into the Boolean function f_I, defined as the disjunction of the Boolean formulae encoding the global states in I, i.e.:

$$f_I(x_1, \dots, x_N) = \bigvee_{g \in I} bf(\{g\}).$$

In the following sections the evaluation function $V : AP \to 2^{AS}$ is associated with a function $f_V : AP \to \mathcal{B}(x_1, \dots, x_N)$ from atomic proposition to the set $\mathcal{B}(x_1, \dots, x_N)$ of Boolean functions over the Boolean variables $(x_1, \dots, x_N)$. Formally, given an atomic proposition $p \in AP$, $f_V(p)$ is the Boolean function encoding the set of global states in which the atomic proposition p holds.

The Boolean functions f_{P_i}, f_P, f_{t_i}, f_t, f_I, and the function f_V are used in the next subsections to compute the set of global states in which a $\mathbf{CTLKD} - \mathbf{A}^{D,C}$ formula φ holds, denoted by $[\![\varphi]\!]$.

3.3.2 Model checking temporal properties

The aim of this section is to present a procedure to compute the set of states (represented as a Boolean function) of a deontic interpreted system *DIS* in which a temporal-only

[2]Notice that this approach would require an explicit listing of all the possible local states and actions. In the implementation of the algorithm this issue is avoided by introducing a form of default values for t_i; see Chapter 5.

formula φ holds. To this end, a temporal transition relation $R_t \subseteq S \times S$ is defined to encode that a joint action $a \in Act$ exists such that two global states are related via the the evolution function t; formally,

$$R_t(g, g') \text{ iff } \left(\exists a \in Act \text{ such that } t(g, a) = g'\right).$$

The temporal relation R_t can be translated into a Boolean function $f_{R_t}(x_1, \dots, x_N, x'_1, \dots, x'_N)$ by quantifying over the Boolean variables encoding joint actions. Using a standard technique, Boolean quantification is translated into a propositional formula:

$$f_{R_t}(x_1, \dots, x_N, x'_1, \dots, x'_N) = \bigvee_{\bar{a} \in \{0,1\}^M} f_t(x_1, \dots, x_N, a_1, \dots, a_M, x'_1, \dots, x'_N) \tag{3.1}$$

(where $\bar{a}$ is a vector of Boolean variables of the form $(a_1, \dots, a_M)$).

The set of global states S, the temporal relation R_t, and the evaluation function V constitute a model $M = (S, R_t, V)$ for the logic **CTL**[3]. Therefore, the labelling algorithm presented in Section 2.2.2.1 can be applied to the computation of the set of states $[\![\varphi]\!] \subseteq S$ in which a formula φ holds. The labelling algorithm $\textsc{mc}_{\mathbf{CTLK}}$ to compute $[\![\varphi]\!]$ is reported in Figure 3.2. The main differences between this algorithm and the one in Figure 2.4 are:

- The set of states computed by $\textsc{mc}_{\mathbf{CTLK}}(\varphi, DIS)$ is a set of global states. Differently from states in a **CTL** model, global states have a further level of detail, being tuples of local states.
- The temporal transition relation R_t is defined in terms of agents' actions, protocols, and transition relations. The Boolean quantification over actions in Equation 3.1 hides the underlying structure of the deontic interpreted systems when reasoning about temporal operators only.

Similarly to Section 2.2.2.1, let $\text{pre}_\exists$ denote a procedure that, given a set of global states $X \subseteq S$, computes the set of global states $Y \subseteq S$ from which a transition is enabled to a global state in X, i.e.:

$$Y = \text{pre}_\exists(X) = \{g \in S | \exists g'.(g' \in X \text{ and } gR_tg')\}.$$

The procedure $\text{pre}_\exists$ is used in the algorithm of Figure 3.3 dealing with the operator EX. The remaining procedures $\textsc{mc}_{\mathbf{CTLK},EG}$ and $\textsc{mc}_{\mathbf{CTLK},EU}$ are defined analogously to the procedures presented in Figures 2.6 and 2.7.

As noted in Section 2.2.2.2 and thanks to Equation 3.1, all the operations on sets of

[3]In a **CTL** model the temporal transition relation R_t is usually required to be serial. This requirement reduces to a requirement of seriality for the evolution function t in DIS.

```
$\textsc{mc}_{\mathbf{CTLK}}(\varphi, DIS)$ {
  $\varphi$ is an atomic formula: return $V(\varphi)$;
  $\varphi$ is $\neg\varphi_1$: return $S \setminus \textsc{mc}_{\mathbf{CTLK}}(\varphi_1, DIS)$;
  $\varphi$ is $\varphi_1 \vee \varphi_2$: return $\textsc{mc}_{\mathbf{CTLK}}(\varphi_1, DIS) \cup \textsc{mc}_{\mathbf{CTLK}}(\varphi_2, DIS)$;
  $\varphi$ is $EX\varphi_1$: return $\textsc{mc}_{\mathbf{CTLK},EX}(\varphi_1, DIS)$;
  $\varphi$ is $EG\varphi_1$: return $\textsc{mc}_{\mathbf{CTLK},EG}(\varphi_1, DIS)$;
  $\varphi$ is $E[\varphi_1 U \varphi_2]$: return $\textsc{mc}_{\mathbf{CTLK},EU}(\varphi_1, \varphi_2, DIS)$;
}
```

Figure 3.2: The labelling algorithm for the temporal fragment of **CTLK**.

```
$\textsc{mc}_{\mathbf{CTLK},EX}(\varphi, DIS)$ {
  $X = \textsc{mc}_{\mathbf{CTLK}}(\varphi, DIS)$;
  $Y = \text{pre}_{\exists}(X)$;
  return $Y$;
}
```

Figure 3.3: The support procedure $\textsc{mc}_{\mathbf{CTLK},EX}(\varphi, DIS)$.

global states appearing in the algorithm of Figure 3.2 may be translated into operations on Boolean formulae. Thus, for a given deontic interpreted system *DIS* and for a given temporal-only formula φ, the algorithm $\textsc{mc}_{\mathbf{CTLK}}(\varphi, DIS)$ computes the Boolean formula encoding the set of global states in which φ holds. Similarly to the standard OBDD-based model checking for **CTL**, the Boolean formulae resulting from this algorithm can be manipulated using OBDDs; details of the implementation of this algorithm using OBDDs (and its extensions presented below) are presented in Chapter 5.

3.3.3 Model checking epistemic properties

This section presents model checking procedures for the epistemic operators of $\mathbf{CTLKD} - \mathbf{A}^{D,C}$. Given a deontic interpreted system *DIS*, let $\Gamma \subseteq \{1, \dots, n\}$ denote a group of agents. As defined in Section 2.1.7.1, the set of epistemic operators is composed by an operator K_i to reason about the epistemic states of a single agent, by an operator E_Γ to express what everybody in a group Γ knows, by an operator D_Γ to reason about the distributed knowledge in a group Γ, and by an operator C_Γ representing common knowledge. Epistemic operators are evaluated in a deontic interpreted system by employing the appropriate accessibility relation. In particular, the operators K_i are evaluated using the epistemic accessibility relations $\sim_i \subseteq S \times S$ defined by the equivalence of local states for agent i, while the operators E_Γ, D_Γ, and C_Γ are evaluated using the accessibility relations R^E_Γ, R^D_Γ, and R^C_Γ, respectively. Accessibility relations for group modalities are defined as follows:

$$\begin{aligned} R^E_\Gamma &= \bigcup_{i \in \Gamma} \sim_i; \\ R^D_\Gamma &= \bigcap_{i \in \Gamma} \sim_i; \\ R^C_\Gamma &= \left(R^E_\Gamma\right)^*. \end{aligned}$$

(i.e., R^C_Γ is the transitive closure of R^E_Γ).

The evaluation of epistemic operators has to be restricted to the set of reachable states. Otherwise, an epistemic formula of the form $K_i\varphi$ may result false at a global state g of a deontic interpreted system DIS because a non-reachable global state g' may be accessible from g via the epistemic relation $\sim_i$, with $DIS, g' \not\models \varphi$. Therefore, the set G of reachable global states needs to be computed before evaluating epistemic operators. The set G is obtained by iterating the following operator $\tau : S \to S$:

$$\tau(Q) = I \cup Q \cup \left\{g \in S | \left(\exists g'. \left(\left(g' R_t g\right) \wedge \left(g' \in Q\right)\right)\right)\right\}. \tag{3.2}$$

Intuitively, $\tau(Q)$ includes the set of initial states I, the set Q itself, and the set of global states that are reachable from Q in a single time step. As the set of global states S is finite, and the operator τ is monotonic, τ admits a (least) fix-point, which can be computed by iterating $\tau(\emptyset)$. Notice that all the set operations appearing in Equation 3.2 can be encoded using appropriate Boolean operations on the Boolean formulae encoding sets of global states. Therefore, Equation 3.2 provides a constructive way to compute a Boolean formula f_G encoding the set of reachable global states.

The set of reachable states G is used to compute the set of states in which a formula of the form $K_i\varphi$ holds, as presented in the procedure of Figure 3.4. This procedure first computes the set of states in which the negation of the formula φ holds, and then builds the pre-image of this set with respect to the epistemic accessibility relation $\sim_i$[4]. The set of states satisfying $K_i\varphi$ is finally computed by taking the complement with respect to G of the set obtained in the last step.

The procedures MC$_{\mathbf{CTLK},E,\Gamma}$ and MC$_{\mathbf{CTLK},D,\Gamma}$ for formulae of the form $E_\Gamma\varphi$ and $D_\Gamma\varphi$ are presented in Figures 3.5 and 3.6.

The procedure for common knowledge is based on the characterisation of common knowledge in terms of fix-point. Indeed, the equivalence [Fagin et al., 1995]:

$$C_\Gamma\varphi \Leftrightarrow E_\Gamma(\varphi \wedge C_\Gamma\varphi)$$

[4]The use of the existential quantifier is motivated by its efficient implementation using OBDDs.

```
$\textsc{mc}_{\mathbf{CTLK},K}(\varphi, i, \mathit{DIS})$ {
  X = $\textsc{mc}_{\mathbf{CTLK}}(\neg\varphi, \mathit{DIS})$;
  Y = $\{g \in G \mid \exists g' \in X \text{ s.t. } g \sim_i g'\}$
  return $\neg$Y $\cap G$;
}
```

Figure 3.4: The support procedure $\textsc{mc}_{\mathbf{CTLK},K}(\varphi, i, \mathit{DIS})$.

```
$\textsc{mc}_{\mathbf{CTLK},E}(\varphi, \Gamma, \mathit{DIS})$ {
  X = $\textsc{mc}_{\mathbf{CTLK}}(\neg\varphi, \mathit{DIS})$;
  Y = $\{g \in G \mid \exists g' \in X \text{ s.t. } R^E_\Gamma(g, g')\}$
  return $\neg$Y $\cap G$;
}
```

Figure 3.5: The support procedure $\textsc{mc}_{\mathbf{CTLK},E}(\varphi, \Gamma, \mathit{DIS})$.

```
$\textsc{mc}_{\mathbf{CTLK},D}(\varphi, \Gamma, \mathit{DIS})$ {
  X = $\textsc{mc}_{\mathbf{CTLK}}(\neg\varphi, \mathit{DIS})$;
  Y = $\{g \in G \mid \exists g' \in X \text{ s.t. } R^D_\Gamma(g, g')\}$
  return $\neg$Y $\cap G$;
}
```

Figure 3.6: The support procedure $\textsc{mc}_{\mathbf{CTLK},D}(\varphi, \Gamma, \mathit{DIS})$.

```
$\textsc{mc}_{\mathbf{CTLK}}C(\varphi, \Gamma, DIS)$ {
  X = $\textsc{mc}_{\mathbf{CTLK}}(\varphi, DIS)$;
  Y = $G$;
  while ( X != Y ) {
    X = Y;
    Y = $\{g \in G | \exists g' \in G$ s.t. $g' \in \textsc{mc}_{\mathbf{CTLK}}(\varphi, DIS)$ and $g' \in X$ and $R^E_\Gamma(g, g')\}$
  }
  return Y;
}
```

Figure 3.7: The support procedure $\textsc{mc}_{\mathbf{CTLK},C}(\varphi, \Gamma, DIS)$.

```
$\textsc{mc}_{\mathbf{CTLK},O}(\varphi, i, DIS)$ {
  X = $\textsc{mc}_{\mathbf{CTLK}}(\neg\varphi, DIS)$;
  Y = $\{g \in G \mid \exists g' \in X$ s.t. $g R^O_i g'\}$
  return $\neg$Y $\cap G$;
}
```

Figure 3.8: The support procedure $\textsc{mc}_{\mathbf{CTLK},O}(\varphi, i, DIS)$.

implies that $[\![C_\Gamma \varphi]\!]$ is the fix-point of the (monotonic) operator $\tau_C : S \to S$ defined by $\tau_C(Q) = [\![E_\Gamma(\varphi \wedge (Q))]\!]$. Hence, $[\![C_\Gamma \varphi]\!]$ can be obtained by iterating $\tau_C(G)$.

Similarly to Section 3.3.2, all the operations on sets appearing in the procedures of Figures 3.4–3.7 can be performed on the Boolean representation of sets of global states which, in turn, can be manipulated using OBDDs.

3.3.4 Model checking correct behaviour

This section introduces model checking procedures for formulae of the form $O_i\varphi$ and $\hat{K}^\Gamma_i \varphi$ expressing, respectively, that φ holds when agent i is functioning correctly, and that agent i knows φ under the assumption that agents in Γ are operating correctly. As defined in Section 2.1.7.2, the relation $R^O_i \subseteq S \times S$ is defined by $R^O_i(g, g')$ iff $l_i(g') \in G_i$, i.e., iff the local state of agent i in global state g' is an element of the set of "green" states for agent i. The procedure $\textsc{mc}_{\mathbf{CTLK},O}$ for formulae of the form $O_i\varphi$ has the same structure of the procedure for $\textsc{mc}_{\mathbf{CTLK},K}$ and it is reported in Figure 3.8.

The procedure for formulae of the form $\hat{K}^\Gamma_i \varphi$ is presented in Figure 3.9. The procedure operates similarly to the procedure $\textsc{mc}_{\mathbf{CTLK},K}$, but the computation of the set Y is performed by taking the intersection of the relation R^K_i with all the relations R^O_j such that $j \in \Gamma$ (in the actual implementation, the universal quantification is translated into the conjunction of the Boolean formulae encoding the accessibility relations R^O_i).

```
$\textsc{mc}_{\mathbf{CTLK},KH}(\varphi, i, \Gamma, DIS)$ {
  X = $\textsc{mc}_{\mathbf{CTLK}}(\neg\varphi, DIS)$;
  Y = $\{g \in G \mid \exists g' \in X$ s.t. $R_i^K(g,g')$ and $R_j^O(g,g')$ for all $j \in \Gamma\}$
  return $\neg$Y $\cap G$;
}
```

Figure 3.9: The support procedure $\textsc{mc}_{\mathbf{CTLK},KH}(\varphi, i, \Gamma, DIS)$.

```
$\textsc{mc}_{\mathbf{CTLK}\langle\langle\cdot\rangle\rangle X}(\varphi, \Gamma, DIS)$ {
  Y = $\{g \in G | (\exists a \in Act_\Gamma, g' \in G)$ s.t. $(\forall b \in Act_{\{1,\dots,n\}\setminus\Gamma}).[R_t(g,g')$ and $t(g,(a \cup b),g')$
        and $g' \in \textsc{mc}_{\mathbf{CTLK}}(\varphi, DIS)$ and $(a \cup b)$ is consistent with the protocols in $g]\}$
  return $Y$;
}
```

Figure 3.10: The support procedure $\textsc{mc}_{\mathbf{CTLK},\langle\langle\cdot\rangle\rangle X}(\varphi, \Gamma, DIS)$.

Similarly to the procedures in the previous Sections, the procedures $\textsc{mc}_{\mathbf{CTLK},O}$ and $\textsc{mc}_{\mathbf{CTLK},KH}$ can operate on the Boolean representation of sets of states.

3.3.5 Model checking strategies

Actions appear in the Boolean encoding of a deontic interpreted system through the definition of the Boolean variables $(a_1, \dots, a_M)$. These variables are used in the Boolean encoding of the protocols f_{P_i} and in the Boolean encoding of the evolution functions t_i. A procedure to verify **ATL**-style operators in non-deterministic interpreted systems (see Section 2.1.7.3) is obtained by quantifying over the Boolean variables for actions. Let Γ be a group of agents, let Act_Γ denote the set of joint actions of agents in Γ, i.e., $Act_\Gamma = \bigtimes_{i \in \Gamma} Act_i$. Let $a \in Act_\Gamma$ and let $b \in Act_{\{1,\dots,n\}\setminus\Gamma}$: $a \cup b$ denotes the joint action α in Act whose local components belong to either a or b, with the appropriate reordering of components. Figure 3.10 presents the procedure for verifying formulae of the form $\langle\langle\Gamma\rangle\rangle X\varphi$ using this notation.

Intuitively, the set $[\![\langle\langle\Gamma\rangle\rangle X\varphi]\!]$ is computed by including all the reachable states from which a joint action in Act_Γ exists, such that φ holds in a global state reachable from g, no matter what actions are performed by the agents in $\{1,\dots,n\}\setminus\Gamma$.

The remaining **ATL**-style operators $\langle\langle\Gamma\rangle\rangle G$ and $\langle\langle\Gamma\rangle\rangle U$ are verified using the appropriate modification of the procedures in Figures 2.6 and 2.7, similarly to the verification procedures employed in [Alur et al., 2002] and implemented in the model checker MOCHA (see Section 2.2.3.2).

As noted in Section 2.1.7.3 and in Figure 2.3, the interpretation of **ATL** in a non-

```
MC-UNIF($DIS, \Gamma, \varphi$) {
  Compute the set $\{DIS\}_\Gamma$
  for each ($X \in \{DIS\}_\Gamma$) {
    if ( MC$_{\mathbf{CTLK}}(\varphi, X) == G$ ) return true;
  }
  return false;
}
```

Figure 3.11: Model checking procedure for Γ-uniform deontic interpreted systems.

deterministic deontic interpreted system may not correspond to the original spirit of **ATL** in concurrent game structures. Indeed, the procedure presented above verifies what agents in a group Γ *may* bring about, perhaps by guessing moves when in epistemically equivalent states. While this is appropriate in certain circumstances, in other cases it is necessary to express what agents can enforce (see the examples in Chapter 6 and the example in Figure 2.3). Γ-uniform interpreted systems, defined on page 35, provide a semantics closer to the original meaning of **ATL** operators. Given a set of agents $\Gamma \subseteq \{1, \ldots, n\}$, let $\{DIS\}_\Gamma$ be the set of Γ-uniform deontic interpreted systems compatible with *DIS*. The set $\{DIS\}_\Gamma$ can be computed by taking all the possible restrictions of the protocols for agents in Γ; as noted in Section 2.1.7.3, there are at most $\prod_{i \in \Gamma} |Act_i|^{|L_i|}$ such protocols, and thus $\{DIS\}_\Gamma$ contains at most $\prod_{i \in \Gamma} |Act_i|^{|L_i|}$ elements. Figure 3.11 presents a procedure to verify whether or not $DIS \models_\Gamma \varphi$. The procedure MC-UNIF($DIS, \Gamma, \varphi$) implements the idea presented in Section 2.1.7.3: a formula φ is true in a class of Γ-uniform deontic interpreted systems iff it is true in at least one Γ-uniform interpreted system compatible with *DIS*. Notice that the MC-UNIF makes calls to the procedure MC$_{\mathbf{CTLK}}$ passing a deontic interpreted system $X \in \{DIS\}_\Gamma$ as a parameter.

As in the case of the previous procedures, all the operations appearing in the verification of **ATL**-style operators in deontic interpreted systems (both non-deterministic and Γ-uniform) can be expressed as operations on the Boolean representation of the parameters which, in turn, can be expressed as OBDDs.

3.3.6 Discussion

The procedures appearing in Figures 3.2–3.10 can be combined in the algorithm MC$_{\mathbf{CTLK}}$ presented in Figure 3.12, which constitutes an algorithm for the verification of $\mathbf{CTLKD} - \mathbf{A}^{D,C}$ formulae in deontic interpreted systems. Optionally, the algorithm MC$_{\mathbf{CTLK}}$ can be called by the procedure presented in Figure 3.11, thereby enabling the verification of the full language of $\mathbf{CTLKD} - \mathbf{A}^{D,C}$ in Γ-uniform interpreted systems.

This algorithm includes a number of Boolean parameters (such as the encodings for actions,

```
$\textsc{mc}_{\mathbf{CTLK}}(\varphi, DIS)$ {
  $\varphi$ is an atomic formula: return $V(\varphi)$;
  $\varphi$ is $\neg\varphi_1$: return $S \setminus \textsc{mc}_{\mathbf{CTLK}}(\varphi_1, DIS)$;
  $\varphi$ is $\varphi_1 \vee \varphi_2$: return $\textsc{mc}_{\mathbf{CTLK}}(\varphi_1, DIS) \cup \textsc{mc}_{\mathbf{CTLK}}(\varphi_2, DIS)$;
  $\varphi$ is $EX\varphi_1$: return $\textsc{mc}_{\mathbf{CTLK},EX}(\varphi_1, DIS)$;
  $\varphi$ is $EG\varphi_1$: return $\textsc{mc}_{\mathbf{CTLK},EG}(\varphi_1, DIS)$;
  $\varphi$ is $E[\varphi_1 U \varphi_2]$: return $\textsc{mc}_{\mathbf{CTLK},EU}(\varphi_1, \varphi_2, DIS)$;
  $\varphi$ is $K_i\varphi$: return $\textsc{mc}_{\mathbf{CTLK},K}(\varphi, i, DIS)$
  $\varphi$ is $E_\Gamma\varphi$: return $\textsc{mc}_{\mathbf{CTLK},E}(\varphi, \Gamma, DIS)$
  $\varphi$ is $D_\Gamma\varphi$: return $\textsc{mc}_{\mathbf{CTLK},D}(\varphi, \Gamma, DIS)$
  $\varphi$ is $C_\Gamma\varphi$: return $\textsc{mc}_{\mathbf{CTLK},C}(\varphi, \Gamma, DIS)$
  $\varphi$ is $O_i\varphi$: return $\textsc{mc}_{\mathbf{CTLK},O}(\varphi, i, DIS)$
  $\varphi$ is $\hat{K}_i^\Gamma \varphi$: return $\textsc{mc}_{\mathbf{CTLK},KH}(\varphi, i, \Gamma, DIS)$
  $\varphi$ is $\langle\langle\Gamma\rangle\rangle X\varphi$: return $\textsc{mc}_{\mathbf{CTLK},\langle\langle\Gamma\rangle\rangle X}(\varphi, \Gamma, DIS)$
  $\varphi$ is $\langle\langle\Gamma\rangle\rangle G\varphi$: return $\textsc{mc}_{\mathbf{CTLK},\langle\langle\Gamma\rangle\rangle G}(\varphi, \Gamma, DIS)$
  $\varphi$ is $\langle\langle\Gamma\rangle\rangle[\varphi_1 U \varphi_2]$: return $\textsc{mc}_{\mathbf{CTLK},\langle\langle\Gamma\rangle\rangle U}(\varphi_1, \varphi_2, \Gamma, DIS)$
}
```

Figure 3.12: The labelling algorithm for $\mathbf{CTLKD} - \mathbf{A}^{D,A}$.

protocols, epistemic and correct behaviour relations) that are not present in the traditional model checking algorithm for **CTL**. Moreover, the temporal relation R_t is not provided explicitly, but it must be derived from other parameters, and the set of reachable states G needs to be computed before performing the verification of formulae.

Correctness of the algorithm: The correctness of the model checking procedures for temporal-only operators derives from the correctness of the procedures employed in the verification of standard **CTL** models. To prove the correctness of the epistemic operators, notice that the operator τ appearing in Equation 3.2 is monotonic, which guarantees the existence of a fix-point G corresponding to the set of reachable states (this set is, by definition, the set of states reachable from the set of initial states via temporal transitions). The procedures in Figures 3.4–3.9 have the same structure than the temporal operator AX, and they employ the set G defined above. A similar argument applies to the procedures for **ATL** operators, which have the same structure used in MOCHA, but limited to the set of reachable global states.

The complexity of this procedure is analysed in the next chapter.

Chapter 4

The complexity of model checking multi-agent systems

This chapter presents complexity results for the problem of model checking multi-agent systems. Section 4.1 deals with the problem of model checking a $\mathbf{CTLKD} - \mathbf{A}^{D,C}$ formula in a model given "explicitly". Section 4.2 investigates the complexity of model checking in models whose representation is given in a "compact" way. Theoretical preliminaries for the analysis of complexity were presented in Section 2.1.6, while results for temporal-only model checking appear in Section 2.2.5.

4.1 The complexity of "explicit" model checking

Given a model $M = (W, R_t, \sim_1, \ldots, \sim_n, R_1^O, \ldots, R_n^O, t, V)$ and a formula φ of the logic $\mathbf{CTLKD} - \mathbf{A}^{D,C}$, the aim of this section is to investigate the complexity of establishing whether or not $M \models \varphi$, as a function of $|M|$ and $|\varphi|$, under the assumption that states and relations are listed explicitly. The following theorem provides a upper bound for model checking a particular class of logics:

Theorem 4.1.1. *(From [Fagin et al., 1995], p.63) Consider a Kripke model $M = (W, R_1, \ldots, R_n, V)$ for a logic interpreted using Kripke semantics (e.g. $\mathbf{S5}^n$, $\mathbf{K}^n$, etc., see Section 2.1.2) and a formula φ of the same logic. There is an algorithm that, given a model M and a formula φ, determines in time $O(|M| \times |\varphi|)$ whether or not $M \models \varphi$.*

The time complexity for model checking fusion of logics (see Section 2.1.5.1) can be derived using the following theorem:

Theorem 4.1.2. *(From [Franceschet et al., 2004]) Let $M = (W, R_1, R_2, V)$ be a model for the fusion of two logics $\mathbf{L}_1$ and $\mathbf{L}_2$, and φ a formula of $\mathbf{L}_1 \otimes \mathbf{L}_2$. The complexity of* *model checking a formula φ of $\mathbf{L}_1 \otimes \mathbf{L}_2$:*

$$O(m_1 + m_2 + n \cdot n) + \sum_{i=1}^{2}((O(k) + O(n)) \cdot C_{\mathbf{L}_i}(m_i, n, k))$$

where $m_i = |R_i|$, $n = |W|$, $k = |\varphi|$, and $C_{\mathbf{L}_i}$ is the complexity of model checking for logic $\mathbf{L}_i$, as a function of m_i, n and k.

A **P**-time algorithm for model checking common knowledge in interpreted systems is provided in [Meyden, 1998], while it is shown in [Alur et al., 2002] that model checking **ATL** formulae is a **P**-complete problem.

The logic $\mathbf{CTLKD} - \mathbf{A}^{D,C}$ is the fusion of logics to reason about time (**CTL**), knowledge and common knowledge and distributed knowledge ($\mathbf{S5}^n$), strategies (**ATL**), and of a logic to reason about correct behaviour (using a "deontic" dimension). This observation, together with Theorems 4.1.1 and 4.1.2, and with the complexity result for **CTL** appearing in Table 2.7, implies the following:

Lemma 4.1.1. *Model checking for the logic $\mathbf{CTLKD} - \mathbf{A}^{D,C}$ is a **P**-complete problem.*

Proof. The problem is **P**-easy: each component of $\mathbf{CTLKD} - \mathbf{A}^{D,C}$ admits a polynomial model checking algorithm, therefore the model checking problem for the fusion of these logics is polynomial by Theorem 4.1.2. The problem is **P**-hard: model checking the temporal fragment of $\mathbf{CTLKD} - \mathbf{A}^{D,C}$ is a **P**-complete problem. □

4.2 The complexity of model checking compact representations

As noted in Section 2.2.5, in many applications models are not described explicitly by listing states and relations. Instead, a compact representation is given; for instance one of the languages presented in Section 2.2.3 may be used to describe temporal models. Under this perspective, deontic interpreted systems can be seen as a compact representations for $\mathbf{CTLKD} - \mathbf{A}^{D,C}$ models. Indeed, the size of the model M_{DIS} associated to a given deontic interpreted system *DIS* can be exponentially larger than *DIS* itself. For instance, if *DIS* is composed of three agents, each of which is characterised by two local states, the number of states of M_{DIS} is 2^3.

The aim of this section is to investigate the complexity of model checking a **CTLK** formula φ in a given deontic interpreted system *DIS*:

$$DIS = \left\langle (G_i, R_i, Act_i, P_i, t_i)_{i \in \{1,\dots,n\}}, I, V \right\rangle$$

in terms of $|\varphi|$ and $|DIS|$. This section shows that model checking **CTLK** formulae in concurrent programs (see Section 2.2.5) is a **PSPACE**-complete problem. This provides a result for the complexity of model checking deontic interpreted systems, as these can be reduced to concurrent programs (see below).

A concurrent program $D = \langle AP, AC, S, \Delta, s^0, L\rangle$ is obtained by the parallel composition of n programs $D_i = \langle AP_i, AC_i, S_i, \Delta_i, s_i^0, L_i\rangle$ (see page 55). Formulae of **CTLK** are interpreted in a concurrent program D in a standard way: temporal operators are interpreted using the transition relation Δ, and epistemic operators are interpreted using epistemic accessibility relations defined by the equivalence of the "local"components $s_i \in S_i$ of the "global" states $s \in S$, as in Section 2.1.7.1.

The following results will be used in the proof of Theorem 4.2.1, presented below. Lemma 4.2.1 states that, a formula $EG\varphi$ is true at a state s in a **CTL** model M iff φ is true on a sequence of states of length $|M|$ starting from s. Lemma 4.2.2 states that $E[\varphi U \psi]$ is true at a state s in a **CTL** model M iff there is a state s' on a path starting from s at a distance not greater than $|M|$ from s, in which $s' \models \psi$, and such that φ holds in all states from s to s'.

Lemma 4.2.1. *Given a Kripke model $M = (S, R, V)$ for* **CTL***, a state $s \in S$, and a formula φ, $M, s \models EG\varphi$ iff there exists a sequence of states π starting from s of length $|\pi| \geq |M|$ s.t. $M, \pi(i) \models \varphi$ for all $0 \leq i \leq |M|$.*

Proof. If $M, s \models EG\varphi$, then there exists a path π from s such that, for all $i \geq 0$, $M, \pi(i) \models \varphi$; as the relation R is serial, this path is infinite (so, obviously, $|\pi| \geq |M|$).

Conversely, if there is a path π from s of length $|\pi| \geq |M|$, then such a path must necessarily include a backward loop. As $M, \pi(i) \models \varphi$ for all i in this loop, it suffices to consider the (infinite) trace generated by this loop to obtain a (semantic) witness for $M, s \models EG\varphi$. □

Lemma 4.2.2. *Given a Kripke model $M = (S, R, V, I)$ for* **CTL***, a state $s \in S$, and two formulae φ and ψ, $M, s \models E[\varphi U \psi]$ iff there exists a sequence of states π starting from s s.t. $M, \pi(i) \models \psi$ for some $i \leq |M|$, and $M, \pi(j) \models \varphi$ for all $0 \leq j < i$.*

Proof. If $M, s \models E[\varphi U \psi]$, by the definition of the until operator, there must exist a state s' in which ψ holds, and φ holds in every state from s to s'. Moreover, the state s' cannot be at a "distance" greater than $|M|$ from s. The other direction is a sufficient condition for $M, s \models E[\varphi U \psi]$. □

Theorem 4.2.1. *Model checking* **CTLK** *specifications in concurrent programs is a PSPACE-complete problem.*

Proof. Proof idea: given a formula φ of **CTLK** and n programs D_i defining a concurrent program D, a deterministic, polynomially-space bounded Turing machine T is defined that

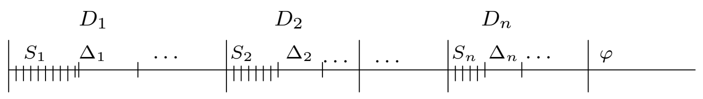

Figure 4.1: Input tape for the Turing machine T.

```
main {
  state = $(1, 1, \dots, 1)$;
  repeat
    if REACHABLE(state) then
      if SATISFIABLE($\neg\varphi$,state) then
        return *yes*;
      else
        move to next state;
      end if
    else
      move to next state;
    end if;
  until last state;
  return *no*;
}
```

Figure 4.2: The main loop for the Turing machine T.

halts in an accepting state iff $\neg\varphi$ is satisfiable in D (i.e., iff there exists a state $s \in S$ s.t. $D, s \models \neg\varphi$). Based on this, it is possible to conclude that the problem of model checking is in **co-PSPACE**. As deterministic complexity classes are closed under complement (Theorem 2.1.3), this implies that the problem is **PSPACE**-complete (the lower bound being given by the complexity of model checking **CTL** in concurrent programs).

Proof details: T is a multi-string Turing machine whose inputs are the n programs D_i and the formula φ. T operates "inductively" on the structure of the formula φ (see also [Cheng, 1995] for similar approaches), by calling other machines ("sub-machines") dealing with a particular logical operator only. The input of T includes the states of the program S_i ($1 \leq i \leq n$), the transition relations, the evaluation functions and all the other input parameters of each D_i. This information can be stored on a single input tape, separated by appropriate delimiters, together with the formula φ. The input tape is depicted informally in Figure 4.1.

T returns *yes* iff there exists a state $s \in S$ such that $D, s \models \neg\varphi$. The machine T iterates over the set of states $s \in S$ and checks whether $\neg\varphi$ holds in one of these or not. If a state is found such that $D, s \models \neg\varphi$, then the machine halts in a *yes* state; if the machine loops over all the states without finding a state satisfying $\neg\varphi$, then T halts in a *no* state. A high-level description of the main loop of the machine is given in Figure 4.2:

In the program above, `state` is a tuple of n values $(s_1, \dots, s_n)$, such that, for all i, $s_i \in S_i$. Notice that, since n is finite, and since $|S|$ is finite, states can be numbered; for instance, the state $(1, 1, \dots, 1)$ is the state obtained by taking the first element from S_1 in D_1, the first element from S_2 in D_2, and so on, following the order depicted in Figure 4.1. The procedure REACHABLE(`state`) verifies that s is reachable from the initial state. The algorithm of Theorem 2.1.1 can be used here, but notice that a polynomial amount of space is needed to store `state` (as it is the product of states of D_i); the algorithm uses the transition relations Δ_i encoded in the input tape to verify whether two states are connected or not.

The procedure SATISFIABLE(φ,`state`) returns *yes* if the formula is satisfiable at `state`, and it returns *no* otherwise. The procedure SATISFIABLE operates recursively on the structure of the formula by calling one of the machines described below. Each machine accepts a state s and a formula, and returns either *no* (the formula is false at s) or *yes* (the formula is true at s). Notice that each machine can call any of the other machines. The following is a description of the formula-specific machines that may be called by SATISFIABLE:

- The machine T_p for atomic formulae simply checks whether or not `state` is in $L($`state`$)$, where the evaluation L is obtained from the evaluations for each program in the input string; if the proposition is true at `state`, then T_p returns *yes*, otherwise it returns *no*.
- The machine $T_\neg$ for formulae of the form $\psi = \neg\varphi$ calls the appropriate machine for φ and returns the opposite.
- The machine $T_\vee$ for the disjunction $\psi = \psi' \vee \psi''$ first calls the machine for ψ'. If the result is *yes*, it outputs *yes*, otherwise it returns the output of the machine for ψ''.
- The machine T_{EX} for formulae of the form $\psi = EX(\varphi)$ implements the program of Figure 4.3. The machine T_{EX} accepts a formula φ and a state as input; similarly to the main loop, the machine iterates over the set of states using the variable `state2`. For each state, the machine checks whether `state2` is reachable from the input `state`; if this is the case, then T_{EX} checks whether or not φ is satisfied at `state2`. If T_{EX} finds such a state, then it halts in a *yes* state; otherwise, if no state can be found, T_{EX} terminates in a *no* state. Notice that this machine uses a polynomial amount of space (the space required to store `state2`).
- The machine T_{EU} for formulae of the form $E[\varphi U \psi]$ implements the program of Figure 4.4. The machine T_{EU} accepts two formulae and a state. The machine operates by iterating over the set of states using the counter `state2`, similarly to the previous machines. In each loop, the machine checks whether ψ holds in `state2` and whether `state` and `state2` are the same state. If this is the case, then the machine halts in a *yes* state. Otherwise, the machine checks whether or not there

```
$T_{EX}$($\varphi$,state) {
  state2 = $(1, 1, \ldots, 1)$;
  repeat
    if REACHABLE(state,state2) then
      if SATISFIABLE($\varphi$,state2) then
        return *yes*;
      else
        move to next state2;
      end if
    else
      move to next state2;
    end if;
  until last state2;
  return *no*;
}
```

Figure 4.3: The machine T_{EX}.

is a sequence of states from `state` to `state2` such that φ holds along the sequence. This check is performed by the procedure PATH(`state`,`state2`,φ,N), which returns *yes* if there is such a path, of length at most 2^N. By Lemma 4.2.2, it is sufficient to take N to be the logarithm of the size of the model which, in turn, can be approximated by $log(Max\{|S_i|\}^n) \leq n \cdot |\{D_i\}_{i\in\{1,\ldots,n\}}|$, where $|\{D_i\}_{i\in\{1,\ldots,n\}}|$ is the sum of the sizes of the programs, i.e., $|\{D_i\}_{i\in\{1,\ldots,n\}}| = \sum\limits_{i\in\{1,\ldots,n\}} |D_i|$ (notice that this value can be computed only once, at the beginning of the run). A recursive algorithm to solve PATH is presented in [Papadimitriou, 1994]; this algorithm employs at most space proportional to N, and it can be extended by adding a simple check for the satisfiability of φ. As there can be at most $|\varphi|$ checks, PATH uses at most $O(n \cdot |\{D_i\}_{i\in\{1,\ldots,n\}}| \cdot |\varphi|)$ space (i.e., it operates in **PSPACE**).

- The machine T_{EG} for formulae of the form $EG(\varphi)$ is defined by taking the deterministic version of the non-deterministic Turing machine NT_{EG} depicted in Figure 4.5. Based on Lemma 4.2.1, this machine guesses a sequence of states of length greater than $|\{D_i\}_{i\in\{1,\ldots,n\}}|$ (as above, this value can be computed at the beginning of the run by evaluating the size of the input) in which φ holds. If (and when) such a sequence is found, the machine returns *yes* (notice that this machine uses a polynomial amount of space and always halts). By Theorem 2.1.2, it is possible to build a deterministic machine T_{EG} in **PSPACE** that returns *yes* iff there exists a sequence of states of length greater than $|\{D_i\}_{i\in\{1,\ldots,n\}}|$ in which φ holds.

- The machine T_K for formulae of the form $\psi = K_i(\varphi)$ is depicted in Figure 4.6. The machine T_K accepts a formula φ, an index i, and a state as input; similarly to T_{EX}, the machine iterates over the set of states using the variable `state2`. For each state,

```
$T_{EU}$($\varphi$, $\psi$,state) {
  state2 = $(1, 1, \ldots, 1)$;
  repeat
    if SATISFIABLE($\psi$,state2) then
      if ( state == state2) return $yes$;
      else
        if ( PATH(state,state2,$\varphi$,$N$) ) then
          return $yes$;
        else
          move to next state2;
        end if;
      end if;
    else
      move to next state2;
    end if
  until last state2;
  return $no$;
}
```

Figure 4.4: The machine T_{EU}.

```
$NT_{EG}$($\varphi$,state) {
  state2 = state;
  counter = 0;
  repeat
    guess a state2;
    if (state is connected with state2) then
      if ( SATISFIABLE($\varphi$,state2) then
        counter = counter + 1;
      else
        return $no$;
      end if
    else
      return $no$;
    end if
  until counter > $|\{D_i\}_{i \in \{1,\ldots,n\}}|$
  return $yes$;
}
```

Figure 4.5: The machine T_{EG}.

```
$T_K(\varphi, i,$state$)$ {
  state2 $= (1, 1, \ldots, 1)$;
  repeat
    if REACHABLE-K(state,state2,$i$)
          and REACHABLE(state2) then
      if SATISFIABLE($\neg\varphi$,state2) then
        return $no$;
      else
        move to next state2;
      end if
    else
      move to next state2;
    end if;
  until last state2;
  return $yes$;
}
```

Figure 4.6: The machine T_K.

the machine checks whether `state2` is reachable from the set of initial states with the procedure REACHABLE(`state2`), and checks whether `state2` is reachable from `state` by means of epistemic accessibility relation i. This last check is performed by the procedure REACHABLE-K, comparing the i-th component of `state` and `state2`; if this is the case, then T_K checks whether or not $\neg\varphi$ is satisfied at `state2`. If T_K finds a state satisfying the conditions above, then it halts in a *no* state (because this state violates the definition of satisfiability for K_i); otherwise, if no state can be found, T_K terminates in a *yes* state. Notice that this machine uses a polynomial amount of space (i.e., the space required to store the value of the counter for `state2`).

Each of the machines above uses at most a polynomial amount of space, and there are at most $|\varphi|$ calls to these machines in each run of T. Thus, T uses a polynomial amount of space.

□

The **PSPACE** complexity result obtained above can be applied with minor modifications to the analysis of the complexity for model checking deontic interpreted systems (and to other formalisms, such as network of automata, see [Lomuscio and Raimondi, 2006a]). A deontic interpreted system can be reduced to a concurrent program: each agent can be associated with a program $D_i = \langle AP_i, AC_i, S_i, \Delta_i, s_i^0, L_i \rangle$, where AC_i is the set of actions for agent i, S_i is the set of local states for agent i, and the evolution function Δ_i is the temporal evolution function t_i for the agent. In the formalism of deontic interpreted systems an agent's evolution function may depend on other agents' actions. The definition of Δ_i can be modified to take in account this requirement. Similarly, the definition of the "global" evolution function t in a deontic interpreted system depends on all agents' actions. The definition of a concurrent program can be modified as follows:

- $AP = \cup_{1 \leq i \leq n} AP_i$;
- $AC = \prod_{1 \leq i \leq n} AC_i$;
- $S = \prod_{1 \leq i \leq n} S_i$;
- $(s, a, s') \in \Delta$ iff $\forall 1 \leq i \leq n$, $(s[i], a, s'[i]) \in \Delta_i$;
- $L(s) = \cup_i L_i(s[i])$.

Notice that, instead of taking the union, AC is now the *Cartesian* product of the agents' actions AC_i, and the transition function Δ is modified accordingly; this modification only impacts the procedure REACHABLE, but it does not affect complexity results. Thus, given a deontic interpreted system and a **CTLK** formula φ, it is possible to obtain a concurrent program D of size equal to the original description using deontic interpreted systems, so that the Turing machine T defined above can be employed to perform model checking of φ. By noting that each concurrent program can be reduced to a deontic interpreted system as well, it is possible to conclude that the problem of model checking **CTLK** formulae in deontic interpreted systems is a **PSPACE**-complete problem.

4.3 Discussion

The proof presented in the previous section differs from the proof of **PSPACE**-completeness for model checking **CTL** specifications in concurrent programs developed in [Kupferman et al., 2000] and presented in Section 2.2.5, in that it does not use results from automata theory. In this sense, the proof of Theorem 4.2.1 provides an alternative proof of the upper bounds for model checking **CTL** formulae in concurrent programs, which can be easily extended to **CTLK**.

Recently, [Hoek et al., 2006] presented complexity results for model checking **ATL** formulae when models are represented in a compact way, showing that this is an **EXP**-complete problem.

It is worth noticing that the verification algorithms presented in Chapter 3 require, in the worst case, space (and time) exponential in the size of the input to perform verification. Indeed, it is known that certain problems [Bryant, 1991] require OBDDs of size exponential in the number of variables. Nevertheless, experimental results show that the average time and space requirements for these algorithms are well below the worst case scenario for most of the examples (see Section 6.5).

A discussion on the complexity of model checking tools for multi-agent systems can be found in Section 6.5.5.

Chapter 5

MCMAS

This chapter describes MCMAS, a Model Checker for Multi-Agent Systems. MCMAS is developed in C/C++, and it should be considered a prototype model checker for multi-agent systems to assess whether the algorithms presented in Chapter 3 may be applied efficiently to large examples.

Section 5.1 presents an overview of the implementation. The syntax of the input language of MCMAS is given in Section 5.2, while Section 5.3 provides some implementation details. Compilation instructions and examples of MCMAS usage are presented in Section 5.4.

5.1 Overview

MCMAS is a model checker for the verification of $\mathbf{CTLKD} - \mathbf{A}^{D,C}$ formulae in deontic interpreted systems, with the possibility of performing verification on the class of Γ-uniform deontic interpreted systems compatible with a given deontic interpreted systems. Deontic interpreted systems are described in MCMAS using the language ISPL (see Section 5.2). MCMAS is released under the terms of the GNU General Public License (GPL); the source code, examples, and installation notes are available from [Raimondi and Lomuscio, 2006].

The structure of MCMAS is depicted in Figure 5.1. The steps 1 to 14, inside the box, are performed automatically upon invocation of the tool.

- In step 1 and 2, the input ISPL file is parsed using standard tools (Lex and Yacc), which define the grammar of ISPL. The input file includes both the description of a deontic interpreted system and a list of formulae to be verified. At this stage MCMAS builds an internal representation of the parameters appearing in the input file (such as agents' names, local states, actions, etc.) using data structures such as lists and maps: these are manipulated using the C++ Standard Template Library (STL).

Deontic interpreted systems description
(text file using ISPL syntax)

```
1   Parse input;
2   Parse the formulae to check
3   if ( $\Gamma$-uniform required ) {
4       compute the set of deontic interpreted
        systems compatible with the ISPL input;
5       for each compatible deontic interpreted system {
6           Build OBDDs for parameters;
7           Compute the set of reachable states;
8           Compute the set of states in which formula holds;
9           if ( reachable states = [[ $\phi$ ]] ) return TRUE;
        }
    } else {
10      Build OBDDs for parameters
11      Compute the set of reachable states;
12      Compute the set of states in which formula holds;
13      if ( reachable states = [[ $\phi$ ]] ) return TRUE;
    }
14  return FALSE;
```

Figure 5.1: High level description of MCMAS.

- In step 3 MCMAS checks whether the parameter to perform verification in Γ-uniform interpreted systems was provided. If this is the case, MCMAS proceeds to step 4; otherwise, it proceeds to step 10. Notice: MCMAS determines the set Γ for each formula that has to be verified by taking all the agents appearing inside any **ATL** operator. Other implementation choices are possible, for instance a list of "uniform" agents could be provided and read from the command line.

- MCMAS performs step 4 only if verification in Γ-uniform interpreted systems is required. In this case, the number of deontic interpreted systems compatible with the ISPL input is computed. As described in Section 2.1.7.3, the number of elements of $\{DIS\}_\Gamma$ is at most $\prod_{i\in\Gamma} |Act_i|^{|L_i|}$. For instance, if the set of local states for an agent x includes a local state l_x and a local state l'_x, such that $P_x(l_x) = \{a_x, a'_x\}$ and $P_x(l'_x) = \{a_x, a'_x\}$ (i.e., two actions are allowed in each local state), then $\{DIS\}_\Gamma$ contains at least four elements.

- In step 6 the OBDDs corresponding to the Boolean encodings are built for each of the Γ-uniform interpreted systems compatible with the given ISPL input. First, the number of Boolean variables needed to encode global states and joint actions is computed; then, the OBDDs representing protocols, evolution functions, set of initial states, etc., are built following the procedure of Section 3.3.1, starting from the internal representation using data structures obtained in step 2. Step 5 employs the CUDD C library [Somenzi, 2005] for the manipulation of OBDDs. Notice that only one element of $\{DIS\}_\Gamma$ is analysed in each loop, and the loop may terminate if the condition in step 9 holds.

- The Boolean representation (using an OBDD) of the set G of reachable states is built in step 7 by computing the fix-point of Equation 3.2.

- The set of states in which a formula holds is computed in step 8, for each formula appearing in the input ISPL file. As above, these sets are computed as Boolean formulae using the procedures appearing in Sections 3.3.2 – 3.3.5, and they are manipulated using their encoding with OBDDs. In step 9, this OBDD is compared to the OBDD encoding the set of reachable states. If the two OBDDs are equal the loop terminates, as a single positive witness is enough to prove $DIS \models_\Gamma \varphi$. Otherwise, the `for` loop continues with another Γ-uniform interpreted system. If no Γ-uniform can be found for a particular formula, MCMAS returns FALSE (step 14).

- The steps from 10 to 13 are executed when verification in Γ-uniform interpreted systems is not required. These steps are similar to steps from 6 to 9, the only difference being the encoding of the parameters. In this case, agents are not required to be uniform and all the protocols may be non-deterministic.

MCMAS is run from the command line and it accepts various options, in addition to the parameter to enable verification in Γ-uniform interpreted systems. The additional parameters include options to modify its verbosity, to inspect OBDDs statistics and memory usage, to enable variable reordering; these can be used to determine the critical points in the verification of large examples, and to fine tune the performance of the tool.

MCMAS is written in C/C++ and it can be compiled on various platforms, including PowerPC (Mac OS X 10.2 to 10.4), Mac Intel (Mac OS X 10.4), Intel (various Pentium versions using Linux 2.4 and 2.6, and using Windows with Cygwin), and SPARC (SunOS 5.8 and 5.9). The source code has been compiled with gcc/g++ from version 2.95 till version 3.4. The current version of MCMAS is version 0.7.

5.2 The language ISPL

ISPL (Interpreted Systems Programming Language) is the input language of MCMAS. An ISPL program describes a deontic interpreted system following closely the formalism presented in Sections 2.1.7.1 and 2.1.7.2.

5.2.1 General structure of an ISPL program

An ISPL program is composed of five sections:

1. Agents' declarations. In this section of ISPL agents are defined using a sequence of declarations, each of which has the following syntax:

   ```
   Agent <agentID>
     <agent_body>
   end Agent
   ```

   where `<agentID>` is a valid ISPL identifier (see below), and `<agent_body>` contains the declaration of local states, actions, protocol, and evolution function for each agent.

2. Evaluation function. In this section the evaluation function $V : AP \rightarrow 2^G$ (see Section 2.1.7.1) is defined as follows:

   ```
   Evaluation
     <proposition_declaration>
   end Evaluation
   ```

   `<proposition_declaration>` is a sequence of lines of the form `<proposition> if <condition_on_states>`, where `<proposition>` is a valid ISPL identifier and `<condition_on_states>` is a Boolean formula defining a set of global states.

3. Initial states. In this section the set of initial states is defined by the syntax:

   ```
   InitStates
     <condition_on_states>
   end InitStates
   ```

   where, as above, `<condition_on_states>` is Boolean formula defining a set of global states.

4. Groups declaration. In this sections the groups of agents used in the formulae to be verified are declared between the keywords (see Section 5.2.2) `Groups <groups_declaration> end Groups`.

5. List of formulae to verify. In this section, identified by the keywords `Formulae <formulae_list> end Formulae`, $\mathbf{CTLKD} - \mathbf{A}^{D,C}$ formulae to be verified are listed. The propositional formulae and the groups appearing in these formulae have to be declared, respectively, in items 2 and 4 above.

Comments may be included in ISPL programs using double dashes at the beginning of a line, for instance:

```
-- This is a comment
```

5.2.2 Formal syntax of ISPL

The basic element of an ISPL program is an identifier. An identifier is any string starting with a letter and followed by any number of letters, digits, or underscore sign. Formally,

```
  ID :: [a-zA-Z][a-zA-Z0-9_]*
```

The reserved keywords of ISPL are listed in Figure 5.2.

As described in the previous Section, an ISPL program is composed of five sections:

```
<Agents_list> <Evaluation> <InitialStates> <Groups> <Formulae>
```

- `<Agent_list>` is a sequence of `<Agent>` declarations of the following form:

  ```
  <Agent> :: "Agent" ID <Agent_body> "end Agent"
  ```

  where

"Agent"	
"Lstate"	(local states)
"Lgreen"	(green local states)
"Action"	
"Protocol"	
"Ev"	(evolution function)
"Evaluation"	
"InitStates"	
"Groups"	
"Formulae"	
"end"	
"if"	
"and"	
"or"	
"->"	(implication)
"AG"	(temporal operator AG)
"EG"	(temporal operator EG)
"AX"	(temporal operator AX)
"EX"	(temporal operator EX)
"X"	(operator X for strategic modalities)
"F"	(operator F for strategic modalities)
"G"	(operator G for strategic modalities)
"AF"	(temporal operator AG)
"EF"	(temporal operator AG)
"A"	(universal path quantifier)
"E"	(existential path quantifier)
"U"	(temporal operator Until)
"K"	(epistemic operator)
"GK"	(epistemic operator for everybody knows)
"GCK"	(epistemic operator for common knowledge)
"O"	(operator for correct behaviour)
"KH"	(operator for epistemic+deontic modality)
"DK"	(epistemic operator for distrib. knowledge)

Figure 5.2: ISPL reserved keywords.

```
<Agent_body> ::
                "Lstate = {" ID "," ID ... "};"
                "Lgreen = {" ID "," ID ... "};"
                "Action = {" ID "," ID ... "};"
                <Protocol>
                <Evolution>
```

Protocols are defined as follows:

```
<Protocol> ::
             "Protocol:"
                 ID ":" "{" ID "," ID... "};"
                 ID ":" "{" ID "," ID... "};"
                 ...
             "end Protocol"
```

The declaration of a protocol must include a line for each local state appearing in `Lstate`. Each line of the protocol starts with an identifier for the local state and it is followed by a list of identifiers for actions in curly braces.

The evolution function of an agent is defined as follows:

```
<Evolution> ::
              "Ev:"
                 ID "if" <Bool_ev_cond> ";"
                 ID "if" <Bool_ev_cond> ";"
                 ...
             "end Ev"
```

where `<Bool_ev_cond>` is a Boolean condition on local states and actions, defined by:

```
<Bool_ev_cond> ::
                 <Bool_ev_cond> "or" <Bool_ev_cond>
                 | <Bool_ev_cond> "and" <Bool_ev_cond>
                 | "not" <Bool_ev_cond>
                 | "Lstate = " ID
                 | "Action = " ID
                 | ID ".Lstate = " ID
                 | ID ".Action = " ID
```

Notice that the last two lines in the definition of `<Bool_ev_cond>` permit the reference to other agents' local states and actions, using an ID (an agent's name) followed by a dot to disambiguate references (see Section 2.1.7.1 for a discussion on the evolution functions). An evolution line of the form `ID "if" <Bool_ev_cond>` is read as: the next local state is `ID` if `<Bool_ev_cond>` is true. It is assumed that, if none of the `<Bool_ev_cond>` holds, then *the agent does not change its local state.* This assumption is key to keep the description of examples compact, because only the conditions causing a change need to be listed.

- The `<Evaluation>` section is defined as follows:

```
<Evaluation> ::
              "Evaluation"
                 ID "if" <Eval_bool_cond> ";"
                 ID "if" <Eval_bool_cond> ";"
                 ...
              "end Evaluation"
```

  Each line in the evaluation function defines a new atomic proposition of AP, identified by `ID`. `<Eval_bool_cond>` is a Boolean condition on the local states of agents, defined by:

```
<Eval_bool_cond> ::
                 <Eval_bool_cond> "or" <Eval_bool_cond>
                 | <Eval_bool_cond> "and" <Eval_bool_cond>
                 | "not" <Eval_bool_cond>
                 | ID ".Lstate = " ID
```

- The section declaring initial states is defined as follows:

```
<InitialStates> ::
                 "InitStates"
                    <Eval_bool_cond>
                 "end InitStates"
```

  where `<Eval_bool_cond>` is a Boolean condition on local states defined as above. The set of global states for which `<Eval_bool_cond>` is true corresponds to the set of initial states I.

- Groups of agents are declared in the section `<Groups>` as follows:

```
<Groups> ::
             "Groups"
                ID "=" "{" ID "," ID... "};"
                ID "=" "{" ID "," ID... "};"
                ...
             "end Groups"
```

Each line in the `Groups` section starts with an identifier (the group name), followed by a list of agents' names in curly braces, defining the members of the group.

- **CTLKD** – $\mathbf{A}^{D,C}$ formulae to be verified are declared in the section `<Formulae>`, defined by:

```
<Formulae> ::
             "Formulae"
                 <formula> ";"
                 <formula> ";"
                 ...
             "end Formulae"
```

The syntax of formulae is as follows:

```
<formula> ::
            <formula> "and" <formula>
            | <formula> "or" <formula>
            | <formula> "->" <formula>
            | "not" <formula>
            | "AG" <formula>
            | "EG" <formula>
            | "AX" <formula>
            | "EX" <formula>
            | "AF" <formula>
            | "EF" <formula>
            | "A[" <formula> "U" <formula> "]"
            | "E[" <formula> "U" <formula> "]"
            | "K(" ID "," <formula> ")"
            | "GK(" ID "," <formula> ")"
            | "GCK(" ID "," <formula> ")"
            | "DK(" ID "," <formula> ")"
            | "O(" ID "," <formula> ")"
            | "KH(" ID "," ID "," <formula> ")"
            | "<<" ID ">>X" <formula>
            | "<<" ID ">>F" <formula>
            | "<<" ID ">>G" <formula>
            | "<<" ID ">>[" <formula> "U" <formula> "]"
            | ID
```

A simple ISPL code with two agents to test ATL operators is depicted in Figure 5.3. More examples are described in Chapter 6.

```
Agent agt1
        Lstate = {l11,l12};
        Lgreen = {l11,l12};
        Action = {a11,a12};
        Protocol:
                l11: {a11,a12};
                l12: {a12};
        end Protocol

        Ev:
          l12 if (Lstate = l11 ) and ( ( Action=a12 )
            or ( Action = a11 and agt2.Action = a21) );
        end Ev
end Agent

Agent agt2
        Lstate = {l21,l22};
        Lgreen = {l21,l22};
        Action = {a21,a22};
        Protocol:
                l21: {a21,a22};
                l22: {a22};
        end Protocol

        Ev:
          l22 if (Lstate = l21) and   ( ( (Action=a21) and (agt1.Action=a11) )
            or ( (Action=a22) and (agt1.Action=a11) ) );
        end Ev
end Agent

Evaluation
        win if ( (agt1.Lstate=l12 and agt2.Lstate=l22) or
                 (agt1.Lstate=l11 and agt2.Lstate=l22) );
        init if ( agt1.Lstate = l11 and agt2.Lstate = l21 );
end Evaluation

InitStates
        ( agt1.Lstate = l11 and agt2.Lstate = l21 );
end InitStates

Groups
        g2 = {agt2};
        g1 = {agt1};
        g3 = {agt1,agt2};
end Groups

Formulae
        init -> <g1>X(win);
        init -> <g1>F(win);
        init -> <g2> F(win);
end Formulae
```

Figure 5.3: A simple ISPL code.

5.3 Implementation details

The source code of MCMAS has been structured in separated “modules”, with a certain number of shared parameters. The shared parameters include the internal representation of the ISPL input, and the OBDD variables and encodings for local states, actions, protocols, etc. The source code of MCMAS includes various subdirectories, each of which corresponds to a specific module.

- Directory `parser`: this directory includes the Lex file `nssis.l` defining the tokens of the grammar of ISPL, and the Yacc file `nssis.y` defining the formal parser for ISPL. The parser includes C code to build the internal representation of ISPL code as a tree, using the functions provided by the module `pnode`.

- Directory `pnode`: this directory includes C code and headers for the manipulation of tree data structures (this module is required by Lex, which is not compatible with C++ STL code). The building blocks of the parse tree are the nodes, defined in `pnode2.h`. All the Boolean conditions, the evaluation function, the definition of initial states, and the formulae to be verified are stored using `pnode` data structures.

- Directory `bdd`: this directory includes the C++ code and headers for the construction and the manipulation of OBDDs. The definition of OBDD's variables and the construction of the OBDDs for all the parameters is managed by the function `bddEncode()` in the file `bdd.cc`, which implements the procedures for Boolean encoding presented in Section 3.3.1. The verification of formulae is launched by the function `check_formula()` in the file `bdd.cc`, implementing the algorithm of Figure 3.12.

  In the case of verification for Γ-uniform interpreted systems, the same structure presented above is repeated in the file `bdduniform.cc`.

  OBDDs are manipulated in MCMAS using the CUDD library [Somenzi, 2005]. This is a C library offering a C++ interface to its functions. A simple example is reported in Figure 5.4, showing how the overloading of the operators `"+"`, `"!"`, `"*"`, `"="`, and `"=="` permits a simple manipulation of Boolean functions.

- Directory `examples`: this directory contains various ISPL examples. Some of these examples are described in detail in Chapter 6.

The root directory of the source tree of MCMAS includes the file `main.cc`. This file contains the function calls to parse input parameters, to perform setup operations at the beginning of a run, and to perform cleaning operations upon completion of verification. The root directory includes also a Makefile and installing instructions.

```
int main(int argc, char* argv[]) {

  Cudd bddmgr; // The OBDD manager
  bddmgr =  Cudd(0,0); // Initialisation of manager
  BDD x = bddmgr.bddVar(); // Declaration of a variable
  BDD y = bddmgr.bddVar();
  BDD f = x + y; // A formula
  BDD g = y + !x; // Another formula

  if ( f == g ) {  // Comparing two formulae
    cout << "f is equal to g";
  } else {
    cout << "f is NOT equal to g";
  }
}
```

Figure 5.4: Simple example of CUDD usage.

5.4 Usage

The compilation of MCMAS requires the following software:

- GNU C/C++ compiler.
- Lex and Yacc (or equivalent tools, such as Flex and Bison).
- CUDD library, compiled with its C++ interface.

Given the above, MCMAS usually compiles easily on most Linux platforms. The compilation under Mac OS X and Windows requires the installation of development tools, which are not part of standard distributions. Binary versions for these systems are available from [Raimondi and Lomuscio, 2006].

MCMAS is a command line tool, and it is run with the command `mcmas` from a system prompt. MCMAS needs at least one input parameter: the name of the ISPL file to parse.

Other input parameters may be provided. The following is an excerpt from the output of the command `mcmas -h`:

```
Usage: mcmas [OPTIONS] FILE
Example: mcmas -v 3 -bdd_stats myfile.ispl

Options:
```

```
$ ./mcmas examples/btp.ispl
*****************************************************************
                       mcmas v. 0.7
                          [...]
*****************************************************************
Encoding BDD parameters...Done.
Checking formulae...
  Formula number 0 is TRUE in the model
  Formula number 1 is TRUE in the model
done, 2 formulae successfully read and checked
```

Figure 5.5: Excerpts from MCMAS output.

```
-v Number      verbosity level ( 0 -- 5, default 0 )
-pnode_info    Print node manager info
-u             Perform verification on uniform interpreted systems
-bdd_stats     Print BDD statistics
-h             This screen
```

It is possible to trace the verification steps by increasing MCMAS verbosity. The option `-pnode_info` prints statistics about the internal memory usage (i.e., the number of nodes used). The option `-bdd_stats` launches a command of the CUDD library to print detailed OBDDs statistics, such as number of nodes, number of reorderings, total memory used, etc. The option `-u` is used to require verification in Γ-uniform interpreted systems. Figure 5.5 reports an excerpt of the output of MCMAS when verifying two formulae with verbosity 0.

Chapter 6

Applications

This chapter presents some examples using the formalism of deontic interpreted systems, their translation into ISPL code, and their verification using MCMAS. These examples belong to various domains: a communication and an anonymity protocol are presented, respectively, in Sections 6.1 and 6.2. Section 6.3 presents examples of strategic reasoning in multi-agent systems, while Section 6.4 introduces a characterisation of diagnosability using epistemic notions. Experimental results for all the examples are reported in Section 6.5.

6.1 The bit transmission problem (with faults)

In the bit-transmission problem [Fagin et al., 1995] a sender S wants to communicate the value of a bit to a receiver R, by using an unreliable communication channel (see Figure 6.1). In this example, the channel may drop messages, but cannot tamper messages; also, at any given time, the channel may transmit messages in one direction but not in the other.

One protocol to achieve communication is as follows: S immediately starts sending the bit to R, and continues to do so until it receives an acknowledgement from R. R does nothing until it receives the bit; from then on, it sends messages acknowledging the receipt to S. S stops sending the bit to R when it receives the first acknowledgement from R, and the protocol terminates here.

This scenario is extended in [Lomuscio and Sergot, 2004] to deal with failures. There are different kinds of faults that can be considered; here it is assumed that R may fail to behave as intended. Two examples are discussed in this section following [Lomuscio and Sergot, 2004]; in the first example, R may fail to send acknowledgements when it receives a message. In the second, R may send acknowledgements even if it has not received any

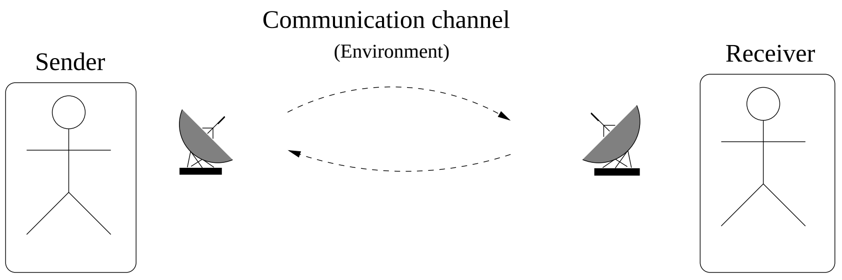

Figure 6.1: The bit transmission problem.

message.

It is possible to represent the scenario described above by means of the formalism of deontic interpreted systems, as presented in [Lomuscio and Sergot, 2004]. To this end, an agent E representing the environment is introduced to model the unreliable communication channel. The local states of the environment record the possible combinations of messages that have been sent in a round, either by S or R. Hence, four possible local states are taken for the environment:

$$L_E = \{(.,.), (sendbit, .), (., sendack), (sendbit, sendack)\},$$

where the first element in the tuple represents the action of S, the second element represents the action of R, and '.' represents a configuration in which no message has been sent by the corresponding agent.

The actions Act_E for the environment correspond to the transmission of messages between S and R on the unreliable communication channel. As mentioned above, it is assumed that the communication channel can transmit messages in both directions simultaneously, and that a message travelling in one direction can get through while a message travelling in the opposite direction is lost. Thus, the set of actions Act_E for the environment is taken as:

$$Act_E = \{S{-}R,\ S{\rightarrow},\ {\leftarrow}R,\ -\}.$$

The action "$S{-}R$" represents the action in which the channel transmits any message successfully in both directions. The action "$S{\rightarrow}$" represents a successful communication from S to R but unsuccessful from R to S. The action "${\leftarrow}R$" represents a successful communication from R to S but unsuccessful from S to R. Finally, the action "$-$" represents the environment stopping messages in either direction. We assume the following constant function for the protocol of the environment P_E:

$$P_E(l_E) = Act_E = \{S{-}R,\ S{\rightarrow},\ {\leftarrow}R,\ -\}, \quad \text{for all } l_E \in L_E.$$

The evolution function for E records simply the actions of Sender and Receiver (details

Final state	**Transition condition**
$(0, ack)$	($l_S = 0$ and $Act_R = sendack$ and $Act_E = S{-}R$) or ($l_S = 0$ and $Act_R = sendack$ and $Act_E = \leftarrow R$)
$(1, ack)$	($l_S = 1$ and $Act_R = sendack$ and $Act_E = S{-}R$) or ($l_S = 1$ and $Act_R = sendack$ and $Act_E = \leftarrow R$)

Table 6.1: Transition conditions for S.

can be found in the code reported in Figure 6.2).

The sender S is modelled by taking the following set consisting of four possible local states:

$$L_S = \{0, 1, (0, ack), (1, ack)\}.$$

They represent the value of the bit S is attempting to transmit, and whether or not S has received an acknowledgement from R.

The set of actions Act_S for S is taken as:

$$Act_S = \{sendbit(0), sendbit(1), \lambda\}.$$

They represent the action of sending a bit of value 0, the action of sending a bit of value 1, and the null action. The protocol for S is defined as follows:

$$P_S(0) = sendbit(0), \quad P_S(1) = sendbit(1),$$
$$P_S((0, ack)) = P_S((1, ack)) = \lambda.$$

Table 6.1 lists (in the right column) the conditions causing a transition for S to the local state appearing in the left column.

Faulty Receiver – 1: In this case it is assumed that R may fail to send acknowledgements when it is supposed to. To this end, the following local states are introduced for R:

$$L'_R = \{0, 1, \epsilon, (0, f), (1, f)\}.$$

The state ϵ is used to record the fact that in the run R has not received any message from S yet; 0 and 1 denote the value of the bit received. The local states (i, f) ($i = \{0, 1\}$) are *faulty* states denoting that, at some point in the past, R received a bit but failed to send an acknowledgement.

The set of actions for R is:

$$Act_R = \{sendack, \lambda\}.$$

The protocol for R is:

$$P'_R(\epsilon) = \lambda, P'_R(0) = P'_R(1) = \{sendack, \lambda\},$$
$$P'_R((0, f)) = P'_R((1, f)) = \{sendack, \lambda\}.$$

(notice: for a correct functioning Receiver it should be $P'_R(0) = P'_R(1) = \{sendack\}$).

The transition conditions for R are listed in Table 6.2.

Final state	**Transition condition**
0	($Act_S = sendbit(0)$ and $l_R = \epsilon$ and $Act_E = S{-}R$) or ($Act_S = sendbit(0)$ and $l_R = \epsilon$ and $Act_E = S{\rightarrow}$)
1	($Act_S = sendbit(1)$ and $l_R = \epsilon$ and $Act_E = S{-}R$) or ($Act_S = sendbit(1)$ and $l_R = \epsilon$ and $Act_E = S{\rightarrow}$)
$(0, f)$	$l_R = 0$ and $Act_R = \epsilon$
$(1, f)$	$l_R = 1$ and $Act_R = \epsilon$

Table 6.2: Transition conditions for R.

Faulty Receiver – 2: In this second case it is assumed that R may send acknowledgements without having received a bit first. This scenario is modelled with the following set of local states L''_R for R:

$$L''_R = \{0, 1, \epsilon, (0, f), (1, f), (\epsilon, f)\}.$$

The meaning of the local states $\epsilon, 0, 1, (0, f)$ and $(1, f)$ is as above; (ϵ, f) is a further *faulty* state corresponding to the fact that, at some point in the past, R sent an acknowledgement without having received a bit first. The set of actions is the same as in the previous example. The protocol is defined as follows:

$$P''_R(\epsilon) = \{sendack, \lambda\},$$
$$P''_R(0) = P''_R(1) = \{sendack\},$$
$$P''_R((0, f)) = P''_R((1, f)) = P''_R((\epsilon, f)) = \{sendack, \lambda\}.$$

The evolution function is a simple extension of Table 6.2.

For both examples, the following set of atomic propositions is introduced:

$$AP = \{\mathbf{bit = 0}, \mathbf{bit = 1}, \mathbf{recbit}, \mathbf{recack}\}.$$

Correspondingly, the following evaluation function is defined:

$$\begin{aligned}
V(\mathbf{bit = 0}) &= \{g \in G \mid \text{ either } l_S(g) = 0 \text{ or } l_S(g) = (0, ack)\};\\
V(\mathbf{bit = 1}) &= \{g \in G \mid \text{ either } l_S(g) = 1 \text{ or } l_S(g) = (1, ack)\};\\
V(\mathbf{recbit}) &= \{g \in G \mid \text{ either } l_R(g) = 1, \text{ or } l_R(g) = 0;\\
&\qquad \text{or } l_R(g) = (0, f) \text{ or } l_R(g) = (1, f)\};\\
V(\mathbf{recack}) &= \{g \in G \mid l_S(g) = (1, ack) \text{ or } l_S(g) = (0, ack)\}.
\end{aligned}$$

The parameters above describe two deontic interpreted systems, one for each faulty behaviour of R; in the following, these deontic interpreted systems are denoted by DIS_{BTP1} and DIS_{BTP2}.

Given the set AP above, various properties of DIS_{BTP1} and DIS_{BTP2} can be evaluated. For example, consider the following temporal and epistemic specifications:

$$\mathbf{recack} \implies (K_S(K_R(\mathbf{bit} = \mathbf{0}) \vee K_R(\mathbf{bit} = \mathbf{1}))); \tag{6.1}$$

$$\mathbf{recack} \implies (\hat{K}_S^R(K_R(\mathbf{bit} = \mathbf{0}) \vee K_R(\mathbf{bit} = \mathbf{1}))); \tag{6.2}$$

$$\neg EF(C_{S,R}((\mathbf{bit} = \mathbf{0})) \vee C_{S,R}((\mathbf{bit} = \mathbf{1}))). \tag{6.3}$$

Formula 6.1 captures the fact that it is always true that, upon receipt of an acknowledgement, S knows that R knows the value of the bit. Formula 6.2 expresses a similar concept, but by using knowledge under the assumption of correct behaviour. By encoding the examples in ISPL it is possible to verify in an automatic way that Formula 6.1 holds in DIS_{BTP1} but not in DIS_{BTP2}. This means that the faulty behaviour of R in DIS_{BTP1} does not affect the key property of the system. On the contrary, Formula 6.2 holds in both DIS_{BTP1} and DIS_{BTP2}; hence, a particular form of knowledge is retained, irrespective of the fault under consideration. Formula 6.3 expresses a general result about communication over a channel with a temporal delay: it is not possible to achieve common knowledge of a message [Fagin et al., 1995].

The ISPL code corresponding to DIS_{BTP1} is reported in Figure 6.2. The ISPL code corresponding to DIS_{BTP2} is available in the source tree of MCMAS. Experimental results for this example are discussed in Section 6.5.

6.2 The protocol of the dining cryptographers

The protocol of the dining cryptographers was introduced in [Chaum, 1988], and model checking of its properties was discussed in [van der Meyden and Su, 2004] (see also Section 2.3.1, page 60). The original wording from [Chaum, 1988] is as follows:

"*Three cryptographers are sitting down to dinner at their favourite three-star restaurant. Their waiter informs them that arrangements have been made with the maitre d'hotel for the bill to be paid anonymously. One of the cryptographers might be paying for the dinner, or it might have been NSA (U.S. National Security Agency). The three cryptographers respect each other's right to make an anonymous payment, but they wonder if NSA is paying. They resolve their uncertainty fairly by carrying out the following protocol:*

Each cryptographer flips an unbiased coin behind his menu, between him and the cryptographer on his right, so that only the two of them can see the outcome. Each cryptographer

```
Agent Sender
   Lstate = {s0,s1,s0ack,s1ack};
   Lgreen = {s0,s1,s0ack,s1ack};
   Action = {sb0,sb1,nothing};
   Protocol:
       s0: {sb0};
       s1: {sb1};
       s0ack: {nothing};
       s1ack: {nothing};
   end Protocol
   Ev:
     s0ack if (  ( (Lstate=s0) and (Receiver.Action=sendack) and
                   (Environment.Action=SR) )
           or ( (Lstate=s0) and (Receiver.Action=sendack) and
                (Environment.Action=R ) ) );
     s1ack if [... As above ...]
   end Ev
end Agent

Agent Receiver
   Lstate = {empty,r0,r1,r0f,r1f};
   Lgreen = {empty,r0,r1};
   Action = {nothing,sendack};
   Protocol:
       empty: {nothing};
       r0: {sendack,nothing};
       r1: {sendack,nothing};
       r0f: {sendack,nothing};
       r1f: {sendack,nothing};
   end Protocol

   Ev:
      r0 if ( ( (Sender.Action=sb0) and (Lstate=empty) and
                 (Environment.Action=SR) ) or
( (Sender.Action=sb0) and (Lstate=empty) and
               (Environment.Action=S) ) );
      r1 if [... as above ...]
      r0f if ( (Lstate = r0 ) and (Action=nothing) );
      r1f if ( (Lstate = r1 ) and (Action=nothing) );
   end Ev
end Agent

Agent Environment
     [... see text ...]
end Agent

Evaluation
        recbit if ( (Receiver.Lstate=r0) or (Receiver.Lstate=r1) or
     (Receiver.Lstate=r0f) or (Receiver.Lstate=r1f) );
        recack if ( (Sender.Lstate=s0ack) or (Sender.Lstate=s1ack) );
        bit0 if ( (Sender.Lstate=s0) or (Sender.Lstate=s0ack));
        bit1 if ( (Sender.Lstate=s1) or (Sender.Lstate=s1ack) );
end Evaluation

InitStates
  ( (Sender.Lstate=s0) or (Sender.Lstate=s1) ) and
  ( Receiver.Lstate=empty ) and ( Environment.Lstate=none );
end InitStates

Formulae
recack -> K(Sender,(K(Receiver,bit0) or K(Receiver,bit1)));
        recack -> KH(Sender,Receiver,K(Receiver,bit0) or K(Receiver,bit1));end Formulae
```

Figure 6.2: ISPL code for the bit transmission problem (excerpts).

then states aloud whether the two coins he can see – the one he flipped and the one his left-hand neighbour flipped – fell on the same side or on different sides. If one of the cryptographers is the payer, he states the opposite of what he sees. An odd number of differences uttered at the table indicates that a cryptographer is paying; an even number indicates that NSA is paying (assuming that the dinner was paid for only once). Yet if a cryptographer is paying, neither of the other two learns anything from the utterances about which cryptographer it is." [Chaum, 1988]

This protocol is the basic building block for the definition of more complex infrastructures enabling anonymous communication over public networks (see, for instance, [Goldschlag et al., 1999]). Notice that similar versions of the protocol can be defined for any number of cryptographers greater than three.

An instance of this example with three cryptographers is encoded in the formalism of deontic interpreted systems by introducing three agents C_i $(i = \{1, 2, 3\})$ to model the three cryptographers, and one agent E for the environment.

The environment is used to select non-deterministically the identity of the payer and the results of the coin tosses. A local state for the environment is a string of the form $< c_1c_2c_3p >$, where $c_i \in \{H, T\}, (i = \{1, 2, 3\})$ represents the result of the coin toss (Head or Tail) for coin i, and $p \in \{1, \dots, 4\}$ represents the payer ($p = 4$ means that the company paid for the dinner). This makes a total of 32 elements in the set L_E encoding the possible local states for the environment. It is assumed that the environment can perform only one action, the null action. Therefore, the protocol P_E is simply mapping every local state to the null action. Also, there is no evolution of the local states for the environment.

The local states of a cryptographer i $(i = \{1, 2, 3\})$ are modelled as a string $< c_ip_iu_i >$ representing, respectively, whether the coins that a cryptographer can see are equal ($c_i = E$) or different ($c_i = D$), whether the cryptographer is the payer ($p_i = Y$) or not ($p_i = N$), and whether the number of "different" utterances reported is even ($p_i = E$) or odd ($p_i = O$). Considering that all these parameters are not initialised at the beginning of the run ($c_i = p_i = u_i = n$), there are 27 possible combinations of these parameters, hence the set L_{C_i} encoding the local states of cryptographer i contains 27 possible local states. For each cryptographer the following set of actions is introduced:

$$Act_i = \{nothing, sayequal, saydifferent\}.$$

Actions are performed in compliance with the protocol stated above:

$$\begin{aligned} P_{C_i}(< ENn >) = P_{C_i}(< DYn >) = \{sayequal\}; \\ P_{C_i}(< EYn >) = P_{C_i}(< DNn >) = \{saydifferent\}. \end{aligned}$$

That is: the action of the cryptographer is *sayequal* if either (i) the cryptographer sees two equal coins and did not pay for the dinner and no utterances have been made yet, or (ii)

Final state	**Transition condition**
$< EYn >$	$l_{C_2} =< nnn >$ and ($l_E =< HHH2 >$ or $l_E =< THH2 >$ or $l_E =< HTT2 >$ or $l_E =< HTT2 >$)
$< DYn >$	$l_{C_2} =< nnn >$ and ($l_E =< HTH2 >$ or $l_E =< TTH2 >$ or $l_E =< HHT2 >$ or $l_E =< HHT2 >$)
$< ENn >$	$l_{C_2} =< nnn >$ and ($l_E =< HHH1 >$ or $l_E =< THH1 >$ or $l_E =< HTT1 >$ or $l_E =< HTT1 >$ or $l_E =< HHH3 >$ or $l_E =< THH3 >$ or $l_E =< HTT3 >$ or $l_E =< HTT3 >$ or $l_E =< HHH4 >$ or $l_E =< THH4 >$ or $l_E =< HTT4 >$ or $l_E =< HTT4 >$)
$< DNn >$	$l_{C_2} =< nnn >$ and ($l_E =< HHT1 >$ or $l_E =< THT1 >$ or $l_E =< HTH1 >$ or $l_E =< HTH1 >$ or $l_E =< HHT3 >$ or $l_E =< THT3 >$ or $l_E =< HTH3 >$ or $l_E =< HTH3 >$ or $l_E =< HHT4 >$ or $l_E =< THT4 >$ or $l_E =< HTH4 >$ or $l_E =< HTH4 >$)
$< EYO >$	$l_{C_2} =< EYn >$ and (($Act_{C_1} = saydifferent$ and $Act_{C_3} = saydifferent$) or ($Act_{C_1} = sayequal$ and $Act_{C_3} = sayequal$))
...	...

Table 6.3: Transition conditions for C_2.

the cryptographer sees two different coins and did pay for the dinner and no utterances have been made yet. The conditions for *saydifferent* are similar. In all the remaining cases the protocol P_{C_i} prescribes the action *nothing*.

The evolution function for a cryptographer C_i lists the conditions causing a change in the local state of C_i. Table 6.3 lists some the transition conditions for cryptographer 2 (the conditions counting the number of utterances are not listed explicitly). The conditions for the remaining cryptographers are defined in a similar way.

As it is clear from Table 6.3, the manual encoding of the example may be cumbersome. In this case, and in other examples presented below, it is more convenient to implement a generator of ISPL code using a traditional programming language (such as C++). The source code of MCMAS includes a generator for this example which takes the number of cryptographers as input, and produces as output the ISPL code corresponding to the encoding presented above.

The following set AP of atomic propositions is defined to reason about the example with three cryptographers:

$$AP = \{\mathbf{paid_1}, \mathbf{paid_2}, \mathbf{paid_3}, \mathbf{even}, \mathbf{odd}\}.$$

Correspondingly, the following evaluation function is introduced:

$$\begin{array}{lcl} V(\mathbf{paid_1}) & = & \{g \in G \mid l_{C_1}(g) =< *Y* >\}; \\ V(\mathbf{paid_2}) & = & \{g \in G \mid l_{C_2}(g) =< *Y* >\}; \\ V(\mathbf{paid_3}) & = & \{g \in G \mid l_{C_3}(g) =< *Y* >\}; \\ V(\mathbf{even}) & = & \{g \in G \mid l_{C_i}(g) =< *E > \text{ for every } i\}; \\ V(\mathbf{odd}) & = & \{g \in G \mid l_{C_i}(g) =< *O > \text{ for every } i\}. \end{array}$$

$< *Y* >$ denotes a local state in which the value of p_i is Y (i.e., the cryptographer paid for dinner), while $< *E >$ and $< *O >$ denote local states in which the value of u_i is either E or O (i.e., either an even or an odd number of utterances have been made). Various properties of this deontic interpreted system, denoted by DIS_{DC3}, are expressed using AP. For instance:

$$DIS_{DC3} \models (\mathbf{odd} \wedge \neg\mathbf{paid_1}) \implies AX(K_{C1}(\mathbf{paid_2} \vee \mathbf{paid_3}) \wedge \neg K_{C1}(\mathbf{paid_2}) \wedge \neg K_{C1}(\mathbf{paid_3})).$$

This formula expresses the claim made at the beginning of this section: if the first cryptographer did not pay for dinner and the number of “different” utterances is odd, then the first cryptographer knows that either the second or the third cryptographer paid for dinner; moreover, in this case, the first cryptographer does not know which of these two is the payer. Analogously, it is possible to check that, if a cryptographer paid for dinner, then there will be an odd number of “different” utterances, that is:

$$DIS_{DC3} \models (\mathbf{paid_1} \vee \mathbf{paid_2} \vee \mathbf{paid_3}) \implies AF(\mathbf{odd}).$$

Consider now the group Γ of the three cryptographers. An interesting property to check is the following:

$$DIS_{DC1} \models \mathbf{even} \implies AX(C_\Gamma(\neg\mathbf{paid_1} \wedge \neg\mathbf{paid_2} \wedge \neg\mathbf{paid_3})).$$

This formula expresses the fact that, in presence of an even number of “different” utterances, it is common knowledge that none of the cryptographers paid for the dinner. Hence, in this protocol common knowledge can be achieved anonymously. Experimental results for this example are discussed in Section 6.5.

Similarly to the bit transmission problem, instances of the protocol where one or more of the cryptographers do not behave correctly (i.e., they “cheat”) can be analysed. More details can be found in [Kacprzak et al., 2006].

6.2.1 A different encoding

The encoding presented above is not the most efficient encoding of the protocol of the dining cryptographers. For instance, the number of utterances (even or odd) is stored separately in each cryptographer, but this information is required only once; similarly, the outcomes of coin tosses are stored in the environment, and each cryptographer has repeated information about them (seeing two equal or different coins).

A different encoding is proposed in [Kacprzak et al., 2006], where an agent is associated to each coin and an agent is introduced to "count" the utterances. Verification is performed by taking the distributed knowledge in a group of agents: intuitively, each group contains all the information needed by a cryptographer, as defined above. This encoding permits an improvement of the performance of MCMAS in the order of 30% on average (see the experimental results appearing in [Kacprzak et al., 2006]).

Essentially, this is an automata-based approach translated into the ISPL language. A discussion of the relationships between automata and agents is beyond the scope of this thesis; more details about it and a comparison with other techniques to encode the protocol of the dining cryptographers can be found in [Kacprzak et al., 2006] and in Section 6.5.

6.3 Strategic games

This section presents three examples of strategic reasoning in deontic interpreted systems. The first two examples make use of uniform agents, because part of the information is hidden. The third example in Section 6.3.3 is an example of a game with perfect information, and it is shown that the encoding using deontic interpreted systems is as natural as the standard game-theoretic approach.

6.3.1 A simple card game

The example depicted in Figure 2.3 and described in Section 2.1.7.3, page 33 (see also [Jonker, 2003, Jamroga, 2004b]), is encoded in the formalism of deontic interpreted systems by introducing two agents: one agent P encodes the player, and another agent encodes the environment. Local states for P are defined as follows:

$$L_P = \{a1, k1, q1, a2, k2, q2\}$$

representing that P holds Ace, King on Queen, either in step 1 or in step 2 (i.e., after changing the card). The actions for P are:

$$Act_P = \{keep, swap, none\}$$

Final state	**Transition condition**
$a2$	$(l_P = a1 \text{ } and \text{ } Action = keep)$ or $(l_P = k1 \text{ } and \text{ } Action = swapandl_E = q)$ or $(l_P = q1 \text{ } and \text{ } Action = swapandl_E = k)$ or
$k2$	$(l_P = k1 \text{ } and \text{ } Action = keep)$ or $(l_P = a1 \text{ } and \text{ } Action = swapandl_E = q)$ or $(l_P = q1 \text{ } and \text{ } Action = swapandl_E = a)$ or
$q2$	$(l_P = q1 \text{ } and \text{ } Action = keep)$ or $(l_P = k1 \text{ } and \text{ } Action = swapandl_E = a)$ or $(l_P = a1 \text{ } and \text{ } Action = swapandl_E = k)$ or

Table 6.4: Transition conditions for P.

and they are performed in compliance with the following protocol:

$$\begin{aligned} P_P(a1) = P_P(k1) = P_P(q1) = \{keep, swap\}; \\ P_P(a2) = P_P(k2) = P_P(q2) = \{none\}. \end{aligned}$$

Intuitively, the protocol prescribes that P can either keep or change its card in the first round, while in the next round no action is performed. The transition conditions for P are listed in Table 6.4, and they translate formally the possible evolution of the system depicted in Figure 2.3.

The environment is modelled with the following set of local states:

$$L_E = \{a, k, q\}.$$

It is assumed that the environment does not perform actions, therefore $Act_E = \{none\}$. The protocol P_E maps every state to this action, and there is no evolution of local states.

Only one proposition is introduced, expressing that the player wins the game:

$$AP = \{\mathbf{pwin}\}$$

with the corresponding evaluation function:

$$\begin{aligned} V(\mathbf{pwin}) \quad = \quad & \{g \in G \mid (l_P(g) = a2 \text{ and } l_E(g) = k) \\ & \text{or } (l_P(g) = k2 \text{ and } l_E(g) = q) \\ & \text{or } (l_P(g) = q2 \text{ and } l_E(g) = a)\}. \end{aligned}$$

Let DIS_{Cards} denote the deontic interpreted system described above. The ISPL code corresponding to DIS_{Cards} is depicted in Figure 6.3. As expected, it is possible to verify that

$$DIS_{Cards} \models \mathbf{init} \implies \langle\langle P \rangle\rangle X(\mathbf{pwin})$$

(where **init** is a proposition true in the set of initial states). However, in a $\{P\}$-uniform deontic interpreted system, it is possible to verify that

$$DIS_{Cards} \not\models_{\{P\}} \mathbf{init} \implies \langle\langle P\rangle\rangle X(\mathbf{pwin}).$$

This last check is performed by including the option `-u` in the command line of MCMAS.

6.3.2 RoadRunner and Coyote

This example illustrates further the different meaning of ATL operators in Γ-uniform deontic interpreted systems and in non-deterministic deontic interpreted systems.

RoadRunner is running in a hilly region of the desert; the main road splits in two small lanes just before the entrance of two tunnels under a mountain. RoadRunner can pick randomly either tunnel; the tunnels are identical and very narrow. Coyote knows Road-Runner has to enter one of the two tunnels, and so he has bought a special tunnel-blocking device from ACME Inc. to catch RoadRunner. The device may be placed in front of either exit of the tunnel (see Figure 6.4).

The example is modelled as a deontic interpreted system DIS_{RC} by taking two agents, an agent C for Coyote and an agent R for RoadRunner. The set of local states for RoadRunner includes two local states:

$$L_R = \{\mathit{left}, \mathit{right}\}$$

representing which tunnel RoadRunner is going to enter. The only action defined for R is *run*, and the protocol assigns this action to every local state. Moreover, local states of R do not change.

Coyote is modelled by means of three local states:

$$L_C = \{\mathit{planning}, \mathit{catch}, \mathit{fail}\}.$$

The set of actions for Coyote is as follows:

$$Act_C = \{\mathit{placeleft}, \mathit{placeright}, \mathit{none}\}.$$

Actions are performed in compliance with the following protocol:

$$\begin{aligned} &P_C(\mathit{planning}) = \{\mathit{placeleft}, \mathit{placeright}\} \\ &P_C(\mathit{catch}) = P_C(\mathit{fail}) = \{\mathit{none}\}. \end{aligned}$$

The evolution function for Coyote prescribes that the next local state for C is *catch* if

```
Agent env
   Lstate = { a,k,q};
   Lgreen = { a,k,q};
   Action = { none,none2 };
   Protocol:
      a: {none};
      k: {none};
      q: {none};
   end Protocol
   Ev:
      a if ( Lstate = a);
   end Ev
end Agent

Agent player
   Lstate = {a1,k1,q1,a2,k2,q2};
   Lgreen = {a1,k1,q1,a2,k2,q2};
   Action = {keep,swap,none};
   Protocol:
      a1: {keep,swap};
      k1: {keep,swap};
      q1: {keep,swap};
      a2: {none};
      k2: {none};
      q2: {none};
   end Protocol
   Ev:
      a2 if ( ( Lstate = a1 ) and (Action=keep) ) or
         ( (Lstate = k1 ) and (Action=swap) and (env.Lstate=q)) or
         ( (Lstate = q1 ) and (Action=swap) and (env.Lstate=k));
      k2 if ( ( Lstate = k1 ) and (Action=keep) ) or
         ( (Lstate = a1 ) and (Action=swap) and (env.Lstate=q)) or
         ( (Lstate = q1 ) and (Action=swap) and (env.Lstate=a));
      q2 if ( ( Lstate = q1 ) and (Action=keep) ) or
         ( (Lstate = k1 ) and (Action=swap) and (env.Lstate=a)) or
         ( (Lstate = a1 ) and (Action=swap) and (env.Lstate=k));
   end Ev
end Agent

Evaluation
   pwin if ( (player.Lstate = a2) and ( env.Lstate=k) ) or
      ( (player.Lstate = k2) and ( env.Lstate=q) ) or
      ( (player.Lstate = q2) and ( env.Lstate=a) );
end Evaluation

InitStates
   ( ( player.Lstate=a1 and env.Lstate=k ) or
     ( player.Lstate=a1 and env.Lstate=q ) or
     ( player.Lstate=k1 and env.Lstate=q ) or
     ( player.Lstate=k1 and env.Lstate=a ) or
     ( player.Lstate=q1 and env.Lstate=a ) or
     ( player.Lstate=q1 and env.Lstate=k ) ) and
     ( env.Action=none and
     (player.Action=keep) or (player.Action=swap) );
end InitStates

Groups
   g1 = {player};
end Groups

Formulae
   init -> <g1>X(pwin);
-- init is a proposition true in the set of initial states
end Formulae
```

Figure 6.3: ISPL code for the card game.

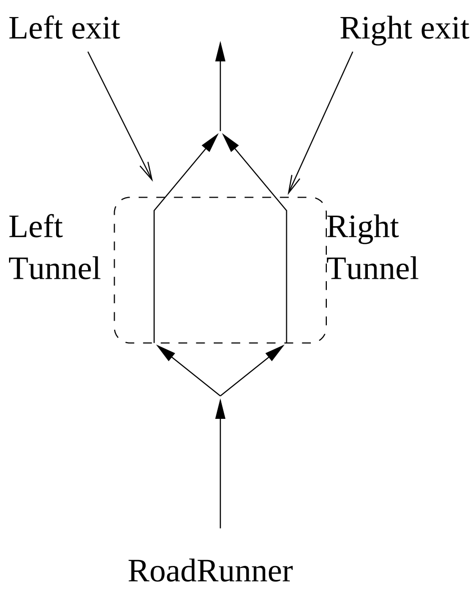

Figure 6.4: Diagram for RoadRunner and Coyote.

l_C = *planning* and Coyote places the special device in front of the tunnel chosen by RoadRunner; in the other case the local state changes to *fail* (notice: the evolution function for C depends on the local state of R, which is treated in this example as the environment).

In the initial state, the local state of C is *planning*, and the local state of R is either *left* or *right*. One proposition is introduced to reason about this example: $AP = \{\mathbf{catch}\}$, with the corresponding evaluation function:

$$V(\mathbf{catch}) \;=\; \{g \in G \mid l_C(g) = \mathit{catch}\}.$$

By using MCMAS Coyote discovers that:

$$\mathit{DIS}_{RC} \models \mathbf{init} \implies \langle\langle C\rangle\rangle X(\mathbf{catch})$$

i.e., it is the case that in the initial state Coyote has a strategy to catch RoadRunner (where **init** is a proposition true in the set of initial states). In fact, any external observer could verify that Coyote knows this very well:

$$\mathit{DIS}_{RC} \models K_C(\mathbf{init} \implies \langle\langle C\rangle\rangle X(\mathbf{catch})).$$

Unfortunately, immediately after placing the ACME device in front of the tunnel, Coyote realises that he was assuming a *lucky* guess on where to place the device. Indeed, under the assumption Coyote is uniform, the formula turns to be false:

$$\mathit{DIS}_{RC} \not\models_{\{C\}} \mathbf{init} \implies \langle\langle C\rangle\rangle X(\mathbf{catch}).$$

From RoadRunner's point of view, however, it is more useful and prudent to reason about what the clumsy Coyote may bring about. Thus, RoadRunner should be more interested in the verification of the non-deterministic deontic interpreted system to discover that it is possible that Coyote catches him. In other words, to analyse the scenario from RoadRunner's point of view, it is possible to check:

$$DIS_{RC} \models K_R(\mathbf{init} \implies \langle\!\langle C \rangle\!\rangle X(\mathbf{catch})).$$

6.3.3 Nim

Nim is a two player game where players in turns remove any number of objects from one of a certain number of heaps. Typically, 3 heaps are present and the game starts with 3 objects in the first heap, 4 in the second, and 5 in the third. The player who takes the last object wins. In a variation of this game, called *Misère*, the player who takes the last object loses. This is a game with *perfect* information.

The example is modelled as a deontic interpreted system by introducing three agents: one agent for each player and one agent for the environment, encoding the three heaps. The set of local states for the environment includes all the possible combinations of objects in the heaps (for the case of 3-4-5 heaps, there are 60 possible states). The environment does not perform any action; the local states of the environment change only according to the actions performed by each player.

The set of local states for the players includes a copy of the local states of the environment (which the agents can observe), and their actions include all the possible moves that can be performed, i.e., removing a number of objects from a heap (there are 60 such actions in the case of 3-4-5 heaps), and a "waiting" action when the other player is moving. In a given local state, the protocol for the agent permits all the actions which remove a number of objects less or equal to the number of remaining objects in each heap. The evolution function for the players either copies the local state of the environment, or it moves the agent to a waiting state.

Two propositions are introduced, in addition to the proposition **init** which holds in the set of initial states:

$$AP = \{\mathbf{p1\_removelast}, \mathbf{p2\_removelast}\}.$$

The evaluation function defines the proposition **p1_removelast** to be true when the first player observes that all the heaps are empty (notice that the observation of an agent follows the action of removing objects). The proposition **p2_removelast** is defined analogously. Similarly to the example of the protocol of the dining cryptographers presented in Section 6.2, the ISPL code representing the deontic interpreted system DIS_{Nim} for Nim is generated using a C++ program, included in the source distribution of MCMAS. The model checker confirmed the known result that the first player can force a win *both* for the Nim and the Misère scenario, by verifying the following formulae:

$$DIS_{Nim} \models \textbf{init} \implies \langle\langle player1 \rangle\rangle[\neg\textbf{player2\_removelast}\ U\ \textbf{player1\_removelast}]$$

$$DIS_{Nim} \models \textbf{init} \implies \langle\langle player1 \rangle\rangle[\neg\textbf{player1\_removelast}\ U\ \textbf{player2\_removelast}]$$

Experimental results for this example are reported in Section 6.5.

6.4 Diagnosability and other specification patterns

Diagnosability is defined as the feasibility of a diagnosis in a given system, based on the observations of *sensors* and *actuators* of the system. A formal investigation of diagnosability appears in [Sampath et al., 1995, Cimatti et al., 2003]. In particular, given a **CTL** model $M = (S, R, V, I)$, [Cimatti et al., 2003] define a *diagnosis condition* to be a pair of non-empty sets of states $c_1, c_2 \subseteq S$ of the system; a diagnosis condition is usually written $(c_1 \perp c_2)$, where $\perp$ is a separator for the two sets of states. Given a set of variables of the system that can be observed by a diagnoser, a diagnosis condition $(c_1 \perp c_2)$ is *diagnosable* iff there are no two execution traces π_1 and π_2 such that π_1 leads to a state in c_1 and π_2 leads to a state in c_2, with the additional constraint that the observable variables remain the same for all the states in π_1 and π_2. For instance, fault detection is expressed in terms of a diagnosis condition as ($\textbf{fault} \perp \neg\textbf{fault}$): this means that there are no two traces such that one trace leads to a faulty state and the other to a non-faulty state when the observable variables remain the same.

It has been shown by [Cimatti et al., 2003] that a temporal-only model checker like NuSMV can be used for the formal verification of diagnosability. The key idea is two build two copies of the system, run them in parallel and constrain the observable variables of both copies to remain equal using the tools provided by the model checker (e.g., using the `INVAR` construct in NuSMV). For instance, a copy of the system may be encoded with a NuSMV module called `test`, and the other copy may be encoded with a module called `twin`. If the execution mode of the module is described by a private variable called `mode`, which takes the value `faulty` when the module is in a faulty state, then fault detection can be expressed by the formula:

$$AG(\neg(\texttt{test.mode} = \texttt{faulty} \wedge \texttt{twin.mode} \neq \texttt{faulty})).$$

A number of systems have been diagnosed using this technique [Cimatti et al., 2003].

The formal verification of diagnosability can benefit from a temporal-epistemic characterisation, and by treating the diagnoser and the system as agents. Indeed, diagnosability is expressed naturally by ascribing a form of knowledge to a diagnoser: a diagnoser is able to diagnose a diagnosis condition $(c_1 \perp c_2)$ iff the diagnoser always knows whether $\neg c_1$ is the case, or $\neg c_2$ is the case. This can be expressed formally in the framework of deontic interpreted systems: the system in which diagnosability needs to be verified may be modelled by introducing a particular agent D (the diagnoser) that stores in its local states the observable variables, and the original system may be modelled by another agent S, "observed" by the diagnoser. In this way, diagnosability is expressed by the formula:

$$AG(K_D(\neg c_1) \vee K_D(\neg c_2)).$$

Notice that the knowledge operator forces the observable variables to remain unchanged; also, this definition of diagnosability is based solely on one model, thereby reducing the size of the problem to be verified if compared to temporal-only model checking.

The concept of distributed knowledge in a group permits a generalisation of the concept of diagnosability. Let Δ be a subset of the set of agents in a deontic interpreted system: intuitively, Δ is a set of a diagnosers, each of which is responsible for the monitoring of a particular aspect of the system (e.g., a part of the observable outputs) while ignoring the remainder. A diagnosis condition $(c_1 \perp c_2)$ is diagnosable by a group of agents iff

$$AG(D_\Delta(\neg c_1) \vee D_\Delta(\neg c_2))$$

The following example illustrates the verification of diagnosability in deontic interpreted systems encoding an electrical circuit composed of a cascade of circuit breakers, a source, and LEDs. This example is part of the test examples of the tool Livingstone, "a model-based health monitoring system developed at NASA Ames Research Center" [Pecheur and Simmons, 2000][1]. The circuit is represented in Figure 6.5.

Each circuit breaker is allowed to be in one of the following states: `on, off, tripped, blown, ufault`. `on` and `off` are "green" states. `tripped` is a resettable fault, `blown` is a non recoverable fault, and `ufault` denotes an unknown fault. A Controller sends (arbitrary) commands to the circuit breakers, and a Diagnoser reads the commands and the outputs as defined in the Livingstone model.

Various assumptions can be made while modelling this example as a deontic interpreted system DIS_{Circ}. Here the following are considered:

- Each circuit breaker is an agent; each led is an agent; the source is an agent.
- For each circuit breaker, it is assumed that a commander agent is allowed to send

[1]The formal encoding of this example has been carried out while the author was visiting Dr. Charles Pecheur at NASA Ames Research Center, between July and September 2004.

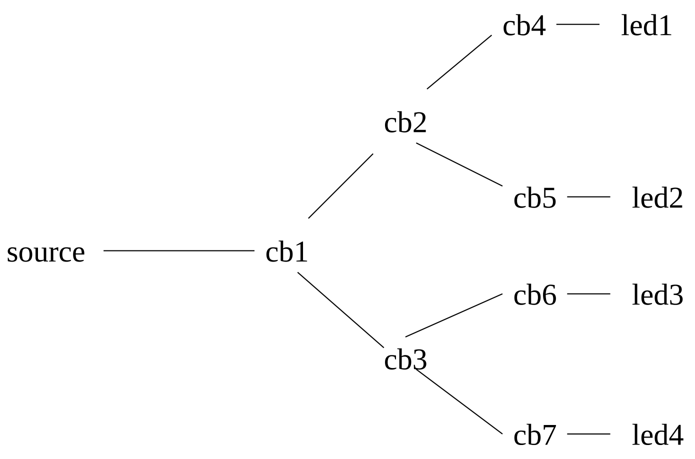

Figure 6.5: A circuit for diagnosability.

random commands to the circuit breakers.

- Two diagnosers are introduced: the first can see the output of the source, the second can see the LEDs.

The ISPL code for this example has been obtained by translating the Livingstone specification into ISPL code, and it is available from the `examples/` directory of MCMAS.

As an example of diagnosability, it is possible to check that the diagnoser obtained by considering the distributed knowledge of the two diagnosers is not able to detect faults correctly, i.e.:

$$DIS_{Circ} \not\models AG(D_{\Delta}(\mathbf{faulty}) \vee D_{\Delta}(\neg\mathbf{faulty}))$$

where **faulty** $= \bigvee f_i$ is a proposition denoting that some component i of the circuit is not working correctly (encoded by the proposition f_i), and Δ is the group composed by the two diagnoser agents for the output and the LEDs. This is because, under the above assumptions, the diagnoser is not able to distinguish between "correct" `off` states and faulty states. The same result can be obtained by using the twin model of the circuit and by verifying it using NuSMV. Experimental results for the two approaches are presented in Section 6.5.

6.4.1 Verification of recoverability

The use of deontic interpreted systems enables the verification of other more complex specification patterns, that are not expressible (or, at least, not in an easy way) using temporal-only formulae. The aim of this section is to provide a formal description and an example of properties such as "the diagnoser knows that, assuming correct behaviour of the system, the system will recover from a given faulty state" (*recoverability* is the ability of a system to recover from some faulty state).

Instead of using the "deontic" accessibility relations R_i^O, correct behaviour can be characterised in terms of *local propositions* [Lomuscio and Sergot, 2003, Anderson, 1958]. Let $g_i \in AP$ be a proposition true in the green states of agent i (notice that these propositions are part of Livingstone models and thus, in the examples from NASA Ames, no manual intervention is required to encode these propositions). For any deontic interpreted system DIS, the following equivalences hold:

$$DIS \models O_i\varphi \Leftrightarrow (g_j \implies \varphi);$$
$$DIS \models \hat{K}_i^j\,\varphi \Leftrightarrow K_i(g_j \implies \varphi).$$

Let $f \in AP$ be a proposition denoting some faulty state, and let φ be a formula denoting some desired states of affairs. Then, the ability of a diagnoser Δ to diagnose recoverability from f on a deontic interpreted system DIS, assuming "correct" functioning conditions for the agents in Γ, can be expressed as:

$$DIS \models f \implies D_\Delta(\neg E[F_\Gamma\ U\ (F_\Gamma \wedge \neg\varphi)]).$$

In the previous expression, $F_\Gamma = \bigwedge_{i\in\Gamma} g_i$ is a Boolean expression composed by the conjunction of the local proposition denoting correct behaviour for agents in Γ. Intuitively, the "until" part of the formula states that there is no "correct" temporal path from "faulty" states which will not reach a state in which φ holds. This fact, in turn, is distributed knowledge between the diagnosers.

A concrete example of recoverability is expressible using the example presented in Figure 6.5 and its formal encoding presented in the previous Section. Let **g_cb** be a formula obtained by taking the conjunction of the local propositions expressing correct behaviour for all the circuit breakers. Let **led_on** be a proposition denoting the global states in which LEDs are on. It is possible to verify automatically with MCMAS that the following formula does not hold in DIS_{Circ}:

$$DIS_{Circ} \not\models \textbf{faulty} \implies D_\Delta(\neg E[\textbf{g\_cb}\ U\ (\textbf{g\_cb} \wedge \neg\textbf{led\_on})]).$$

Intuitively, the formula expresses that the diagnoser is not able to diagnose recoverability because circuit breakers may be in unrecoverable faulty states, and the diagnoser cannot distinguish them from recoverable faulty states.

6.5 Experimental results

This section presents the experimental results obtained in the verification of the examples presented above using MCMAS. Notice that, for all the examples, time results are not

	$\|M\|$	OBDDs **variables**	**Memory (MBytes)**
DIS_{BTP1}	$\approx 4 \cdot 10^6$	19	≈ 4.5
DIS_{BTP2}	$\approx 4 \cdot 10^6$	19	≈ 4.5

Table 6.5: Space requirements for the bit transmission problem.

Model construction	**Verification**	**Total**
0.045sec	<0.001sec	0.045sec

Table 6.6: Running time (for one formula) for the bit transmission problem.

affected by the structure of the formula, nor by the number of formulae being verified. This is caused by the fact that time required by the algorithm of Figure 3.12 is a fraction (in the order of 0.1% – 0.5%) of the time required for the construction of the OBDDs representing protocols, temporal relations, reachable states, etc. Therefore, it makes sense to discuss simply the time required "for an example": this is the time reported in the tables below. All the tests have been performed using a 2.8GHz Intel Pentium IV, 1GB of RAM, running Linux 2.6.8, with the exception of the diagnosability example.

6.5.1 The bit transmission problem

In this example there are 4 local states and 3 actions for S, 5 (or 6) local states and 2 actions for R, and 4 local states and 4 actions for E. In total, the size of the state space is $\approx 2 \cdot 10^3$. The size of the model is defined here as the size of the state space with the addition of the size of the relations. The size of the relations can be approximated with the size of the state space to the power of two, hence $|DIS_{BTP1}| \approx |DIS_{BTP2}| \approx 4 \cdot 10^6$ (notice that the sizes of both cases are very similar). Also, a measure of the size of the model is given by the number of Boolean variables used to encode local states and actions. Table 6.5 reports the space requirements for DIS_{BTP1} and DIS_{BTP2}.

Average time results for the bit transmission problem (both examples) are reported in Table 6.6. The verification time for one formula has been computed by evaluating the time difference between two runs of the same ISPL code, but with a different number of formulae. In the first run only one formula has been verified, while in the second run 20 formulae have been verified.

6.5.2 The protocol of the dining cryptographers

The protocol of the dining cryptographers is suitable for testing the scalability of MCMAS, because the ISPL code corresponding to any number of cryptographers is generated automatically by a C++ code taking the number of cryptographers as the only input parameter (this generator is distributed with the source code of MCMAS).

N.Crypt.	$\lvert M\rvert$	OBDDs vars.	OBDDs nodes	Memory (MBytes)
3	$\approx 7 \cdot 10^{13}$	46	$\approx 10^4$	≈ 4.4
4	$\approx 2 \cdot 10^{18}$	62	$\approx 6 \cdot 10^4$	≈ 5.2
5	$\approx 2 \cdot 7.5^{22}$	76	$\approx 8 \cdot 10^4$	≈ 5.6
6	$\approx 1.2 \cdot 10^{27}$	90	$\approx 1.6 \cdot 10^5$	≈ 7.1
7	$\approx 2 \cdot 10^{31}$	104	$\approx 1.7 \cdot 10^5$	≈ 7.5
8	$\approx 1.3 \cdot 10^{36}$	120	$\approx 1.2 \cdot 10^7$	≈ 230

Table 6.7: Space requirements for the dining cryptographers.

N.Crypt.	Model construction	Verification	Total
3	1.1sec	<0.1sec	1.2sec
4	5.1	<0.1	5.2
5	18.7	<0.1	18.8
6	125.9	≈0.1	126.0
7	649	≈0.1	649
8	9643	≈1	9644

Table 6.8: Running time (for one formula) for the protocol of the dining cryptographers.

Table 6.7 presents the memory requirements for the verification of scenarios from three to eight cryptographers (the size of the model is defined as in the previous section). This table shows the number of OBDD nodes allocated to encode the parameters: notice that this is typically a fraction of the size of the model. Such a difference shows that the reduction in the state space obtained using OBDDs is significant in this example.

Average time requirements for the verification of one formula are reported in Table 6.8.

6.5.3 Strategic games

The ISPL code corresponding to DIS_{RC} and DIS_{Cards} generates very small examples. To evaluate the performance of the verification of **ATL** operators using MCMAS, the Nim example has been verified using 3-4-5 heaps (this example is denoted by DIS_{Nim345}) and with 5-5-5 heaps (this example is denoted by DIS_{Nim555}). Space results for all the examples are reported in Table 6.9. Notice that the number of OBDDs variables required for DIS_{Nim345} is similar to the number of variables for DIS_{Nim555}: this is caused by the fact that 60 local states are possible for the environment in the case of 3-4-5 heaps, and 125 local states in the case of 5-5-5 heaps. In the first case, 6 Boolean variables are required, while in the second case 7 Boolean variables are required. Although the difference in the number of variables is small between the two examples, the number of OBDD nodes and the Memory usage increase by a factor of ten. Thus, it seems that this example does not scale up as well as the example of the dining cryptographers, probably because a “good” reordering of variables cannot be found by the OBDD library to reduce the number of

Example	OBDDs vars.	OBDDs nodes	Memory (MBytes)
DIS_{Cards}	13	≈ 400	4.1
DIS_{RC}	9	115	4.1
DIS_{Nim345}	51	$\approx 7 \cdot 10^4$	≈ 7.1
DIS_{Nim555}	57	$\approx 8 \cdot 10^5$	≈ 41

Table 6.9: Space requirements for strategic games.

Example	Model construction	Verification	Total
DIS_{Cards}	0.15 sec	< 0.01sec	0.15sec
DIS_{RC}	0.19	< 0.01	0.19
DIS_{Nim345}	18	< 0.2	18
DIS_{Nim555}	248	≈ 0.3	248

Table 6.10: Running time (for one formula) for strategic games.

nodes.

Time results for these examples are reported in Table 6.10. Notice the difference in time between DIS_{Nim345} and DIS_{Nim555}, for the same reasons presented above.

While the time results for non-deterministic and Γ-uniform deontic interpreted systems are the same for the examples DIS_{Cards} and DIS_{RC}, it is likely that for larger examples the time requirements for verification in Γ-uniform deontic interpreted systems will be larger. In this case, the verification time may depend on the structure of the formula (false formulae requiring more time).

6.5.4 Diagnosability

The performance of MCMAS to verify diagnosability has been investigated in comparison with the performance of NuSMV. Space and time results for the verification of one formula using either MCMAS or NuSMV are reported in Table 6.11. The experimental results have been obtained using a 3.0 GHz Intel Pentium IV, 2GBytes of RAM, running Linux 2.6.9, available temporarily at NASA Ames.

The results in Table 6.11 refer to the verification of one formula for diagnosability. Similarly to the other examples, verification of recoverability using MCMAS requires a similar amount of time and space.

Tool	Time	OBDDs vars
MCMAS	2.39sec	90
NuSMV	10.47sec	235

Table 6.11: Average verification results for diagnosability.

6.5.5 Discussion

The experimental results presented above show that the performance of MCMAS remains, on average, well below the worst case requirements.

- **Space requirements**. The size of a model generated from ISPL code is exponential in the size of the code itself. For instance, consider the sizes of the models in Table 6.7: these range from 10^{13} to 10^{36} by increasing (linearly) the number of cryptographers. Structures of this size cannot be dealt with explicitly using the hardware currently available (1000Gbytes = 10^{12} bytes is the size of the biggest hard drive available at present). Nevertheless, the encoding of this example is reduced to structures of size $< 10^7$ using MCMAS and OBDDs, thereby enabling verification of examples that cannot be verified using an explicit manipulation of the parameters appearing in the algorithm of Figure 3.12. As mentioned in Section 4.3, it is known that under certain circumstances the size of the OBDDs representing the Boolean formulae may be exponential in the number of Boolean variables (thereby matching the size of the "explicit" model). However, this appears not to be the case for any of the examples verified with MCMAS, in line with previous experiments.

- **Time requirements**. As mentioned in Section 2.2.5, page 57, the operation of Boolean quantification on OBDDs may require time exponential in the number of variables. The time required for the actual verification of formulae (after the construction of the necessary parameters) shows that this is not the case for the examples above. Indeed, verification requires a number of quantification operations when temporal operators need to be verified, but the time required for verification is typically a fraction of the time required for the construction of the parameters. By running MCMAS with increased verbosity it is possible to check that the time required for the construction of the parameters is influenced by the search of a reordering of variables that offers a compact representation of the OBDDs (this is known to be a **NP**-complete problem). This appears to be the bottleneck for the verification of the large examples presented above.

- **Complexity considerations**. The complexity of model checking an ISPL program can be estimated by reducing it to a concurrent program (see Section 4.2), using the simple mapping provided in [Raimondi and Lomuscio, 2005b]. Therefore, model checking temporal-epistemic properties in ISPL programs is a **PSPACE**-complete problem (notice that this is true also for VerICS programs). However, the actual implementation of MCMAS requires, in the worst case, an exponential time to perform verification, because OBDDs are used. The same result applies to all the available model checkers for multi-agent systems.

While the aim of this thesis is not to build the most efficient model checker for multi-agent systems, it still makes sense to compare the experimental results obtained using MCMAS with other model checkers. The comparison can be carried out on two levels:

1. **Qualitative comparisons**. The model checkers mentioned in Section 2.3 all have different input languages. For instance, MCK defines local states of agents using observation functions, and VerICS uses networks of automata to model multi-agent systems. If a single example were to be encoded using these model checkers and using MCMAS the results would be, in fact, *three different examples.* For instance, the protocol of the dining cryptographers can be encoded using any of the previous model checkers. However, the size of the model generated using the input language of MCK (which can be estimated using the number of Boolean variables required to encode the parameters) would be much smaller than the model generated by MCMAS or VerICS, thanks to the possibility of using observation functions in MCK. Therefore, for this particular example and for a given number of cryptographers, MCK would verify always a smaller model. In other instances a network of communicating automata is more efficient than MCK (for instance, to model mutual exclusion problems).

   Under this perspective, MCK seems to be more suitable for examples where information is shared among the agents, as it can be stored as part of the environment. MCMAS and VerICS, instead, seem to be more suitable for "autonomous" agents, with a particular emphasis on coordination using shared actions in VerICS. Coordination via shared actions may ease the description of examples in certain circumstances (e.g., mutual exclusion), but it may not be suitable in others (e.g., when agents act autonomously, for instance in the bit transmission problem).

2. **Quantitative comparisons**. Taking into account the above observations, a comparison of different model checkers has been presented in [Ditmarsch et al., 2005] and in [Kacprzak et al., 2006]. The first paper compares MCK, MCMAS, and the tool DEMO for the verification of propositional dynamic epistemic logic^2 on the verification of a particular example: the Russian cards game. For this example, the tool DEMO performs better than MCK and MCMAS (which have comparable results), because the input language of DEMO can encode this example in an efficient way.

   VerICS and MCMAS are compared in [Kacprzak et al., 2006]. The approach taken in this work is different from other comparison attempts, in that a common representation for an example is defined first, and then formulae are verified in the common representation. In particular, the protocol of the dining cryptographers is encoded using a network of automata in VerICS, while MCMAS defines an agent for every automaton defined in VerICS: this approach defines two models of identical size. Various formulae are verified, including the following (where n is the number

^{2}The tool is available from [Ditmarsch et al., 2006].

N. crypt.	MCMAS **time**	**VerICS for 6.4**	**VerICS for 6.5**
4	4sec	$\approx$ 31000sec	$<$ 1sec
5	6	$\approx$ 106000	$<$ 1sec
6	424	N/A	$<$ 1 sec
8	8101	N/A	$<$ 1 sec
100	N/A	N/A	4sec
1000	N/A	N/A	520

Table 6.12: Experimental results for MCMAS and VerICS.

of cryptographers):

$$AG(\mathbf{even} \implies K_1(\bigwedge_{i\in\{1,\dots,n\}} \neg\mathbf{paid_i})); \tag{6.4}$$

$$AG(\neg\mathbf{paid}_1 \implies K_1(\bigvee_{i\in\{2,\dots,n\}} \neg\mathbf{paid}_i)). \tag{6.5}$$

The first formula expresses the true claim that, if there is an even number of utterances, then the first cryptographer knows that none of the cryptographers paid for the dinner. The second formula is false, and it expresses that if the first cryptographer did not pay for the dinner, then he knows that some of the remaining cryptographers paid for it. The time required for the verification of these formulae is reported in Table 6.12. These results confirm that the verification time in MCMAS is not affected by the structure of the formula: Formula 6.4 and Formula 6.5 required the same amount of time for verification, reported in the first column. On the contrary, verification times for VerICS are crucially dependent on the structure of the formula being verified. The reason is the structure of the bounded model checking algorithm implemented by VerICS: if a counter-example is found with a small value of the bound (as in the case of 6.5) verification is extremely efficient and the tool can check examples with up to 1000 cryptographers (last column). For true formulae, however, maximal paths need to be encoded as Boolean formulae, and this operation may be inefficient: this is confirmed by the time results for Formula 6.4.

These experimental results show that there is no “best” model checker, nor “best” technique. Instead, depending on the example being verified, a tool may be better than another because of its input language and other features, and a verification technique may be better than another due to the particular structure of the formulae being verified.

Given the comparisons above, the techniques and the tool presented in this thesis seem to offer, on average, a satisfactory performance.

Chapter 7

Conclusion

7.1 Contribution

The goal of this thesis has been the development of techniques and tools for the formal verification of multi-agent systems using model checking. The main contributions to this area of research are summarised below:

- Theoretical contributions: traditional OBDD-based methodologies for temporal-only model checking have been extended to multi-modal logics for time, knowledge, correct behaviour, and strategies. To this end, the Boolean encoding of the parameters required by the model checking algorithm has been redefined in terms of the formalism of deontic interpreted systems. Additionally, complexity results have been presented for model checking these logics, and for model checking compact representations.

- Development of MCMAS: this is a software tool developed in C/C++ for the automatic verification of deontic interpreted systems. The tool defines the language ISPL for describing examples and it implements OBDD-based procedures for efficient verification.

- Application examples: various multi-agent systems scenarios have been encoded, including communication and anonymity examples, hardware diagnosability, and strategic reasoning. Experimental results have been presented confirming the effectiveness of this approach.

7.2 Benefits and comparisons

Traditional temporal-only specification patterns (in the sense of [Dwyer et al., 1998]) enable the formalisation of a number of requirements for complex systems. In many circumstances, however, model checking techniques for multi-agent systems introduce substantial benefits with respect to temporal-only model checking. These benefits include:

- a richer expressivity: in addition to temporal reasoning, multi-agent systems allow to reason about "knowledge" (and other modalities) in a formal way, for instance when describing communication and security protocols;
- a more intuitive language for expressing requirements: the key properties of many scenarios are difficult to express in terms of temporal-only logic. The addition of other modal operators make clearer the correspondence between logic formulae and plain text requirements;
- an improved efficiency: as shown in the example of diagnosability, the direct verification of other modalities, in addition to the temporal modalities, may reduce the complexity of the verification problem.

This thesis specifically addressed the above points by providing model checking methodologies and a prototype model checker, that has been tested against a number of examples.

Comparisons. References and comparisons with related work appear in previous chapters (in particular, see Section 6.5.5, page 126). On a theoretical level, the main contributions of this thesis differ from the works presented in Section 2.3 in various respects. Differently from [Benerecetti et al., 1998] the methodology presented here is *computationally grounded*, in the sense of [Wooldridge, 2000a]. Also, instead of relying on existing model checkers as in [Wooldridge et al., 2002] and [Bordini et al., 2003b, Bordini et al., 2003a], Chapter 3 introduced a self-contained methodology, which avoids the translation into temporal-only model checkers.

The algorithms and the implementation presented in this thesis use OBDDs and, in this respect, they differ from all the SAT-based approaches presented in Section 2.3 (Section 6.5.5 has investigated the differences between the two techniques in more detail, using experimental results).

Although [Gammie and van der Meyden, 2004] use OBDDs, they restrict the verification to a particular class of interpreted systems, and do not consider a number of operators that are available in MCMAS; these include operators to reason about distributed knowledge, common knowledge, "correct behaviour" and strategies of agents. Moreover, as mentioned in Section 6.5.5, the input language of MCK and MCMAS are substantially different, and they are suited for different classes of examples.

7.3 Future work

This thesis has shown that model checking for multi-agent systems introduces a number of benefits with respect to temporal-only model checking. Nevertheless, some issues need to be addressed before model checking for multi-agent systems reaches the maturity of traditional model checking. In particular, open issues not considered in this thesis include:

- Correspondence between actual system and the input of model checkers: this problem arises for both traditional and MAS model checkers, when large examples have to be encoded using a manual translation. Even if *computationally grounded* theories of agency [Wooldridge, 2000a] (and the corresponding verification methodologies) are a necessary condition for the verification of such examples, the correspondence between an actual system and its representation requires care. In the case of model checkers for MAS a further issue arises when encoding a scenario using agents: what is an agent? The example presented in Section 6.4 is a concrete instance of this issue: Livingstone models are composed of so called "modules" (for instance, a circuit breaker is a module). But are all modules agents? Is a circuit breaker an agent? This kind of correspondence is an open question.

- In many circumstances, understanding why a formula is false in a model may be as useful as knowing that a formula is true in a model. Counter-examples can be generated automatically by some temporal model checkers[1]; unfortunately, even in the case of temporal-only model checkers, counter-examples are typically difficult to understand. Counter-examples may become even more complex after introducing non-temporal modalities. Thus, the effective generation of human-readable counter-examples for multi-agent systems needs to be addressed with care in a mature model checker for multi-agents systems. In parallel with this issue, fairness conditions [Clarke et al., 1999] need to be introduced to avoid obviously un-wanted behaviours of agents.

- Deontic interpreted systems offer a computationally grounded, fine grain semantics for multi-agent systems. In certain cases this fine grain semantics could be refined further by introducing variables to describe the local states of the agents. This is the case, for instance, with the example in Section 6.4: the diagnoser could be described more easily using one variable for each observable. This change would require a complete redefinition of the protocols, the evolution and the evaluation functions, but it would make deontic interpreted systems an even more "computationally grounded" semantics for multi agent systems.

- Optimisation techniques usually included in temporal model checkers have not been introduced in MCMAS. Techniques that could be implemented in MCMAS include

[1]Model checkers for multi-agent systems do not support this feature yet.

abstraction, on-the-fly model checking, caching, and strategies for reordering OBDD variables.

- A graphical interface and an on-line version of MCMAS are currently under development, with the aim increasing the number of potential MCMAS users.

.

Bibliography

[Agotnes, 2005] Agotnes, T. (2005). Action and knowledge in Alternating-time Temporal Logic. *Synthese*. Special issue on Knowledge, Rationality and Action.

[Alur and Dill, 1994] Alur, R. and Dill, D. (1994). A theory of Timed Automata. *Theoretical Computer Science*, 126(2):183–235.

[Alur et al., 1998] Alur, R., Henzinger, T., Mang, F., Qadeer, S., Rajamani, S., and Tasiran, S. (1998). MOCHA: Modularity in model checking. In *Proceedings of the 10th International Conference on Computer Aided Verification (CAV'98)*, volume 1427 of *LNCS*, pages 521–525. Springer-Verlag.

[Alur et al., 2006] Alur, R., Henzinger, T., Mang, F., Qadeer, S., Rajamani, S., and Tasiran, S. (2006). `http://embedded.eecs.berkeley.edu/research/mocha/`.

[Alur et al., 1997] Alur, R., Henzinger, T. A., and Kupferman, O. (1997). Alternating-time temporal logic. In *Proceedings of the 38th IEEE Symposium on Foundations of Computer Science (FOCS'97)*, pages 100–109. IEEE Computer Society.

[Alur et al., 2002] Alur, R., Henzinger, T. A., and Kupferman, O. (2002). Alternating-time temporal logic. *Journal of the ACM*, 49(5):672–713.

[Anderson, 1958] Anderson, A. R. (1958). A reduction of deontic logic to alethic modal logic. *Mind*, 58:100–103.

[Beizer, 1990] Beizer, B. (1990). *Software Testing Techniques*. Van Nostrand Reinhold, New York, 2nd edition.

[Benerecetti et al., 1998] Benerecetti, M., Giunchiglia, F., and Serafini, L. (1998). Model checking multiagent systems. *Journal of Logic and Computation*, 8(3):401–423.

[Bengtsson et al., 1998] Bengtsson, J., Larsen, K., Larsson, F., Pettersson, P., Yi, W., and Weise, C. (1998). New generation of Uppaal. In *Proceedings of the International Workshop on Software Tools for Technology Transfer*.

[Beyer et al., 2003] Beyer, D., Lewerentz, C., and Noack, A. (2003). Rabbit: A tool for BDD-based verification of real-time systems. In Hunt, W. A. and Somenzi, F., editors, *Proceedings of the 15th International Conference on Computer Aided Verification (CAV 2003)*, LNCS 2725, pages 122–125. Springer-Verlag, Berlin.

[Biere et al., 1999a] Biere, A., Cimatti, A., Clarke, E., and Zhu, Y. (1999a). Symbolic model checking without BDDs. In *Proc. of TACAS'99*, volume 1579 of *LNCS*, pages 193–207. Springer-Verlag.

[Biere et al., 1999b] Biere, A., Clarke, E., Raimi, R., and Zhu, Y. (1999b). Verifying safety properties of a PowerPC microprocessor using symbolic model checking without BDDs. In *Proceedings of the 11th International Conference on Computer Aided Verification (CAV'99)*, volume 1633 of *LNCS*, pages 60–71. Springer-Verlag.

[Blackburn et al., 2001] Blackburn, P., de Rijke, M., and Venema, Y. (2001). *Modal Logic*, volume 53 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press.

[Bordini et al., 2003a] Bordini, R., Fisher, M., Pardavila, C., Visser, W., and Wooldridge, M. (2003a). Model checking multi-agent programs with CASP. In *Proceedings of the 15th International Conference on Computer Aided Verification (CAV'03)*, volume 2725 of *LNCS*, pages 110–113. Springer-Verlag.

[Bordini et al., 2003b] Bordini, R. H., Fisher, M., Pardavila, C., and Wooldridge, M. (2003b). Model checking AgentSpeak. In Rosenschein, J. S., Sandholm, T., Michael, W., and Yokoo, M., editors, *Proceedings of the Second International Joint Conference on Autonomous Agents and Multi-agent systems (AAMAS-03)*, pages 409–416. ACM Press.

[Brat et al., 2000] Brat, G., Havelund, K., Park, S., and Visser, W. (2000). Model checking programs. In *Proceedings of the 15th International Conference on Automated Software Engineering (ASE-2000)*, pages 3–12. IEEE Computer Society.

[Brayton et al., 1996] Brayton, S., Hachtel, G., Sangiovanni-Vincentelli, A., Somenzi, F., Aziz, A., Cheng, S., Edwards, S., Khatri, S., Kukimoto, Y., Pardo, A., Qadeer, S., Ranjan, R., Sarwary, S., Shiple, T., Swamy, G., and Villa, T. (1996). VIS: A system for verification and synthesis. In *Proceedings of the 8th International Conference on Computer Aided Verification (CAV'96)*, volume 1102 of *LNCS*, pages 428–432. Springer-Verlag.

[Bryant, 1986] Bryant, R. E. (1986). Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691.

[Bryant, 1991] Bryant, R. E. (1991). On the complexity of VLSI implementations and graph representations of boolean functions with application to integer multiplication. *IEEE Trans. Comput.*, 40(2):205–213.

[Burch et al., 1992] Burch, J. R., Clarke, E. M., McMillan, K. L., Dill, D. L., and Hwang, L. J. (1992). Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170.

[Chaum, 1988] Chaum, D. (1988). The dining cryptographers problem: Unconditional sender and recipient untraceability. *Journal of Cryptology*, 1(1):65–75.

[Chellas, 1980] Chellas, B. (1980). *Modal Logic: An Introduction.* Cambridge University Press, Cambridge.

[Cheng, 1995] Cheng, A. (1995). Complexity results for model checking. Technical Report RS-95-18, BRICS - Basic Research in Computer Science, Department of Computer Science, University of Aarhus.

[Cimatti et al., 2002] Cimatti, A., Clarke, E. M., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., and Tacchella, A. (2002). NuSMV2: An open-source tool for symbolic model checking. In *Proceedings of the 14th International Conference on Computer Aided Verification (CAV'02)*, volume 2404 of *LNCS*, pages 359–364. Springer-Verlag.

[Cimatti et al., 2003] Cimatti, A., Pecheur, C., and Cavada, R. (2003). Formal verification of diagnosability via symbolic model checking. In *Proceedings of IJCAI03*, volume 1871 of *LNCS*, pages 363–369. Springer Verlag.

[Clarke and Emerson, 1981] Clarke, E. and Emerson, E. (1981). Design and synthesis of synchronization skeletons for branching-time temporal logic. In *Proceedings of Workshop on Logic of Programs*, volume 131 of *LNCS*, pages 52–71. Springer-Verlag.

[Clarke et al., 2004a] Clarke, E., Kroening, D., and Lerda, F. (2004a). A tool for checking ANSI-C programs. In Jensen, K. and Podelski, A., editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer.

[Clarke et al., 2006] Clarke, E., Kroening, D., and Lerda, F. (2006). `http://www.cs.cmu.edu/~modelcheck/cbmc/`.

[Clarke et al., 2004b] Clarke, E., Kroening, D., Strichman, O., and Ouaknine, J. (2004b). Completeness and complexity of bounded model checking. In *5th International Conference on Verification, Model Checking, and Abstract Interpretation*, volume 2937 of *Lecture Notes in Computer Science*, pages 85–96.

[Clarke et al., 1986] Clarke, E. M., Emerson, E. A., and Sistla, A. P. (1986). Automatic verification of finite-state concurrent systems using temporal logic specifications: A practical approach. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263.

[Clarke et al., 1999] Clarke, E. M., Grumberg, O., and Peled, D. A. (1999). *Model Checking*. The MIT Press, Cambridge, Massachusetts.

[Cohen and Levesque, 1990] Cohen, P. R. and Levesque, H. J. (1990). Intention is choice with commitment. *Artificial Intelligence*, 42(2-3):213–261.

[Cox et al., 2005] Cox, J., Bartold, T., and Durfee, E. (2005). A distributed framework for solving the multiagent plan coordination problem. In *Proceedings of the Fourth International Conference on Autonomous Agents and Multiagent Systems (AAMAS'05)*, pages 821–827.

[Daws et al., 1995] Daws, C., Olivero, A., Tripakis, S., and Yovine, S. (1995). The tool KRONOS. In *Hybrid Systems III*, volume 1066 of *LNCS*, pages 208–219. Springer-Verlag.

[Dennett, 1987] Dennett, D. (1987). *The Intentional Stance*. MIT Press.

[Ditmarsch et al., 2005] Ditmarsch, H., Hoek, W., Meyden, R., and Ruan, J. (2005). Model checking russian cards. In *Proceedings of Mochart — Third International Workshop on Model Checking and Artificial Intelligence*, volume 149(2). Electronic Notes in Theoretical Computer Science.

[Ditmarsch et al., 2006] Ditmarsch, H., Hoek, W., Meyden, R., and Ruan, J. (2006). `http://homepages.cwi.nl/~jve/papers/04/demo/`.

[Dwyer et al., 1998] Dwyer, M. B., Avrunin, G. S., and Corbett, J. C. (1998). Property specification patterns for finite-state verification. In Ardis, M., editor, *Proceedings of the 2nd Workshop on Formal Methods in Software Practice (FMSP'98)*, pages 7–15, New York. ACM Press.

[Dwyer et al., 2006] Dwyer, M. B., Avrunin, G. S., and Corbett, J. C. (2006). `http://patterns.projects.cis.ksu.edu/`.

[Emerson, 1990] Emerson, E. A. (1990). Temporal and modal logic. In van Leeuwen, J., editor, *Handbook of Theoretical Computer Science*, pages 996–1071. Elsevier Science Publishers.

[Emerson and Halpern, 1985] Emerson, E. A. and Halpern, J. Y. (1985). Decision procedures and expressiveness in the temporal logic of branching time. *Journal of Computer and System Sciences*, 30(1):1–24.

[Fagin et al., 1995] Fagin, R., Halpern, J. Y., Moses, Y., and Vardi, M. Y. (1995). *Reasoning about Knowledge*. MIT Press, Cambridge.

[Franceschet et al., 2004] Franceschet, M., Montanari, A., and de Rijke, M. (2004). Model checking for combined logics with an application to mobile systems. *Automated Software Engineering*, 11:289–321.

[Gabbay et al., 2003] Gabbay, D., Kurucz, A., Wolter, F., and Zakharyaschev, M. (2003). *Many-Dimensional Modal Logics: Theory and Applications*, volume 148 of *Studies in Logic*. Elsevier.

[Gammie and van der Meyden, 2004] Gammie, P. and van der Meyden, R. (2004). MCK: Model checking the logic of knowledge. In *Proceedings of 16th International Conference on Computer Aided Verification (CAV'04)*, volume 3114 of *LNCS*, pages 479–483. Springer-Verlag.

[Goldblatt, 1992] Goldblatt, R. (1992). *Logics of Time and Computation, Second Edition, Revised and Expanded*, volume 7 of *CSLI Lecture Notes*. CSLI, Stanford. Distributed by University of Chicago Press.

[Goldschlag et al., 1999] Goldschlag, D., Reed, M., and Syverson, P. (1999). Onion routing. *Communications of the ACM*, 42(2):39–41.

[Goranko and Jamroga, 2004] Goranko, V. and Jamroga, W. (2004). Comparing semantics for logics of multi-agent systems. *Synthese*, 139(2):241–280.

[Harel, 1984] Harel, D. (1984). Dynamic logic. In Gabbay, D. and Guenthner, F., editors, *Handbook of Philosophical Logic, Volume II: Extensions of Classical Logic*, volume 165 of *Synthese Library*, pages 497–604. D. Reidel, Dordrecht.

[Hendrik, 2006] Hendrik, L. (2006). `http://staff.science.uva.nl/~lhendrik/`.

[Henzinger et al., 2003] Henzinger, T. A., Jhala, R., Majumdar, R., , and Sutre, G. (2003). Software verification with blast. In *Proceedings of the Tenth International Workshop on Model Checking of Software (SPIN)*, pages 235–239. Lecture Notes in Computer Science 2648, Springer-Verlag.

[Hoek and Wooldridge, 2003a] Hoek, W. and Wooldridge, M. (2003a). Cooperation, knowledge, and time: Alternating-time temporal epistemic logic and its applications. *Studia Logica*, 75(1):125–157.

[Hoek and Wooldridge, 2003b] Hoek, W. and Wooldridge, M. (2003b). Towards a logic of rational agency. *Logic Journal of the IGPL*, 11(2):135–159.

[Hoek et al., 2006] Hoek, W. v., Lomuscio, A., and Wooldridge, M. (2006). On the complexity of practical atl 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)*, Hakodake, Japan. ACM Press. To appear.

[Hoek and Wooldridge, 2002] Hoek, W. v. and Wooldridge, M. (2002). Model checking knowledge and time. In *SPIN 2002 – Proceedings of the Ninth International SPIN Workshop on Model Checking of Software*, Grenoble, France.

[Holzmann, 1997] Holzmann, G. J. (1997). The model checker SPIN. *IEEE transaction on software engineering*, 23(5):279–295.

[Holzmann, 2003] Holzmann, G. J. (2003). *SPIN Model Checker, The: Primer and Reference Manual*. Addison Wesley Professional.

[Hughes and Cresswell, 1996] Hughes, G. E. and Cresswell, M. J. (1996). *A New Introduction to Modal Logic*. Routledge, New York.

[Huth and Ryan, 2004] Huth, M. R. A. and Ryan, M. D. (2004). *Logic in Computer Science: Modelling and Reasoning about Systems (2nd edition)*. Cambridge University Press, Cambridge, England.

[Jamroga, 2004a] Jamroga, W. (2004a). Some remarks on alternating temporal epistemic logic. In Dunin-Kęplicz, B. and Verbrugge, R., editors, *Proceedings of the International Workshop on Formal Approaches to Multi-Agent Systems (FAMAS'03)*, pages 133–140.

[Jamroga, 2004b] Jamroga, W. (2004b). *Using Multiple Models of Reality. On Agents who Know how to Play Safer.* PhD thesis, University of Twente, Enschede, The Netherlands.

[Jamroga and van der Hoek, 2004] Jamroga, W. and van der Hoek, W. (2004). Agents that know how to play. *Fundamenta Informaticae*, 62:1–35.

[Jonker, 2003] Jonker, G. (2003). Feasible strategies in alternating-time temporal epistemic logic. Master's thesis, University of Utrech, The Netherlands.

[Kacprzak et al., 2006] Kacprzak, M., Lomuscio, A., Niewiadomski, A., Penczek, W., Raimondi, F., and Szreter, M. (2006). Comparing BDD and SAT based techniques for model checking Chaum's dining cryptographers protocol. *Fundamenta Informaticae*. to appear.

[Kitano, 1998] Kitano, H., editor (1998). *RoboCup-97: Robot Soccer World Cup I*, volume 1395 of *LNCS*. Springer-Verlag.

[Kozen, 1983] Kozen, D. (1983). Results on the propositional mu-calculus. *Theoretical Computer Science*, 27:333–354.

[Kupferman et al., 2000] Kupferman, O., Vardi, M. Y., and Wolper, P. (2000). An automata-theoretic approach to branching-time model checking. *Journal of the ACM*, 47(2):312–360.

[Lomuscio and Raimondi, 2006a] Lomuscio, A. and Raimondi, F. (2006a). The complexity of model checking concurrent programs against CTLK specifications. In *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems (AAMAS'06)*, pages 548–550, Hakodake, Japan. ACM Press.

[Lomuscio and Raimondi, 2006b] Lomuscio, A. and Raimondi, F. (2006b). MCMAS: A model checker for multi-agent systems. In Hermanns, H. and Palsberg, J., editors, *Proceedings of TACAS 2006, Vienna*, volume 3920, pages 450–454. Springer Verlag.

[Lomuscio and Raimondi, 2006c] Lomuscio, A. and Raimondi, F. (2006c). 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, Hakodake, Japan. ACM Press.

[Lomuscio et al., 2003] Lomuscio, A., Raimondi, F., and Sergot, M. J. (2003). Towards model checking interpreted systems. In *Proceedings of 2nd International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS'03)*, pages 1054–1055, Melbourne. ACM Press.

[Lomuscio and Sergot, 2003] Lomuscio, A. and Sergot, M. (2003). Deontic interpreted systems. *Studia Logica*, 75(1):63–92.

[Lomuscio and Sergot, 2004] Lomuscio, A. and Sergot, M. (2004). A formalisation of violation, error recovery, and enforcement in the bit transmission problem. *Journal of Applied Logic*, 2(1):93–116.

[Marrero et al., 1997] Marrero, W., Clarke, E., and Jha, S. (1997). Model checking for security protocols. Technical Report CMU-CS-97-139, Carnegie Mellon University.

[McCarthy, 1979] McCarthy, J. (1979). Ascribing mental qualities to machines. In Ringle, M., editor, *Philosophical Perspectives in Artificial Intelligence*. Harvester Press.

[McMillan, 1992] McMillan, K. (1992). The SMV system. Technical Report CMU-CS-92-131, Carnegie-Mellon University.

[McMillan, 1993] McMillan, K. L. (1993). *Symbolic Model Checking*. Kluwer Academic Publishers.

[McMillan, 2002] McMillan, K. L. (2002). Applying SAT methods in unbounded symbolic model checking. In *Proc. of the 14th Int. Conf. on Computer Aided Verification (CAV'02)*, volume 2404 of *LNCS*, pages 250–264. Springer-Verlag.

[Meyden, 1998] Meyden, R. (1998). Common knowledge and update in finite environments. *Information and Computation*, 140(2):115–157.

[Meyden and Shilov, 1999] Meyden, R. v. and Shilov, H. (1999). Model checking knowledge and time in systems with perfect recall. In *Proceedings of Proc. of FST&TCS*, volume 1738 of *Lecture Notes in Computer Science*, pages 432–445, Hyderabad, India.

[Meyden and Wong, 2003] Meyden, R. v. and Wong, K. (2003). Complete axiomatizations for reasoning about knowledge and branching time. *Studia Logica*, 75(1):93–123.

[Moore, 1990] Moore, R. C. (1990). A formal theory of knowledge and action. In Allen, J., Hendler, J., and Tate, A., editors, *Readings in Planning*, pages 480–519. Kaufmann, San Mateo, CA.

[Moura et al., 2004] Moura, L., Owre, S., Rueß, H., Rushby, J., Shankar, N., Sorea, M., and Tiwari, A. (2004). SAL 2. In Alur, R. and Peled, D., editors, *Computer-Aided Verification, CAV 2004*, volume 3114 of *Lecture Notes in Computer Science*, pages 496–500, Boston, MA. Springer-Verlag.

[Moura et al., 2006] Moura, L., Owre, S., Rueß, H., Rushby, J., Shankar, N., Sorea, M., and Tiwari, A. (2006). `http://sal.csl.sri.com`.

[Muscettola et al., 1998] Muscettola, N., Nayak, P. P., Pell, B., and Williams, B. C. (1998). Remote agent: to go boldly where no AI system has gone before. *Artificial Intelligence*, 103(1–2):5–47.

[Nabialek et al., 2004] Nabialek, W., Niewiadomski, A., Penczek, W., Pólrola, A., and Szreter, M. (2004). VerICS 2004: A model checker for real time and multi-agent systems. In *Proceedings of the International Workshop on Concurrency, Specification and Programming (CS&P'04)*, volume 170 of *Informatik-Berichte*, pages 88–99. Humboldt University.

[Otterloo et al., 2003] Otterloo, S., van der Hoek, W., and Wooldridge, M. (2003). Knowledge as strategic ability. *Electronic Notes in Theoretical Computer Science*, 85(2):1–23.

[Papadimitriou, 1994] Papadimitriou, C. H. (1994). *Computational Complexity.* Addison-Wesley.

[Pecheur and Simmons, 2000] Pecheur, C. and Simmons, R. (2000). From Livingstone to SMV: Formal verification of autonomous spacecrafts. In *Proceedings of the First Goddard Workshop on Formal Approaches to Agent-Based Systems (FAABS)*, volume 1871 of *Lecture Notes in Computer Science*, pages 103–113. Springer Verlag.

[Penczek and Lomuscio, 2003] Penczek, W. and Lomuscio, A. (2003). Verifying epistemic properties of multi-agent systems via bounded model checking. *Fundamenta Informaticae*, 55(2):167–185.

[Penczek et al., 2004] Penczek, W., Szreter, M., Woźna, B., and Zbrzezny, A. (2004). SAT-based unbounded model checking for TCTL. Technical report, ICS PAS, Ordona 21, 01-237 Warsaw. To appear.

[Penczek et al., 2002] Penczek, W., Woźna, B., and Zbrzezny, A. (2002). Bounded model checking for the universal fragment of CTL. *Fundamenta Informaticae*, 51(1-2):135–156.

[Pettersson and Larsen., 2000] Pettersson, P. and Larsen., K. G. (2000). Uppaal2k. *Bulletin of the European Association for Theoretical Computer Science*, 70:40–44.

[Pnueli, 1981] Pnueli, A. (1981). The temporal semantics of concurrent programs. In *Theoretical Computer Science*, volume 13, pages 1–20. Elsevier Science Publishers.

[Pucella, 2005] Pucella, R. (2005). Logic column 11: The finite and the infinite in temporal logic. *SIGACT NEWS*, 36:86.

[Quielle and Sifakis, 1981] Quielle, J. P. and Sifakis, J. (1981). Specification and verification of concurrent systems in CESAR. In *Proceedings of the 5th International Symposium on Programming*, volume 131 of *LNCS*, pages 337–351. Springer-Verlag.

[Raimondi and Lomuscio, 2004a] Raimondi, F. and Lomuscio, A. (2004a). Automatic verification of deontic interpreted systems by model checking via OBDDs. In *Proceedings of the Sixteenth European Conference on Artificial Intelligence (ECAI04)*, pages 53–57. IOS PRESS.

[Raimondi and Lomuscio, 2004b] Raimondi, F. and Lomuscio, A. (2004b). Symbolic model checking of deontic interpreted systems via OBDDs. In *Proceedings of DEON04, Seventh International Workshop on Deontic Logic in Computer Science*, volume 3065 of *Lecture Notes in Computer Science*, pages 228–242. Springer Verlag.

[Raimondi and Lomuscio, 2004c] Raimondi, F. and Lomuscio, A. (2004c). Towards model checking for multiagent systems via OBDDs. In *Proceedings of the Third NASA Workshop on Formal Approaches to Agent-Based Systems (FAABS III)*, volume 3224 of *Lecture Notes in Computer Science*, pages 213–221. Springer Verlag.

[Raimondi and Lomuscio, 2004d] Raimondi, F. and Lomuscio, A. (2004d). Verification of multiagent systems via ordered binary decision diagrams: an algorithm and its implementation. In Jennings, N. R., Sierra, C., Sonenberg, L., and Tambe, M., editors, *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'04)*, volume II, pages 630–637. ACM.

[Raimondi and Lomuscio, 2005a] Raimondi, F. and Lomuscio, A. (2005a). Automatic verification of multi-agent systems by model checking via OBDDs. *Journal of Applied Logic.* To appear in Special issue on Logic-based agent verification.

[Raimondi and Lomuscio, 2005b] Raimondi, F. and Lomuscio, A. (2005b). The complexity of symbolic model checking temporal-epistemic logics. In *Proceedings of Concurrency, Specification & Programming (CS&P)*, pages 421–432. Warsaw University.

[Raimondi and Lomuscio, 2005c] Raimondi, F. and Lomuscio, A. (2005c). Model checking knowledge, strategies, and games in multi-agent systems. Technical Report RN/05/01, Department of Computer Science, UCL, London, UK.

[Raimondi and Lomuscio, 2006] Raimondi, F. and Lomuscio, A. (2006). `http://www.cs.ucl.ac.uk/staff/f.raimondi/MCMAS/`.

[Raimondi et al., 2005] Raimondi, F., Pecheur, C., and Lomuscio, A. (2005). Applications of model checking for multi-agent systems: verification of diagnosability and recoverability. In *Proceedings of Concurrency, Specification & Programming (CS&P)*, pages 433–444. Warsaw University.

[S. W. Squyres et al., 2004a] S. W. Squyres et al. (2004a). The Opportunity Rover's Athena Science Investigation at Meridiani Planum, Mars. *Science*, pages 1698–1703.

[S. W. Squyres et al., 2004b] S. W. Squyres et al. (2004b). The Spirit Rover's Athena Science Investigation at Gusev Crater, Mars. *Science*, pages 1698–1703.

[Sampath et al., 1995] Sampath, M., Sengupta, R., Lafortune, S., Sinnamohideen, K., and Teneketzis, D. (1995). Diagnosability of discrete-event systems. *IEEE Transactions on Automatic Control*, 40(9):1555–1575.

[Schnoebelen, 2003] Schnoebelen, P. (2003). The complexity of temporal logic model checking. In *Proceedings of the 4th Conference Advances in Modal Logic (AiML'2002)*, volume 4 of *Advances in Modal Logic*, pages 437–459. King's College Publications.

[Sistla and Clarke, 1985] Sistla, A. P. and Clarke, E. M. (1985). The complexity of propositional linear temporal logic. *Journal of the ACM*, 32(3):733–749.

[Somenzi, 2005] Somenzi, F. (2005). CUDD: CU decision diagram package - release 2.4.0. `http://vlsi.colorado.edu/~fabio/CUDD/cuddIntro.html`.

[Tarski, 1955] Tarski, A. (1955). A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309.

[van der Meyden and Su, 2004] van der Meyden, R. and Su, K. (2004). Symbolic model checking the knowledge of the dining cryptographers. In *Proceedings of the 17th IEEE Computer Security Foundations Workshop (CSFW'04)*, pages 280–291, Washington, DC, USA. IEEE Computer Society.

[van Drimmelen, 2003] van Drimmelen, G. (2003). Satisfiability in alternating-time temporal logic. In *LICS '03: Proceedings of the 18th Annual IEEE Symposium on Logic in Computer Science*, pages 208–213, Washington, DC, USA. IEEE Computer Society.

[Vardi and Wolper, 1986] Vardi, M. Y. and Wolper, P. (1986). An automata-theoretic approach to automatic program verification. In *Proc. 1st Symp. on Logic in Computer Science*, pages 332–344, Cambridge.

[Wooldridge, 2000a] Wooldridge, M. (2000a). Computationally grounded theories of agency. In Durfee, E., editor, *Proceedings of ICMAS, International Conference of Multi-Agent Systems*, pages 13–22. IEEE Press.

[Wooldridge, 2000b] Wooldridge, M. (2000b). *Reasoning about Rational Agents.* MIT Press.

[Wooldridge, 2002] Wooldridge, M. (2002). *An introduction to multi-agent systems.* John Wiley, England.

[Wooldridge et al., 2002] Wooldridge, M., Fisher, M., Huget, M., and Parsons, S. (2002). Model checking multiagent systems with MABLE. In *Proceedings of the First International Conference on Autonomous Agents and Multiagent Systems (AAMAS-02)*, pages 952–959, Bologna, Italy.

[Wooldridge and Jennings, 1995] Wooldridge, M. and Jennings, N. R. (1995). Intelligent agents: theory and practice. *Knowledge Engineering Review*, 2(10):115–152.

[Wooldridge and Lomuscio, 2000] Wooldridge, M. and Lomuscio, A. (2000). Multi-agent $\mathcal{VSK}$ logic. In Ojeda-Aciego, M., de Guzmán, I. P., Brewka, G., and Pereira, L. M., editors, *Logics in Artificial Intelligence — Proceedings of the Seventh European Workshop, JELIA 2000 (LNAI Volume 1919)*, pages 300–312. Springer-Verlag.

[Woźna et al., 2004] Woźna, B., Lomuscio, A., and Penczek, W. (2004). Bounded model checking for deontic interpreted systems. In *Proc. of the 2nd Workshop on Logic and Communication in Multi-Agent Systems (LCMAS'04)*, volume 126 of *Electronic Notes in Theoretical Computer Science*, pages 93–114. Elsevier.

[Woźna et al., 2005] Woźna, B., Lomuscio, A., and Penczek, W. (2005). Bounded model checking for knowledge over real time. In *Proceedings of the 4st International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS'05)*, volume I, pages 165–172. ACM Press.

[Woźna and Zbrzezny, 2005] Woźna, B. and Zbrzezny, A. (2005). Bounded Model Checking for the existential fragment of TCTL and Diagonal Timed Automata. In Czaja, L., editor, *Proceedings of the International Workshop on Concurrency, Specification and Programming (CS&P'05)*, pages 586–597. Warsaw University.

[Yovine, 1997] Yovine, S. (1997). KRONOS: A verification tool for real-time systems. *Springer International Journal of Software Tools for Technology Transfer*, 1(1-2):123–133.