VERIFYING REQUIREMENTS FOR RESOURCE-BOUNDED AGENTS

ABDUR RAKIB
M.TECH. (IIT KHARAGPUR, INDIA)

A THESIS SUBMITTED TO THE UNIVERSITY OF NOTTINGHAM
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
IN THE SCHOOL OF COMPUTER SCIENCE

JUNE 2011

Abstract

This thesis presents frameworks for the modelling and verification of resource-bounded reasoning agents. The resources considered include the time, memory, and communication bandwidth required by agents to achieve a goal. The scalability and expressiveness of standard model checking techniques is investigated using two typical multi-agent reasoning problems which can be easily parameterised to increase or decrease the problem size. Both a complexity analysis and experimental results suggest that reasonably sized problem instances are unlikely to be tractable for a standard model checker without steps to reduce the branching factor of the state space. We propose two approaches to address this problem: the use of abstract specifications to model the behaviour of some of the agents in the system, and exploiting information about the reasoning strategy adopted by the agents. Abstract specifications are given as Linear Temporal Logic (LTL) formulae which describe the external behaviour of the agents, allowing their temporal behaviour to be compactly modelled. Conversely, reasoning strategies allow the detailed specification of the ordering of steps in the agent's reasoning process. Both approaches have been combined in an automated verification tool TVRBA for rule-based multi-agent systems which allows the designer to specify information about agents' interaction, behaviour, and execution strategy at different levels of abstraction. The TVRBA tool generates an encoding of the system for the Maude LTL model checker, allowing properties of the system to be verified. The scalability of the new approach is illustrated using three case studies.

Acknowledgement

First and foremost I would like to thank my supervisors Brian Logan and Natasha Alechina for their wonderful support all throughout my stay at Nottingham. Under Brian's guidance, I have learnt everything I know about agents and formal verification. He initiated me into the world of research by turning my attention towards various interesting problems including those studied in this thesis. He then taught me, listened to me, trusted me, and guided me with unwavering enthusiasm and patience and steered me towards solutions. Brian and Natasha have maintained a great research atmosphere in their Agents Lab, and have been an inspiration and constant source of encouragement for us in terms of vision and ideas in research. My second biggest fortune was in having good colleagues in Agents Lab during these years. I would like to thank them all for the fruitful discussions about research and so many other aspects of my life.

I would like to thank Henrik Nilsson, my internal assessor, for his show of support and encouragement and comments on parts of the thesis, and to Michael Fisher for accepting to act as my external examiner. I would like to thank those people who have given courses on formal verification and everyone with whom I have had interactions during my attendance at ESSLLI'07&08 and EASSS'07&08—both research and otherwise.

I gratefully acknowledge the financial support of the UK Engineering and Physical Sciences Research Council (EPSRC Reference: EP/E031226/1) and the School of Computer Science at the University of Nottingham. I would like to express my greatest appreciation to all the members of the School of Computer Science at the University of Nottingham and my friends at Nottingham, including Syed Iqbal Anwar, for their direct or indirect help and cooperation in completing a great task.

I would like to thank my parents, my sisters and brothers, for their love and encouragement. A very special thanks to my wife for her love, her patience and understanding. She supported me in every possible way, and was always there to listen and help.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Distributed problem solving is an emerging research area that combines aspects of artificial intelligence (AI) and distributed processing. One of the primary focuses of this approach is the study of co-operative activity in systems composed of multiple interacting intelligent agents. Such systems are known as multi-agent systems (MAS). When solving problems, each intelligent agent in the system requires some basic resources such as *time* (number of computational steps), *space* (amount of memory) and perhaps *communication bandwidth* (number of messages that need to be exchanged). The trend towards ever smaller agent platforms means that resource utilisation is becoming an increasingly important factor in agent design and deployment. However the complex, often distributed, derivations implied by modern agent designs make it hard for agent developers to predict system resource requirements a priori. The development of formal frameworks and practical verification tools to exploit them is therefore key to the successful development of provably correct agent designs for emerging resource-limited agent paradigms including, for example, agent-based sensor networks. The aim of this thesis is to present frameworks for explicit modelling of system resources, specifying and ultimately verifying the properties of resource-bounded multi-agent systems. In this chapter we discuss the motivations for developing such techniques and tools, followed by a brief outline of existing approaches to formal verification. The problem statement and methodology of the thesis are then described. Finally, the contributions

and the structure of the remainder of the thesis is outlined.

1.1 Motivation

In recent years intelligent agents have been the focus of much attention from the AI community. In AI research, agent-based systems technology has emerged as a new paradigm for conceptualizing, designing, and implementing sophisticated software systems. In general, multi-agent systems research refers to software agents. However, the agents in a multi-agent system could also be for example robots. The concept of agents, in the context of this thesis is used to refer to autonomous reasoning agents, where agents are capable of reasoning about their behaviour (using a knowledge base and inference rules) and interactions (capable of communicating with each other). That is agents are primarily viewed as doing some kind of inference over a knowledge base, e.g., using resolution or forward chaining rules (modus ponens). An agent is autonomous if it encapsulates its behaviour and internal state [Jennings and Wooldridge, 1998]. This means that an agent itself has control over its own actions and behaviour. Intelligent agents are being used in wide variety of applications that include small systems like email filtering and prioritizing [Boone, 1998], wireless sensor network technology [Tynan et al., 2005, Platt et al., 2008], and safety-critical systems [Callantine, 2002, 2003] to e-commerce applications [Sun and Finnie, 2004].

While agents provide great benefits in developing many complex software applications (e.g., systems that have multiple components, distributed over networks, exhibit dynamic changes, and require autonomous behaviour [Wooldridge, 2009]), they also present new challenges to application developers, namely how to ensure the *correctness* of system designs (will a system behave expectedly for all possible legal inputs), *termination* (will a system produce an output at all), and *response time* (how much computation will a system have to do before it generates an output). These problems become even more challenging in the case of multi-agent systems, where agents exchange information via messages. Therefore, when a number of autonomous agents

interact it is very difficult to predict the behaviour of the system and guarantee that the desired functionalities will be fulfilled. Consequently, the systems must be verified to show that they are correct with respect to their specifications.

The analysis and verification of MAS is not an easy task due to their dynamic nature, and the complex interactions between agents [Bordini et al., 2007a]. Nevertheless, there has been interest in using formal methods to specify and verify agent-based systems [Fisher et al., 2007]. For the last few years, Lomuscio and colleagues [Raimondi and Lomuscio, 2007, Lomuscio and Penczek, 2007] have been working on automatic verification of multi-agent systems. The outcome of their research includes developing MCMAS [Lomuscio et al., 2009] a model based verification tool. A strand of work on model-checking properties of agent programming languages is carried out by various researchers, including those presented in [Wooldridge et al., 2002, Bordini et al., 2003, 2004, 2006, Dennis et al., 2008a]. Some other significant works on modelling and verifying multi-agent systems include [Rao and Georgeff, 1993, Shoham, 1993, Fisher and Wooldridge, 1997, Benerecetti et al., 1998, de Giacomo et al., 2000, van der Hoek and Wooldridge, 2002]. However, all these works are based on the classical approach of knowledge representation, they do not model resources such as time, space and communication restrictions on the agent's ability to derive consequences of its beliefs.

There is a growing body of work where the agent's deduction steps are explicitly modelled in the logical framework, for example [Duc, 1997, Alechina et al., 2004], which makes it possible to model the time it takes the agent to arrive at a certain conclusion; a different kind of limitation on the depth of belief reasoning allowed is studied in [Fisher and Ghidini, 1999]. Both the time and space limitations on the agent's knowledge were considered in step logics [Elgot-Drapkin et al., 1991], the framework however does not support verification of space requirements for solving a certain problem. In more recent work [Albore et al., 2006, Alechina et al., 2006, 2007] Alechina and colleagues have taken some preliminary steps towards the automated verification

of resource requirements of reasoning agents. In [Albore et al., 2006, Alechina et al., 2007] Alechina and colleagues have considered only single agents (and no communication costs), which reason using relatively simple logical formalisms, and have mostly focused on a single resource (memory). In addition, their model checking work has adopted simple direct encodings of finite state machines into the representational language used by the model-based planner MBP [Bertoli et al., 2001] which they use for automatic verification. While sufficient for small problems, this approach is unlikely to scale to the verification of non trivial agent systems. An approach to modelling multi-agent systems and communication has been studied in [Alechina et al., 2006]. However, in this framework memory bounds have not been imposed and scalability of the verification approach is not explored.

Thus, none of the existing approaches allow us to express computational (memory and time) and communication resource limitations altogether, and these approaches do not allow the verification of multi-agent systems considering the interaction between different resources (time, memory and communication bandwidth). Hence there is a need to define frameworks for the representation, specification and verification of resource-bounded agents. This will involve explicit modelling of computational (space and time) and communication resources, implementing practical tools for analysing resource requirements for systems of autonomous reasoning agents , investigating trade-offs between multiple resource bounds, addressing the limitations of the existing approaches, namely the issues of expressiveness and scalability.

1.2 Approaches to formal verification

Automated verification encompasses many different techniques that include testing, run-rime verification, static analysis, theorem proving, and model checking. The various verification techniques mentioned in this section have their own advantages and disadvantages. However, this thesis is concerned with the problem of formal verification for multi-agent systems using model checking. Model checking is an automatic

technique that has been proven very effective in verifying many hardware and software system designs. In the following, we briefly describe the verification techniques mentioned above. However theorem proving and model checking approaches will be described in more detail later in the literature review.

Testing is one of the popular approaches used in verifying traditional software systems. The most common testing methods applied to software systems are *correctness testing*, *performance testing*, *security testing* and *reliability testing* [Pan, 1999]. In testing, the verification is performed by running a number of test cases and checked whether the required properties hold in all these runs. However, the main problem of testing is that it can never completely identify all the defects within design. This is because it must be ensured that the maximum number of different system behaviours are covered using a minimum number of test cases. But the problem is to select a sensible set of test cases. Therefore, testing is applied for a selective test cases which cover only a portion of the system behaviour. While testing may help improve quality for more conventional software systems, it falls short for complex artificial intelligent systems [Wang et al., 2001].

Run-time verification also called runtime monitoring or runtime checking, is another method to increase the quality of critical system design. Run-time verification is concerned with monitoring and analysis of system executions, i.e., the system behaviour is considered at run-time. In this process the input-output behaviour of the system is observed during execution. The observed behaviour (log traces) of the system can be monitored and verified dynamically to satisfy given requirements expressed in temporal logic formulae [D'Angelo et al., 2005]. In recent years several runtime verification systems have been developed that include Java PathExplorer [Havelund and Rosu, 2001], ARVE [Shin et al., 2007], and Mcc [Sharma et al., 2009]. Run-time verification has to deal with finite traces only of the target system, and again it only observes partial executions and thus this technique also provides incomplete verification results.

Static program analysis is another verification technique. It is often used for auto-

matically discovering errors of a target program considering its all possible execution paths. This technique is applied to statically analyse the dynamic properties of a program at compile time without actually executing it [Nielson et al., 2005]. Abstract interpretation is considered to be one of the basic static analysis techniques which is successfully used in program optimization and verification [Cousot and Cousot, 1977, Cousot, 2003]. The most popular abstract interpretation based static analysers which have been developed to support program optimization and verification include PAG [Martin, 1998], ASTRÉE analyser [Cousot et al., 2005] and PolySpace [Technologies, 2008]. Abstract interpretation consists in considering an abstract semantics, which is a sound approximation of the concrete program semantics. For instance, consider the set of concrete points $P = \{(x, y) \in \mathbb{R}^2 \mid x^2 + y^2 \leq 1\}$, then any polygon including P is a sound abstraction of P. Sometimes abstraction may lose precision and the consequence of over approximation of the possible concrete executions can leads to false alarms. Patrick Cousot [Cousot, 2008] argues that: "a grand challenge for abstract interpretation is to extend its scope to complex systems, from specification to implementation, not only to the program part, as is presently the case".

Automated theorem proving is a logic based proof theoretic approach [Gallier, 1986]. Theorem provers typically use a very expressive logic for expressing the implementation and the specification. The system implementation is expressed as a set of axioms, and the specification is expressed as a theorem to be proved in the axiomatic system. The verifier tries to find a proof of the theorem according to the inference rules of the logic. There has been considerable work on verification of multi-agent systems using theorem proving. State of the art of theorem proving tools can be thought of as an interactive tools. This is usually only partially automated and requires an extensive user interaction [Bharadwaj, 1996]. Moreover, the user must be familiar with the logic and the 'proof system' the prover is based on.

Model checking is a model-based verification approach [Clarke et al., 2000]. The description of a system (also known as model) is given by the specification language

of a model checker and verifies that a temporal logic formula holds for the model. However, for the verification of MAS, the properties of the system to be verified are often specified in combined modal and temporal logics, such as temporal logics of belief or temporal logics of knowledge. The output of a model checker is either a confirmation or a denial that the property is violated. If a system state that violates the temporal formula is found, then a model checker usually returns a counterexample. This is very useful for debugging purposes.

1.3 Problem statement

Let us consider an agent that has a finite knowledge base and some rules of inference which allow it to derive new information from its knowledge base. It is intuitively clear that some derivations require more time and/or memory than others (e.g., to store all the relevant information, or to store intermediate results), and that two agents with the same knowledge base and the same set of inference rules, but with different amounts of memory, may not be able to answer the same queries or may take different amounts of time to answer them. If two agents need to communicate in order to answer a query, then the number and size of messages that must be exchanged will depend on the query, and the time and memory available to the agents.

For a given problem and system of reasoning agents, many different solution strategies may be possible, each involving different commitments of computational resources (time and memory) and communication by each agent. For different multi-agent systems, different solution strategies will be preferred depending on the relative costs of computational and communication resources for each agent. These tradeoffs may be different for different agents (e.g., reflecting their computational capabilities or network connection) and may reflect the agent's commitment to a particular problem. For a given system of agents with specified inferential abilities and resource bounds it may not be clear whether a particular problem can be solved at all, or, if it can, what computational and communication resources must be devoted to its solution by each

agent. For example, we may wish to know whether a goal can be achieved if a particular agent, perhaps possessing key information or inferential capabilities, is unable (or unwilling) to contribute more than a given portion of its available computational resources or bandwidth to the problem.

Therefore, the computational resources required by a reasoning agent to achieve a goal or answer a query is of considerable theoretical and practical interest. From a theoretical point of view, it is related to the questions investigated in proof complexity [Haken, 1984, Alekhnovich et al., 2002], of the lower bounds on the size of proofs in deductive systems, and of lower bounds on memory required to verify them. However, a detailed discussion of the theoretical foundation is out of scope for this thesis. From a practical point of view, the question of whether an agent has sufficient resources (memory, time or communication bandwidth) to achieve its goal(s) is clearly a major concern for the agent developer. As agent tasks become more open ended, the amount of resources required to achieve them becomes harder to predict a priori. For example, the reasoning capabilities of agents assumed by many web service applications is non trivial and the time, memory and communication requirements correspondingly difficult for the agent developer to determine a priori. Despite the importance of the topic, there has been relatively little work in this area. While the temporal aspects of reasoning have been considered in the literature mentioned before, there has been no detailed treatment of computational (memory and time) and communication resource requirements, and no systematic investigation of resource trade-offs in resource-bounded reasoners.

In this thesis we define frameworks for the representation, specification and verification of resource-bounded agents. The frameworks allow us to model computational and communication resources explicitly, and to reason about and verify tradeoffs between time, memory and communication in systems of distributed reasoning agents. We are interested in properties such as:

i) there is a possibility that agent i will derive formula φ in n_T time steps while

exchanging fewer that n_C messages;

ii) agent i will always derive formula φ in n_T time steps while exchanging fewer that n_C messages;

iii) every request of agent i will be responded by agent j in n_T time steps (where i and j are distinct agents in the system).

We show how state-of-the-art model checkers can be used to encode and verify properties of systems of distributed reasoning agents. We describe the encoding and report results of model checking experiments which show that even simple systems have rich patterns of trade-offs between multiple (time, memory, and communication) resource-bounds.

1.4 Methodology

The work presented in this thesis is a part of a research project[1] to provide theoretical foundations and practical tools for analysing resource requirements for systems of reasoning agents. The theoretical foundations are based on temporal doxastic[2] logics. We only give a brief description of the logical formalisms whenever necessary to describe our frameworks; full details can be found in [Nga, 2010]. In this thesis, we focus on developing formal frameworks and practical verification tools for analysing resource requirements for systems of reasoning agents. The thesis research methodology is as follows.

1. **Preliminary definition of computational models** To model an agent reasoning about a knowledge base KB and a formula φ in logic L, we represent the states[3] of the agents as assignments to finitely many formulae (all subformulae

[1]This work was partially supported by the Engineering and Physical Sciences Research Council [grant number EP/E031226].

[2]A doxastic logic is a modal logic concerned with reasoning about beliefs. In the next chapter we will see that modal logic plays a prominent role in specifying, reasoning about, and verifying multi-agent systems.

[3]Note that throughout this thesis we use the term *state* and the term *configuration* interchangeably.

of KB and φ). Transitions correspond to applications of inference rules of L to formulae which have a value true or false in the state. The constraint on memory corresponds to a restriction that each agent i in the system has memory of size $n_M(i)$ where one unit of memory corresponds to the ability to store an arbitrary formula. The constraint on communication corresponds to a restriction that each agent's communication ability is $n_C(i)$: in any valid run of the system, agent i can perform at most $n_C(i)$ communication actions.

2. **Investigating trade offs** In [Albore et al., 2006, Alechina et al., 2007] Alechina and colleagues show that there exist examples of propositional formulae which require less time (computational steps) to prove with larger memory, and are still provable but require longer derivations with smaller memory. In this research work we investigate such trade-offs systematically using two typical multi-agent reasoning problems which can be easily parameterized to increase or decrease the problem size. The first class of problems we consider is a resolution based distributed reasoning system, where as the second class of problems corresponds to a distributed system of rule-based agents. We investigate trade-offs between memory and communication costs, and between communication costs and time measured as the number of transitions required by the multi-agent system to achieve the goal, provided the reasoners execute in parallel. Our approach is informed by work in proof complexity on the relationships between the size of proofs and space required to verify them. We also draw on the notions of communication complexity [Yao, 1979] to represent communication costs as a function of the number or size of formulae which reasoners have to exchange to solve a common task.

3. **Scalability analysis** We analyse the scalability issues while verifying resource bounded agents based on two example scenarios mentioned above. We use model checking tools Mocha [Alur et al., 1998a] and NuSMV [Cimatti et al., 2000] while verifying properties of the systems. In order to improve scalability

of model checking for larger problems, we analyse the problem and its encoding complexity to better understand the scalability issues. Both the complexity analysis and experimental results suggest that reasonably sized problem instances are unlikely to be tractable for a standard model checker without steps to reduce the branching factor of the state space.

4. **Developing an automatic tool** We propose two approaches to address the scalability issues identified above: the use of abstract specifications to model the behaviour of some of the agents in the system, and exploiting information about the reasoning strategy adopted by the agents. Abstract specifications are given as Linear Temporal Logic (LTL) formulae which describe the external behaviour of the agents, allowing their temporal behaviour to be compactly modelled. Conversely, reasoning strategies allow the detailed specification of the ordering of steps in the agent's reasoning process. Both approaches have been combined in an automated verification tool TVRBA for rule-based multi-agent systems which allows the designer to specify information about agents' interaction, behaviour, and execution strategy at different levels of abstraction. The tool TVRBA generates an encoding of the system for the Maude LTL model checker [Eker et al., 2003], allowing properties of the system to be verified.

5. **Evaluation** We illustrate the scalability of the new approach by comparing it to results obtained using traditional model checking techniques for a synthetic distributed rule-based reasoning problem (mentioned before). We also show how to further improve scalability by using abstract agents specified in terms of temporal doxastic formulae through two case studies. The experimental evaluation determines the relationship between the size of a knowledge base and time required by the tool to verify resource requirements for typical system properties.

1.5 Thesis contribution

The work presented in this thesis includes material from some of my published papers [Alechina et al., 2008a, 2009a, 2008b, 2010] which were co-authored by my supervisors, and my colleague Nguyen Hoang Nga. In each of the first three papers [Alechina et al., 2008a, 2009a, 2008b], there are two parts: the first part presents a logical framework for resource-bounded MAS and the second part develops and experimentally verifies model checker encodings for resource-bounded MAS formalised using those logical frameworks. My contributions to these papers focus mostly on the devleopment and experimental evaluation of the encodings. The last paper [Alechina et al., 2010] is mostly my work. It presents the abstraction and strategy based verification technique developed in this thesis including the development of the TVRBA verification tool.

The main contributions of this thesis are:

- A brief survey of the logical frameworks for MAS, and formal techniques to verification of such systems. Some limitations of current approaches are also discussed.
- Model checking encoding and verification of resolution-based systems, and the analysis of the problem and its encoding complexity.
- Model checking encoding and verification of rule-based systems, and the analysis of the problem and its encoding complexity.
- Identifying the scalability issues in verifying the above systems, and proposing a new approach to model checking MAS which uses strategies and abstraction.
- Development of an automated verification tool TVRBA that supports strategies and abstraction and uses Maude as a backend for model checking.
- The scalability of the new approach is illustrated using three case studies.

1.6 Thesis structure

The rest of this thesis is organised as follows: Chapter 2 and Chapter 3 review the background literature. We briefly survey the background and history of modal and temporal logics which are used to represent multi-agent systems with a special emphasis on resource-bounded agents. We then look at two approaches to verification: model checking and theorem proving. Finally, we analyse the limitations of the current approaches.

Chapter 4 and Chapter 5 define frameworks for the representation, specification and verification of resource-bounded agents. We consider two typical distributed reasoning systems as mentioned before which reason using (propositional) resolution and (propositional) rules, respectively. We use conventional model checking techniques and use symbolic model checkers Mocha and NuSMV to verify properties of those systems and investigate trade-offs between multiple resource bounds. Furthermore, we analyse the problem and its encoding complexity to better understand the scalability issues.

Chapter 6 presents a framework for the automated verification of multi-agent rule-based systems, which allows the use of abstract specifications consisting of Linear Time Temporal Logic (LTL) formulae to specify some of the agents in the system. The framework also allows the use of agents explicit reasoning strategies. The rules are extended from propositional to first-order horn clause rules. That is rules of an rule-based agent can either be propositional or first-order horn clauses.

Chapter 7 describes an encoding based on the Maude rewriting system which implements the approach to verification described in Chapter 6, and shows how the desired properties of the system can be verified using Maude LTL model checker. It also presents an automated verification tool TVRBA that uses Maude as a backend for model checking.

Chapter 8 illustrates the scalability of the new approach which uses strategy and abstraction using three case studies. In the first case study, we re-implement an exam-

ple scenario introduced in Chapter 5. To illustrate the application of the framework on more complex examples we consider two more case studies: a route planning example, and a home health-care monitoring alarm system.

Chapter 9 summarises briefly the work undertaken, and suggests some possible future lines of research.

These chapters are followed by bibliographic references and appendices.

Chapter 2

Logical formalisms for MAS

In this chapter, we present a brief survey of existing research on modal logics concentrating on the approaches which have influenced the work presented in this thesis. Two different areas are identified and addressed in the literature review. The first area is the theoretical foundations (in general), which is addressed in this chapter. The second area is concerned with the practical tools and formal verification of multi-agent systems, which will be addressed in the next chapter.

2.1 Modal logics

Modal logics are regarded as the most suitable and versatile logical formalisms for specifying, reasoning about, and verifying multi-agent systems. In essence, modal logic extends propositional or first-order logic to include the modal operators. That is modal logics often use modes of truth. The most well-known modalities that are used in modal logics include *possibility* and *necessity*. For example, the following are modal propositions: "it is possible that φ", and "it is necessary that φ" for some proposition φ. The operators "it is possible that" ($\Diamond$) and "it is necessary that" ($\Box$) are called modal operators. The *possibility* operator can be expressed using *necessity* (and vice versa) as follows: $\Diamond\ \varphi \equiv \neg\Box\neg\varphi$ (and $\Box\ \varphi \equiv \neg\Diamond\neg\varphi$). The presence of a modal operator in front of a proposition specifies a way in which the rest of the proposition can be said

to be true.

In the literature [Hintikka, 1962], a wide variety of modal logics have been proposed including *epistemic* logic which deals with the mental attitude of knowledge and *doxastic* logic which treats belief. These logics have become popular in Computer Science and AI to describe the informational aspects such as knowledge and belief of agents. Epistemic modalities deal with the certainty of sentences, and the $\Box$ operator is usually translated as K_i which can be read as "agent i knows that". Similarly, a doxastic logic uses $\Box$, often written as B_i which can be read as "agent i believes that".

The language of basic modal logic is that of propositional logic with two extra connectives $\Box$ and $\Diamond$. Let P be a set of propositional variables. The formulae of basic modal logic are defined by the following grammar:

$$\varphi ::= \quad p \mid \neg\varphi \mid \varphi_1 \rightarrow \varphi_2 \mid \Box\varphi$$

where $p \in P$. Classical abbreviations for $\wedge$, $\vee$, $\leftrightarrow$, and $\Diamond$ are defined as usual.

The semantics for modal logic are given by Kripke structures of the form $\mathcal{M} = \langle W, v, R\rangle$, where:

- W is a non-empty set of states or worlds;
- $v : P \times W \rightarrow \{\text{true,false}\}$ is a truth assignment function;
- $R \subseteq W \times W$ is a binary relation on W, known as the accessibility relation.

The truth of a formula in a model $\mathcal{M} = \langle W, v, R\rangle$ and $w \in W$ is defined inductively as follows:

- $\mathcal{M}, w \models p$ iff $v(p, w) =$ true;
- $\mathcal{M}, w \models \neg\varphi$ iff $\mathcal{M}, w \not\models \varphi$;
- $\mathcal{M}, w \models \varphi \rightarrow \psi$ iff either $\mathcal{M}, w \not\models \varphi$ or $\mathcal{M}, w \models \psi$;
- $\mathcal{M}, w \models \Box\varphi$ iff for all $v \in W$ such that $R(w, v), \mathcal{M}, v \models \varphi$.

For a more detailed description of modal logics we refer the interested reader to [Blackburn et al., 2001]. Throughout this thesis we will see combinations of logics that have been used to specify multi-agent systems. When specifying MAS we may also need to represent temporal aspects of systems, which are typically modelled using temporal logics [Pnueli, 1979]. In the following section, we present two temporal logics which are often used to model dynamic behaviour of concurrent systems.

2.2 Temporal logics

Temporal logic can express properties about how the system evolves along computations. In the literature several temporal logics have been proposed, e.g., [Pnueli, 1979][Clarke et al., 2000, pp. 27–30], each one has its own collection of temporal operators. These logics are categorised into *linear* time and *branching* time logics. In linear time logic (LTL), formulae are interpreted over paths. When we interpret a formula over a set of paths, we always quantify universally over all possible paths in the set. In branching time logic, known as computation tree logic (CTL), the computation is viewed as a tree-like structure. The logic CTL allows path quantifications, i.e., we can reason about all or some paths starting in a state in the tree.

2.2.1 Computation tree logic

Syntax of CTL : The formulae in CTL are classified into state and path formulae. The state formulae are assertions about the atomic propositions in the states and their branching structure, on the other hand the path formulae express temporal properties of paths. The basic components of CTL formulae are $AP = \{p_1, p_2, \ldots\}$ a set of propositional variables, standard Boolean connectors, temporal operators X (next), G (globally), F (eventually), U (until), and path quantifiers A (universal) and E (existential). The formulae of CTL are constructed inductively as follows:

$$\varphi ::= \top \mid p \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid EX\varphi \mid EG\varphi \mid EU(\varphi_1, \varphi_2)$$

where $p \in AP$ and the path quantifier E stands for "for at least one path". The classical abbreviations for $\vee, \rightarrow, \leftrightarrow$ and $\bot$ are defined as usual.

In CTL the basic operators are EX, EG, and EU. Other operators such as AX, AG, AF, AU (the path quantifier A stands for "for all paths") and EF can be expressed in terms of EX, EG, and EU:

$$EF\varphi \quad \text{iff} \quad EU(\top, \varphi)$$

$$AX\varphi \quad \text{iff} \quad \neg EX\neg\varphi$$

$$AG\varphi \quad \text{iff} \quad \neg EF\neg\varphi \quad \text{iff} \quad \neg EU(\top, \neg\varphi)$$

$$AF\varphi \quad \text{iff} \quad AU(\top, \varphi) \quad \text{iff} \quad \neg EG\neg\varphi$$

$$AU(\varphi_1, \varphi_2) \quad \text{iff} \quad \neg EU(\neg\varphi_2, \neg\varphi_1 \wedge \neg\varphi_2) \wedge \neg EG\neg\varphi_2$$

Semantics of CTL: The semantics of CTL is defined by a state transition graph $\mathcal{M} = (\mathcal{S}, \mathcal{S}_I, \mathcal{T}, \mathcal{L})$ where

i) $\mathcal{S}$ is a finite non-empty set of states of $\mathcal{M}$;

ii) $\mathcal{S}_I \subseteq \mathcal{S}$ is a non-empty set of states, called the set of initial states of $\mathcal{M}$;

iii) $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{S}$ is a set of pairs of states, called the transition relation of $\mathcal{M}$;

iv) $\mathcal{L} : \mathcal{S} \rightarrow \wp(AP)$ is a function, called the labelling function of $\mathcal{M}$.

Let $s \in \mathcal{S}$ and a formula φ over the language of $\mathcal{M}$, then the relation of semantic entailment $\mathcal{M}, s \models \varphi$ is defined inductively on the structure of φ as follows:

- $(\mathcal{M}, s) \models \top$,
- $(\mathcal{M}, s) \models p$ iff $p \in \mathcal{L}(s)$,
- $(\mathcal{M}, s) \models \neg\varphi$ iff $\mathcal{M}, s \not\models \varphi$,
- $(\mathcal{M}, s) \models \varphi_1 \wedge \varphi_2$ iff $\mathcal{M}, s \models \varphi_1 \wedge \mathcal{M}, s \models \varphi_2$,

- $(\mathcal{M}, s) \models EX\varphi$ iff $\exists$ a path $\pi = s_0, s_1, s_2, \ldots$ s.t. $s_0 = s \wedge (s_i, s_{i+1}) \in \mathcal{T}$, $\mathcal{M}, s_1 \models \varphi$,
- $(\mathcal{M}, s) \models EG\varphi$ iff $\exists$ a path $\pi = s_0, s_1, s_2, \ldots$ s.t. $s_0 = s \wedge (s_i, s_{i+1}) \in \mathcal{T}$, $\forall i\, \mathcal{M}, s_i \models \varphi$,
- $(\mathcal{M}, s) \models EU(\varphi_1, \varphi_2)$ iff $\exists$ a path $\pi = s_0, s_1, s_2, \ldots$ s.t. $s_0 = s \wedge (s_i, s_{i+1}) \in \mathcal{T}$, $\exists i\, \mathcal{M}, s_i \models \varphi_2 \wedge \forall (j < i)\, \mathcal{M}, s_j \models \varphi_1$.

2.2.1.1 Expressing properties in CTL

A wide variety of system properties can be expressed using CTL. In this section we give some generic [Clarke et al., 2000] CTL properties which are often used in verifying finite state concurrent systems.

Liveness and safety properties. A liveness property states that: "something **good** will eventually happen" [Lamport, 1977], i.e., eventually (after a finite number of steps) some formula φ holds. Reachability of a state satisfying a formula φ can be expressed as the existence of a path satisfying $EF\varphi$. A safety property states that: "something **bad** will never happen" [Lamport, 1977]. Safety properties can be expressed as non-reachability of a state satisfying φ, i.e., the property $AG\neg\varphi$.

In the following we mention a number of useful example properties which can be stated in CTL.

Responsiveness. In distributed systems it is often the case that one process sends requests that have to be responded to by other processes. For such systems we are interested in the responsiveness property: every request must eventually be responded to. Assuming that the *request* is expressed by a formula φ and *response* by a formula ψ, one can express responsiveness by the formula $AG(\varphi \rightarrow AF\psi)$.

Mutual exclusion. Two or more processes are not allowed to enter the same critical section of a concurrent system simultaneously. Assuming that there are two processes P_1, P_2, and that formulae φ_i, where $i = 1, 2$ denote that P_i is in the critical section,

mutual exclusion can be expressed by the formula $AG\neg(\varphi_1 \wedge \varphi_2)$.

Precedence. From all reachable states satisfying φ, it is possible to maintain φ continuously until reaching a state satisfying ψ, can be expressed using the formula $AG(\varphi \rightarrow E[\varphi U \psi])$.

Non-blocking. A process can always request to enter its critical section, can be expressed using the formula $AG(\varphi \rightarrow EX\psi)$.

2.2.2 Linear temporal logic

Syntax of LTL: The basic components of LTL formulae are $AP = \{p_1, p_2, \ldots\}$ a set of propositional variables, standard Boolean connectors, and temporal operators X (next), G (globally), F (eventually), U (until), and R (release). The formulae of LTL are constructed inductively as follows:

$$\varphi ::= \top \mid p \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid X\varphi \mid U(\varphi_1, \varphi_2)$$

where $p \in AP$. Classical abbreviations for $\vee$, $\rightarrow$, $\leftrightarrow$ and $\bot$ are defined as usual. Other temporal operators can be expressed as: $R(\varphi_1, \varphi_2) \equiv \neg(U(\neg\varphi_1, \neg\varphi_2))$, $F\varphi \equiv U(\top, \varphi)$ and $G\varphi \equiv R(\bot, \varphi) \equiv \neg\, F\neg\varphi$.

Semantics of LTL: The semantics of LTL is defined by a state transition graph $\mathcal{M} = (\mathcal{S}, \mathcal{S}_I, \mathcal{T}, \mathcal{L})$ where

i) $\mathcal{S}$ is a finite non-empty set of states of $\mathcal{M}$;

ii) $\mathcal{S}_I \subseteq \mathcal{S}$ is a non-empty set of states, called the set of initial states of $\mathcal{M}$;

iii) $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{S}$ is a set of pairs of states, called the transition relation of $\mathcal{M}$;

iv) $\mathcal{L} : \mathcal{S} \rightarrow \wp(AP)$ is a function, called the labelling function of $\mathcal{M}$.

A path π of $\mathcal{M}$ is an infinite sequence $s_0, s_1, \ldots, s_n, \ldots$ of states such that $(s_i, s_{i+1}) \in \mathcal{T}$ for all $i \geq 0$. For all $i = 0, 1, 2, \ldots$ denote by π_i the sequence of states $s_i, s_{i+1}, s_{i+2}, \ldots$,

then $\pi_0 = \pi$. The truth of an LTL formula φ on π of $\mathcal{M}$ denoted by $(\mathcal{M}, \pi) \models \varphi$ is defined inductively on the structure of φ as follows:

- $(\mathcal{M}, \pi) \models \top$,
- $(\mathcal{M}, \pi) \models p$ iff $p \in \mathcal{L}(s_0)$,
- $(\mathcal{M}, \pi) \models \neg\varphi$ iff $(\mathcal{M}, \pi) \not\models \varphi$,
- $(\mathcal{M}, \pi) \models \varphi_1 \wedge \varphi_2$ iff $(\mathcal{M}, \pi) \models \varphi_1$ and $(\mathcal{M}, \pi) \models \varphi_2$,
- $(\mathcal{M}, \pi) \models X\varphi$ iff $(\mathcal{M}, \pi_1) \models \varphi$,
- $(\mathcal{M}, \pi) \models U(\varphi_1, \varphi_2)$ iff $\exists k \geq 0 : (\mathcal{M}, \pi_k) \models \varphi_2 \ \wedge \ \forall\, 0 \leq i < k \ : \ (\mathcal{M}, \pi_i) \models \varphi_1$.

The truth of an LTL formula using other Boolean connectives and temporal operators can be defined analogously.

2.2.2.1 Expressing properties in LTL

Some typical properties expressed in LTL which are often used in verifying concurrent systems are given below.

Reachability: $F\neg\varphi$, $U(\varphi, \psi)$ etc.

Safety: $G\neg\varphi$, $U(\varphi, \psi) \vee F\varphi$ etc.

Liveness: $G\varphi \rightarrow F\psi$, $G(\varphi \rightarrow F\psi)$ etc.

Precedence: $G(\varphi \rightarrow U(\psi_1, \psi_2))$

Mutual exclusion: $G\neg(\varphi_1 \wedge \varphi_2)$

Fairness: $GF\varphi$, $GF\varphi \rightarrow GF\psi$ etc.

2.3 Logics for resource-bounded agents

The possible worlds model for logics of knowledge and belief was originally introduced by Hintikka [Hintikka, 1962], in this model an agent's beliefs are characterized as a set of possible worlds. The later work of Kripke [Kripke, 1963] has shown how possible worlds may be incorporated into the semantic framework of a logic. However, representing knowledge in terms of possible worlds semantics suffers from the *logical omniscience problem*, a term coined by Hintikka [Hintikka, 1962]. Logical omniscience presupposes that an agent knows all logical consequences of its beliefs and all valid sentences including tautologies.

Representing knowledge in terms of traditional possible worlds semantics is quite useful. However, such semantics do not account the fact that agents possess limited computational resources. In traditional possible worlds semantics if an agent considers possible a world where a formula φ is true and $\varphi \rightarrow \psi$, then the agent instantly imagines the formula ψ is also true in that world. However, practical agents in a multi-agent system take time, space and perhaps communication to derive the consequences of their beliefs. Thus the classical approach to knowledge representation poses a problem when modelling resource-bounded reasoners.

There are number of proposed solutions to solve the logical omniscience problem which develop alternative logical formalisms for representing knowledge and belief. These include Levesque's [Levesque, 1984] logic of implicit and explicit belief, Fagin and Halpern's [Fagin and Halpern, 1985] logic of general awareness, Konolige's [Konolige, 1986] deduction model of belief. However, we do not discuss these logics because these approaches do not explicitly take account of time, space and/or communication. Logical research which represents reasoning as a process that explicitly requires resources, includes Elgot et al.'s step logic [Elgot-Drapkin and Perlis, 1990], Halpern et al.'s algorithmic knowledge [Halpern et al., 1994], Duc's dynamic logic [Duc, 1995, 1997], and Alechina et al.'s [Alechina et al., 2007, 2006, Albore et al., 2006] logics for resource bounded agents, among the others. In the following

we present a brief review of these logical formalisms.

2.3.1 Step-logic

Elgot-Drapkin and Perlis introduced step logic [Elgot-Drapkin and Perlis, 1990] to explicitly model the time that an agent requires for its belief derivation. The logic is characterized by a language, a set of inference rules, and a set of observations. The reasoning systems proceed to draw conclusions in steps, certain conclusions or observations may arise after some time steps based on the agent's earlier beliefs. The logic is non-monotonic, it cannot in general retain or inherit all conclusions from one step to the next. Step-logic is a pair $\langle SL^n, SL_n \rangle$ of meta-theory (SL^n) and its corresponding agent-theory (SL_n). The meta-theory is used as a scientific theory for the user who might think about agent's reasoning. However, it is not used formally for knowledge representation, e.g., in computer program. The agent-theory is the main part of the step-logic which aims to capture the on-going reasoning of agents. In this logic the agent's deduction steps are explicitly modelled in the logic, which makes it possible to model the time it takes the agent to arrive at a certain conclusion. However, it does not capture the space required.

In a later work [Elgot-Drapkin et al., 1991] Elgot-Drapkin and colleagues proposed a memory-based model of reasoning, based on the step-logic framework, where bounds on working memory were considered. The memory model contains five key components STM (short-term memory), LTM (long-term memory), ITM (intermediate-term memory), QTM (queue-term memory), and RTM (recent-term memory). STM contains the set of beliefs that are currently active, and its size is bounded. STM is structured as a FIFO queue, when new facts are brought into STM, old facts must be evicted due to the bounded size. LTM is a large database where beliefs are held as a series of tuples of the form $\langle T_1, T_2, \ldots, T_n, B \rangle$, where the T_i and B represent logical formulae. ITM is just a chronological list of past items that were in STM; it is a history of the agent's thoughts. QTM holds incoming (new) items briefly, to check whether they (i.e.,

copies of them) are already in STM, before letting them enter STM. It prevents STM from being flooded with identical copies of items. RTM holds items that have been in STM recently, since they are presumed relevant to current issues the agent is working on and thus are handy to have easy access to those items.

An inference cycle is the process of updating the system's current beliefs (STM). The system moves from a given state (STM) to a new state (STM) by performing four different mechanisms simultaneously. These four mechanisms are direct observations, modus ponens (MP), semantic retrieval (from LTM), and episodic retrieval (from ITM). In order to model this simultaneity, the implementation uses a temporary waiting queue (QTM) which holds the next cycle's STM facts until all four mechanisms have finished working on the old STM facts. Once they have finished, elements of QTM are placed into STM one at a time, disallowing repetition of facts in STM. Throughout this process, older elements in STM are moved into ITM to maintain STM's size.

In the step-logic framework an agent's beliefs are seen as a set of sentences that changes over time. The logic models belief reasoning by allowing inference one-step at a time by means of a time arguments in the agent-theory. Step logic takes memory bounds into account but does not address issues like verification of space requirements for solving a certain problem.

2.3.2 Algorithmic knowledge

Halpern and colleagues [Halpern et al., 1994] presented a framework to capture the computational properties of knowledge based on interpreted systems [Fagin et al., 1995] to take the computational aspects of knowledge into account. The formalism of interpreted systems consist of four key components:

i) a set of local states: which describes the private information of each agent;

ii) a set of actions: which describe the set of possible actions that an agent can perform;

iii) a protocol: which is a rule that manages the actions to be performed in a local state and non-deterministic choice will be made when multiple actions are enabled; and

iv) an evolution function: which describes the evolution of the system.

A global state is a sequence $(s_e, s_1, \ldots, s_n)$ of local states s_i along with an environment state s_e. A run of the system $r : \mathbb{N} \rightarrow G$ is a function from time (assumed to be discrete) to global states. A system $\mathcal{R}$ is a set of runs. The interpreted system $\mathcal{I}$ is a pair $(\mathcal{R}, \pi)$, where π is a truth assignment function.

Based on this framework, agents are assumed to possess a procedure which they use to produce knowledge. The intuition is that the agent knows a fact φ if it can compute that it knows φ. This is modelled by saying that the agent has an algorithm to decide if it knows φ. It is evident that an agent's knowledge depends on its local state: to decide if agent i knows φ, the algorithm takes as input a local state and the formula φ and returns the output as one of the answer "YES" , "NO", and "?". If $r(m) = (s_e, s_1, \ldots, s_n)$ and $r_i(m) = s_i = \langle A, l \rangle$ is the local state of an agent i. For a state (r, m) of the system on run r at time m, $alg_i(r, m)$ is used to denote the algorithm A and $data_i(r, m)$ is used to denote the local data l. Halpern and colleagues denote the algorithmic knowledge by the modal operator X_i which is defined as

$$(\mathcal{I}, r, m) \models X_i\varphi \text{ iff } A(\varphi, l) = \text{"YES"}, \text{ for } A = alg_i(r, m) \text{ and } l = data_i(r, m).$$

This differs from standard interpreted systems in that $X\varphi$ is true in a state, if the output of the agent's algorithm is "YES" with inputs φ and its local data.

The algorithmic knowledge approach does not address time or memory boundedness explicitly. In this approach, agents are assumed to possess a procedure which they use to produce knowledge. However, the approach is concerned with the result rather than the process of producing knowledge.

2.3.3 Dynamic logic

In [Duc, 1995, 1997] Duc proposed an epistemic logic to reason about agents that are logically non-omniscient. The language of the logic is based on formulae of the form $K\varphi$ (the agent knows φ) for some propositional formula φ, and closed under negation, conjunction and two future modal operators $\langle R\rangle$ and $[R]$ for each agent. Duc defines operators $\langle R\rangle$ and $[R]$ which can be thought of something is true "at some future time" and "at all future times" respectively. For instance $\langle R\rangle K\varphi$ has the following meaning: sometimes after using inference rule R the agent knows φ, whereas $[R]K\varphi$ formalizes the fact that always after using inference rule R the agent knows φ. Agents represented within this framework are non-omniscient because their actual beliefs at a single time point need not be closed under any law. Agents will believe all consequences of their beliefs eventually, after some interval of time. However, Jago [Jago, 2006] showed that the future modality $\langle R\rangle$ used in the dynamic logic does not capture specific resource bounds on the agent's computational ability. For example, if φ is a very large propositional (modality-free) tautology then $\langle R\rangle B\varphi$ is a theorem. If the agent's memory is not large enough to hold the sentence in its memory, then it is not correct to read $\langle R\rangle B\varphi$ as " the agent believes φ at some future time".

In Duc's dynamic logic agent's deduction steps are explicitly modelled which makes it possible to model the time it takes the agent to arrive at a certain conclusion, however dynamic logic doesn't take memory bounds into account.

2.3.4 An explicit model of memory

In [Alechina et al., 2007] Alechina and colleagues have taken some preliminary steps towards the automated verification of resource requirements of reasoning agents. An agent consists of a knowledge base (KB) and some rules of inference (R), and it can derive new information from KB using R. The proposed framework investigates whether the agent has sufficient memory to derive a given formula φ, and if so what would be the length of the shortest derivation when a bounded memory size is given. They show

that the memory requirements may differ for logically equivalent knowledge bases as well as inference rules available to the agent. For example, consider an agent with knowledge base $KB_1 : \{A, A \rightarrow B, B \rightarrow C, C \rightarrow D\}$ that reasons using modus ponens (MP), and another agent with knowledge base $KB_2 : \{A, A \rightarrow B, A \wedge B \rightarrow C, B \wedge C \rightarrow D\}$ which reasons using MP and conjunction introduction ($\wedge_I$). Each agent's memory usage is modelled as the maximal number of formulae in the agent's memory at any given time. Then to derive the goal formula D from the two logically equivalent knowledge bases KB_1 and KB_2, the memory requirements are different: 2 and 3 respectively.

The resulting logic BML^d is interpreted on transition systems. Firstly, they have defined the language and transition systems for definite reasoners that works for rule-based agents. The transition system of this model is defined as a triple $\langle S, R, \pi \rangle$, where S is a set of states, R is a transition relation on S which is serial, and π is formula assignment function which may assign a set of complex, contradictory formulae to a state. The bounds on memory are restricted by allowing π to assign at most n formulae to any given state. Then they have introduced a more complex logic for agents reasoning by cases that need to maintain a set of epistemic alternatives. A transition system of this kind of reasoner comprises of a 6-$tuple$ $\langle S, R, W, y, t, f \rangle$, where S is a set of states, R is a relation on S, W is a set of epistemic alternatives, y is an assignment function that assigns a set of epistemic alternatives to a given state, t and f are functions which determines the *true* and *false* values for a given formula in a world. The memory bound is imposed by the condition $|t(w)| + |f(w)| \leq n$, where $t(w)$ and $f(w)$ are disjoint. Interesting properties of an agent that can be expressed include, e.g., $EX^{\leq n}B\varphi$: the agent believes φ (or φ is in the agent's memory) in n timesteps.

2.3.5 An explicit model of communication

A formalism for how the beliefs of communicating rule-based agents change over time is studied in [Alechina et al., 2006]. Here a multi-agent case is considered where

agents communicate with each other by asking or telling. Agents communicate only literals, they cannot ask or tell the rules they believe. The modalities "Ask" and "Tell" are introduced to the agents internal language. A multi-agent model is defined as an $(n{+}3)$-$tuple$: $\langle \mathcal{S}, \mathcal{T}, \{i\}_{i\in\{1,\ldots,n\}}, \{f_i\}_{i\in\{1,\ldots,n\}}\rangle$, where $\mathcal{S}$ and $\mathcal{T}$ represents set of states and accessibility relation respectively, $\{i\}_{i\in\{1,\ldots,n\}}$ is a set of agents and each f_i is a labelling function corresponding to an agent i which assigns a set of formulae to a given state. To capture the agent's behaviour some conditions are applied to the assignment functions f_i and the accessibility relation $\mathcal{T}$. The model of the proposed logic can be encoded in the description language of a model checker to verify properties of agents automatically. However, memory bounds have not been imposed in this framework, so it is considered that agents possess enough space to store beliefs and assumed that the agent's reasoning is monotonic. Interesting properties of a system that can be expressed include, e.g., $\Box^n B_i\varphi$, where $\Box^n$ stands for n nestings of $\Box$. The property $\Box^n B_i\varphi$ states that always agent i believes formula φ in n timesteps. Since in this framework the formulae are not deleted once they are in the agent's memory, $\Box^{\leq n} B\varphi$ entails $\Box^n B\varphi$, so using the notation $\Box^n B_i\varphi$ is enough to state that always agent i believes φ in n timesteps.

2.4 Analysis and expressiveness

In this survey of logical formalisms for multi-agent systems, we have mainly focused on the logics of limited or restricted reasoning. There is a growing body of work where the agent's deduction steps are explicitly modelled in the logic, for example [Duc, 1997], [Alechina et al., 2007, Albore et al., 2006], which makes it possible to model the time it takes the agent to arrive at a certain conclusion. Both the time and space limitations on the agent's knowledge were considered in step logics [Elgot-Drapkin et al., 1991]. However, [Elgot-Drapkin et al., 1991] are not concerned with expressing and verifying space requirements for systems while solving a particular problem, rather they are concerned with restricting the size of short term memory to isolate any

possible contradictions. The logical framework presented in [Alechina et al., 2007] investigates whether an agent with a knowledge base KB, has sufficient memory to derive a given formula φ. The logical syntax contains both temporal and epistemic operators. Interesting properties of an agent that can be expressed include, for example, the agent can derive a goal formula φ from its knowledge base KB as $EFB\varphi$ (there is some future state where the agent believes formula φ). Similarly, that a formula is derivable in n timesteps can be expressed as $EX^{\leq n}B\varphi$. However, while this work represents a significant advance on the state of the art in doxastic logics, it considers only single agents and ignores communication costs.

In [Alechina et al., 2006] Alechina and colleagues have presented a complete and sound modal logic which describes how the beliefs of communicating agents which reason using rules evolve over time. This logic can be used to express and verify temporal properties of multi-agent systems such as "if agent i asks agent j λ (literal), agent j is guaranteed to reply within n inference cycles". However, memory bounds have not been imposed in this framework.

Therefore, there is a need for logical formalisms which will be expressible enough for the representation of real (resource-bounded) reasoning agents with different reasoning capabilities (e.g., agents reasoning in propositional logic, resolution, or rule-based agents) in a cooperative setting. The logical framework will allow us to express properties of systems to investigate trade-offs between multiple resource bounds (memory, time and communication bandwidth). In § 4.4 we present a temporal doxastic logic, BMCL-CTL developed by Nga and Alechina [Alechina et al., 2009a], which allows us to describe a set of reasoning agents with bounds on time, memory and the number of messages they can exchange.

Chapter 3

Formal verification approaches to MAS

Formal verification techniques for multi-agent systems are still in their infancy. This is because of the complex nature of multi-agent systems and difficulties in verifying properties which have non-trivial dynamics specified with rich languages. Nevertheless, there has been considerable work on verification of multi-agent systems using both proof theoretic and model theoretic approaches. In model based verification approaches model checking techniques are used which are based on the semantics of the specification language. In contrast, proof theoretic approaches generally rely on theorem proving techniques. This chapter reviews the state-of-the-art in verifying multi-agent systems.

3.1 The proof theoretic approach

In the proof theoretic approach, theorem proving techniques are used to show syntactically that the specification is a logical consequence of a given set of premises. In this approach, in order to discover a deductive proof, logical expressions are manipulated by means of rules of inference of the form:

$$\frac{P_1, P_2, \ldots, P_n}{C}\mathsf{Name}$$

where C is a conclusion, and $P_1, P_2, \ldots, P_n$'s are premisses. The inference rule says that if all the premisses are derivable, then the conclusion is guaranteed to hold.

Inference rules may have no premisses: in that case their conclusion automatically holds. Such rules are called axioms. A formal proof is a finite sequence of formulae, each of which is either an axiom or the result of applying a rule of inference to previous members of the sequence. A logical theory consists of a grammar for formulae, a collection of axioms, and a collection of rules of inference.

Proving properties by hand is often infeasible, and instead automated theorem proving techniques are typically used. There are a range of approaches to automated theorem proving from proof assistants (e.g., the Coq proof assistant [Huet et al., 2009]) to fully automated systems (e.g., MSPASS [Hustadt and Schmidt, 2000] and TeMP [Hustadt et al., 2004]). A proof assistant is an interactive proof editor (or other interface) with which a human can guide the search for proofs. This may include, for example, the invention and ordering of lemmas. Automated theorem proving, on the other hand, deals with the development of computer programs that find a proof for any given formula if there is one without human intervention (though the user may still have to choose which axioms to include from the logical theory). As noted in Chapter 2, formal verification requires a formal model of the verified system and formal descriptions of the properties to be verified. In theorem proving techniques, both the system and desired properties are specified in a single specification language. Rushby [Rushby, 2001] argues that: "one of the most fundamental choices in this approach to verification is that of the logic on which to base the specification language. There is a trade-off, at least in theory, between expressiveness of the logic and the automation that can be provided for it". In general automated theorem provers use classical first-order logic as a specification language. Classical first-order logic is expressive enough to allow the specification of many systems and properties. However, in some cases a non-classical or possibly a higher-order logic must be used, which can limit the degree of automation of the proof procedure.

3.1.1 Verification using theorem proving

There has been considerable work on verifying properties of agent-based systems using theorem proving. In the remainder of this section, we briefly review previous work in which theorem provers have been used to verify properties of multi-agent systems. Note that this section is not intended to be a comprehensive survey, but merely to give a flavour of the use of proof theoretic approach to verifying multi-agent systems.

3.1.1.1 Modelling and verification framework using ConGolog

In [Lespérance et al., 1999], Lespérance et al. have proposed a framework modelling business processes for requirements analysis using ConGolog [Lespérance et al., 1995, Giacomo et al., 2000], and verifying that the processes satisfy certain properties. The semantics of ConGolog is based on an extended version of the situation calculus [McCarthy and Hayes, 1987]. Situation calculus is a first order language for representing dynamic domains. It uses the following constructs to model a system.

- Actions: all changes to the world are the result of actions, which are denoted by function symbols.
- Situations: a possible world history which is simply a sequence of actions is represented by a first-order term called a *situation*.
- Fluents: relations whose truth values vary from situation to situation, called *relational fluents* and functions whose denotations vary from situation to situation are called *functional fluents*.

The actions in a domain are specified by providing the following axioms.

i) Action preconditions: which describes when actions may be performed.

ii) Action postconditions: which describes what would be the affects after performing an action.

The Golog [de Giacomo et al., 1997] logic-programming language includes the following constructs (δ, possibly subscripted, ranges over Golog programs) for complex actions:

- a primitive action;
- $(\delta_1; \delta_2)$ sequence of actions;
- $(\delta_1 | \delta_2)$ non-deterministic choice between actions;
- $\pi v.\delta$ non-deterministic choice of arguments;
- δ^* non-deterministic iteration;
- $\{\mathbf{proc}\ P_1(\vec{v_1})\ \delta_1\ \mathbf{end}; \ldots; \mathbf{proc}\ P_n(\vec{v_n})\ \delta_n\ \mathbf{end}; \delta\}$ procedures.

The ConGolog language contains all features of Golog with some additional constructs (P represents a program)—**if** ϕ **then** δ_1 **else** δ_2 (conditional execution); **while** ϕ **do** δ (loop execution); $(\delta_1 \| \delta_2)$ (concurrent execution); $(\delta_1 \rangle\rangle \delta_2)$ (priority based execution); $\langle \phi \rightarrow \delta \rangle$ (interrupt on execution).

To illustrate the use of the framework, a simple mail-order business domain was modelled for simulation and verification. The system consists of two agents: the *order desk operator* agent, who processes payment for orders while waiting for the phone to ring, and when it does, receives an order from a customer; and the *warehouse operator* agent, who fills the orders that the order desk operator has received, and ships orders for which the order desk operator has processed payment; whenever a shipment is delivered by a supplier, the warehouse operator receives the shipment. The system is described in ConGolog in terms of *situations* (or state), *actions*, and *fluents*. The system starts in a particular *situation* (or state) and evolves into various other possible situations through *actions* performed by the agents. For instance in this example, $ShipOrder(i, order)$ represents the action of agent i shipping $order$, the relational fluent $OrderShipped(order)$ represents the fact that $order$ has been shipped. Given the specification of the mail-order business domain, interesting properties from the point

of view of verification include (i) no order is ever shipped before payment is processed, i.e., $\forall order \cdot OrderShipped(order) \supset PaymentProcessed(order)$; and (ii) the mail-order business should have income, i.e., there is a situation where $\exists order \cdot Payment\ Processed(order)$. A user-assisted verification tool based on theorem proving is developed [Shapiro et al., 1997]. The tool relies on an encoding of the ConGolog semantics in a form that the PVS [Owre et al., 1992] theorem prover can reason with. The tool can be used to verify the above properties automatically.

In later work [Shapiro et al., 2002], Shapiro et al., introduced the Cognitive Agent Specification Language (CASL), and proposed a framework CASLve for specifying and verifying properties of multi-agent systems implemented in ConGolog. CASL models knowledge using a possible worlds semantics adapted to the situation calculus. $K(i, s', s)$ is used to denote that in situation s, agent i thinks that it could be in situation s'. An agent i knows a formula φ , if φ is true in all situations K-accessible by agent i. In CASL, goals are modelled using an accessibility relation W over possible situations. $W(i, s', s)$ holds if in situation s, agent i considers that in situation s' all its goals are satisfied. Agent communication is achieved through two generic communication actions, $informWhether(i, j, \varphi)$ where agent i informs agent j of the truth value of the proposition φ, and $informRef(i, j, \theta)$ where agent i informs agent j of the value of the term θ. The verification framework CASLve is again based on the theorem prover PVS and has been used to specify a meeting scheduler multi-agent system in CASL, and to prove that all bounded-loop CASL programs terminate.

3.1.1.2 Verifying knowledge properties of security protocols

In [Dixon et al., 2007], Dixon et al. have investigated the application of temporal logic of knowledge to the specification and verification of security of protocols. The proposed framework allows modelling security protocols using temporal epistemic logic, specifying and ultimately verifying some interesting properties using theorem proving. The resulting logic named as $KL_{(n)}$, is a fusion of LTL with epistemic logic. Formulae

of $KL_{(n)}$ are constructed from a set P of primitive propositions. The language of $KL_{(n)}$ contains standard Boolean connectors, temporal operators of LTL, and for knowledge a set of unary modal connectives K_i is introduced for each agent i for a set of agents $Ag = \{1, 2, \ldots, n\}$. The formulae of $KL_{(n)}$ are constructed inductively as follows:

$$\varphi ::= \top \mid p \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid X\varphi \mid U(\varphi_1, \varphi_2) \mid K_i\varphi$$

where $p \in P$. The semantics of $KL_{(n)}$ are defined in the obvious way using transitions systems. The proposed framework is illustrated using a simple system known as the Needham-Schroeder protocol (NSP) with public keys [Needham and Schroeder, 1978]. The Needham-Schroeder protocol with public keys establishes authentication between an agent i who initiates a protocol and an agent j who responds to i. Interesting properties of NSP that can be expressed include, e.g.,

$$G(rcv(j, m1, pub\_key(j)) \rightarrow XK_j\, val\_nonce(N_i, a_n))$$

which states that "once j receives the nonce of i encoded by j's public key then j knows the value of that nonce". Here $m1$ represents the first message of NSP, i.e., i's identity and nonce encoded in j's public key and a_n is the actual value of i's nonce. Dixon et al. have shown how these simple properties can be verified by hand using clausal resolution for $KL_{(n)}$ as well as using the resolution theorem prover TeMP to carry out these proofs automatically.

3.1.1.3 Discussion

The modelling and verification frameworks presented in [Lespérance et al., 1999, Shapiro et al., 2002] describe agents' knowledge and goals based on situation calculus and *knowledge producing actions* [Scherl and Levesque, 1993]. The frameworks allow a user to systematically describe the effects of actions on the world and the mental states of the agents. They are able to verify properties of simple problems, however the scalability of these approaches has not been explored.

The proposed framework presented in [Dixon et al., 2007] shows how communication protocols can be modelled using temporal epistemic logic and verified some interesting properties using theorem proving techniques. The authors have shown that how to verify properties of such systems using the existing theorem prover TeMP via a translation to the monodic fragment of first-order temporal logic. While the approach has a number of advantages it also has some drawbacks, e.g., the specification for a simple systems like NSP requires many axioms and initial conditions, and the propositionalisation of the first-order axioms can result in a large specification [Dixon et al., 2007].

The approaches discussed above show how theorem proving techniques can be used to verify properties of agent-based systems. However, these approaches do not model resources (such as time, space and/or communication) explicitly.

3.2 The model theoretic approach

The model based verification approach uses model checking techniques, which are based on the semantics of the specification language. Applying model checking to a design comprises three components. First, a detailed description M (model) of the system has to be given using the description language of the model checker. Second, a property φ of the system has to be given by means of some property specification language, e.g., linear time logic (LTL) or computation tree logic (CTL). The expressive power of LTL and CTL is not comparable. There are properties that can be expressed in LTL but cannot be expressed in CTL, and vice-versa [Clarke et al., 2000, pp. 30–31]. Third, once the model M and the system property φ are given, a model checker will check whether or not $M \models \varphi$. The third phase is completely automatic. Thus the model checking problem can be stated simply as given a formula φ of some logical language and a model M, to determine whether or not φ is valid in model M. A pictorial representation of the model checking process is shown in Figure 3.1, a detailed description can be found in [Clarke et al., 2000].

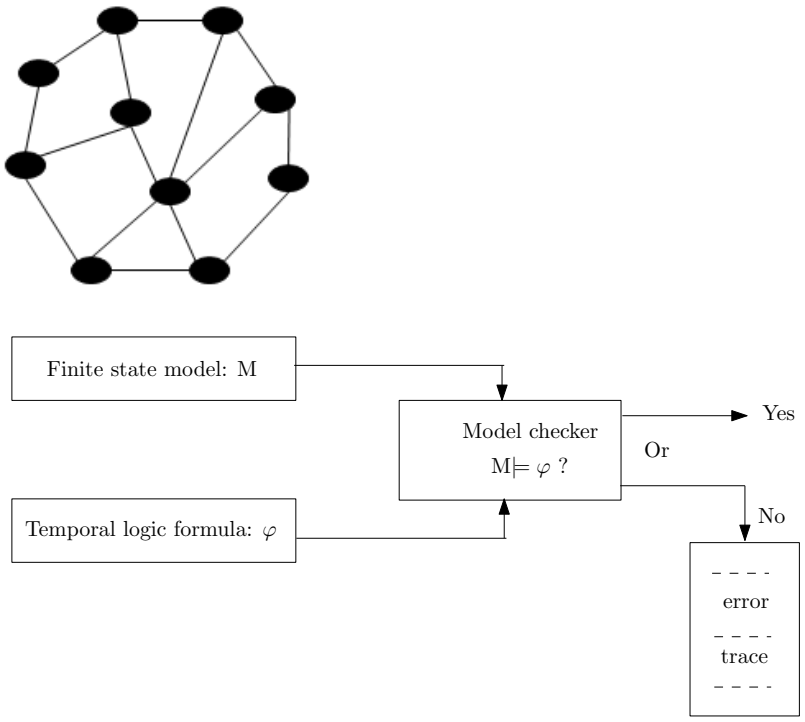


Figure 3.1: Model checking method

In the literature, different approaches to model checking have been developed; these include approaches to model checking of interpreted systems [Lomuscio et al., 2009], model checking techniques for programs [Godefroid, 1997, Visser et al., 2003, Henzinger et al., 2003], an approach to model checking of Petri nets [Gardey et al., 2005], to mention only a few of many examples. Each approach allows for a different class of systems to be analysed. For instance, some model checking approaches represent search state spaces and transitions explicitly, others express these concepts implicitly. These approaches also use different modelling techniques. For example, the approach presented in [Visser et al., 2003] translates Java programs into Promela specifications for model checking. Whereas the approach presented in [Godefroid, 1997] uses a different technique to perform model checking on a concurrent system: instead of trying to extract a model (from programs written in traditional languages like C), it explores the state-space of the system by replacing the scheduler of the concurrent system. This allows it to apply model checking to actual programs.

In the following sections, we introduce the state of the art techniques in model-

checking CTL and LTL, namely symbolic model-checking and automata-based model-checking. We then survey how model checking techniques have been used to verify properties of multi-agent systems.

3.2.1 Explicit-state model checking

In explicit-state model checking, an explicit representation of the system's global state graph is usually given by a state transition function. That is, it uses a graph to represent a Kripke structure with nodes for states and edges for transitions. Unlike symbolic model checking where model checking algorithms manipulate sets of states, in explicit-state model checking states are manipulated individually. In this approach the model checking algorithms are essentially graph search algorithms. In many practical problems the state transition graph is enormous. Therefore, when verifying properties of the systems, due to the large size of the search space it is hardly ever possible to explore the full state space.

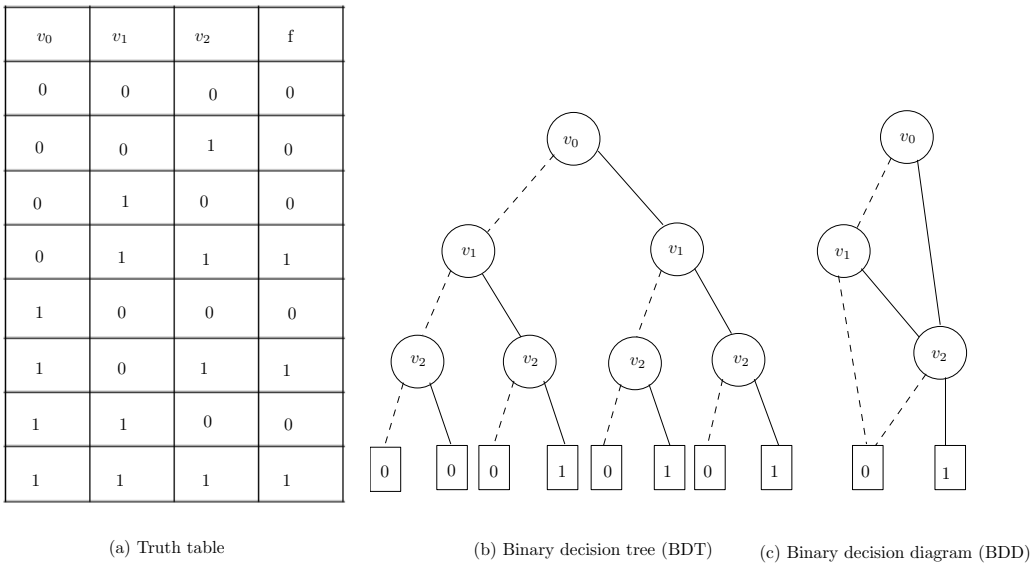
3.2.2 Symbolic model checking

In symbolic model checking the system to be verified is modelled as a finite state transition system, and the specifications are often expressed in linear or branching time temporal logic. Model checking algorithms work by exhaustively exploring the state space of the state transition system and checking automatically that the specification is satisfied. The states and the transition relations are both represented symbolically by means of Boolean formulae.

Let $V = \{v_0, v_1, \ldots, v_{n-1}\}$ be a set of n Boolean variables required to encode the system to be verified. Then a state of the system can be described by assigning values to all the variables in V, for example, $(v_0 = b_0, v_1 = b_1, \ldots, v_{n-1} = b_{n-1})$ represents a possible state of the system, where $b_i \in \{0, 1\}$. That is, a state is a mapping of state variables to values. With n variables, there are 2^n possible states of the system.

The transition relation of the system is represented by the current state and the

next sate valuation of the Boolean variables i.e., a transition is a binary relation on states. When exploring the state space of the model to be verified, the model checker iteratively applies the transition relation to the current state resulting in the next state which is known as image computation. The algorithms for symbolic model checking are implemented entirely by manipulating Boolean formulae. Nonetheless, to make symbolic model checking practical, an efficient data structure is required to represent Boolean formulae. Reduced ordered binary decision diagrams [Bryant, 1986, 1992] serve this purpose.



v_0	v_1	v_2	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

(a) Truth table (b) Binary decision tree (BDT) (c) Binary decision diagram (BDD)

Figure 3.2: Representation of Boolean function

3.2.2.1 Representation of Boolean functions

A simple and straightforward way to encode Boolean functions is to use the truth tables. However given the exponential growth in size as the number of inputs increase, truth tables are not suitable for functions with a large number of inputs. A Boolean function $f(v_0, v_1, \ldots, v_{n-1})$ can also be represented as a Binary Decision Tree (BDT) of height n. In the BDT for $f(v_0, v_1, \ldots, v_{n-1})$ each path defines a Boolean assignment $b_0, b_1, \ldots, b_{n-1}$ for the variables of f and the leaves are labelled with the Boolean value $f(b_0, b_1, \ldots, b_{n-1})$.

As an illustration (the example using here is adapted from [Bryant, 1992]), a truth table and BDT of a Boolean function $f : \mathbb{B}^3 \rightarrow \mathbb{B}$ defined by $f(v_0, v_1, v_2) = (v_0 \vee v_1) \wedge v_2$ is depicted in Figure 3.2 (a) and (b) respectively. The internal nodes of the BDT are labelled by Boolean variables. Every internal node v_i $(0 \leq i \leq 2)$ has exactly two children. The two arcs from v_i $(0 \leq i \leq 2)$ to the children are labelled by 0 (shown as a dashed line) and by 1 (shown as a solid line). The nodes of every path in the tree have unique labels; the leaves of tree are labelled by 0 and 1.

The BDT representation of a Boolean function shows that like truth tables, the BDT is also not very compact; in fact the number of leaves is identical to the size of the truth table of f. However, BDTs may contain redundancy, which can be transformed into a more compact data structure by identifying redundancies and then eliminating them. The resulting diagram will no longer be a tree but it will become a directed acyclic graph (DAG). To identify redundancies, consider the leftmost subtree rooted at the node v_2 in Figure 3.2(b). The two children (leaves of the tree) of the node v_2 evaluate to 0. This shows that the value of the Boolean formula f is 0 independently of the test of this particular node. Therefore, these type of tests can be eliminated and whole subtree can be reduced to 0. This transformation rule is called *elimination of redundant tests*. The Figure 3.2(b) also shows that there are three identical subtrees rooted at v_2, they can be merged into one subtree which transforms the tree into a DAG, this transformation rule is called *merging isomorphic subdags*. It is obvious that applying the *merging isomorphic subdags* rule, all leaves with value $0(1)$ can be merged into a single leaf. By applying these transformation rules to a BDT, the resulting data structure is called binary decision diagram(BDD), depicted in Figure 3.2(c).

The shape and size of a BDD depend on the order of its variables. When a BDD is built for a given Boolean function, the order of selecting variables on different paths of the DAG may be different. A BDD of a Boolean function is ordered if on all paths through the DAG the variables respect a given linear order say, $\rho : v_0 < v_1 < \ldots < v_{n-1}$. The size of the BDD is determined both by the Boolean function being

represented and the chosen ordering of the Boolean variables. Different orders can result in a dramatic increase or decrease in size of the BDDs [Bryant, 1992]. When the ordering of the variables is fixed in advance for a Boolean function, there is a unique reduced ordered BDD corresponding to this order. The following theorem due to Bryant [Bryant, 1986] proves a key property that reduced ordered BDDs form a canonical representation for Boolean functions.[1]

Theorem [Canonicity of BDD] *"For any Boolean function f, there is a unique(up to isomorphism) reduced ordered binary decision diagram"*.

3.2.2.2 Kripke model to BDDs and reachability analysis

In this section, we consider a simple case and show how Kripke models can be represented as BDDs. Let us consider the Kripke model depicted in Figure 3.3, where the system consists of two states s_0 and s_1 and three possible transitions (s_0, s_1), (s_1, s_0) and (s_1, s_1). It is easy to see that the system can be encoded using a single Boolean variable say v_0 such that the valuation $v_0 = 0$ represents state s_0 and the valuation $v_0 = 1$ represents state s_1. In order to represent transition relation, two sets of variables $\{v_0\}$ and $\{v_0'\}$ are required containing the variable v_0 as current state variable and v_0' as the next state variable, where $v_0' = next(v_0)$. If there is a transition, e.g., from state $s_0 \equiv (v_0 = 0)$ to state $s_1 \equiv (v_0 = 1)$, then the corresponding Boolean function can be represented as $\neg v_0 \wedge v_0'$. The disjunction of all these transitions form the transition relation of the model. Therefore, the transition relation of this model can be represented as $\mathcal{T}(\, v_0, v_0' \,) := (\neg v_0 \wedge v_0') \vee (v_0 \wedge \neg v_0') \vee (v_0 \wedge v_0')$. The corresponding BDDs representation of the transition relation $\mathcal{T}(\, v_0, v_0' \,)$ is shown at the bottom of the Figure 3.3.

The model-checking algorithm for CTL works by annotating states by sub-formulae of the formula to be verified, starting from simpler sub-formulae. For this, we need to recursively compute the set of states reachable from a given state. For example, if the

[1]Reduced ordered binary decision diagram: from now on through out the thesis, we shall use the only abbreviation BDD to mean reduced ordered binary decision diagram.

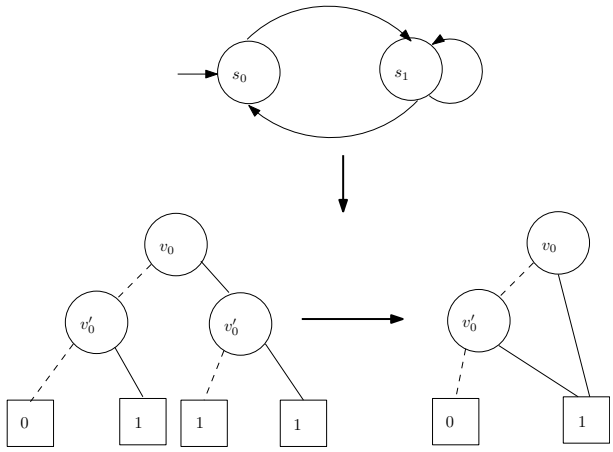


Figure 3.3: Boolean representation of Kripke model

subformula we are working with is $EU(\varphi_1, \varphi_2)$ and the states are already annotated with φ_1 and φ_2, then we start with a set of states satisfying φ_2, and keep annotating the states which are predecessors of those states and satisfy φ_1 with $EU(\varphi_1, \varphi_2)$. The standard way to compute it is to use the fixed point iteration algorithm implemented in terms of basic BDD operations. The main idea is to stay at the BDD level when finding the next states of a set of states. This can be done by the image computation of a set of states reachable in at most one step. Consider a system M whose state can be encoded by n Boolen variables $V = \{v_0, v_1, \ldots, v_{n-1}\}$. A set S_i of states of M can be viewed as a Boolean function of variables in V. Let $V' = \{v_0', v_1', \ldots, v_{n-1}'\}$ be the set of next-state variables and $\mathcal{T}(\vec{v}, \vec{v}')$ be the transition relation, where $\vec{v}$ and $\vec{v}'$ are the vectors of the current state and next state variables, respectively. The transition relation is thus a Boolean function over $2n$ variables. Then the image computation can be defined as $Img(S_i) = (\exists \vec{v} \cdot S_i(\vec{v}) \wedge \mathcal{T}(\vec{v}, \vec{v}'))[\vec{v}/\vec{v}']$, where $[\vec{v}/\vec{v}']$ denotes the remaining operation of replacing each next-state variable v_i' by the corresponding current-state variable v_i, $S_i(\vec{v})$ represents the BDDs of the set of states reached in i or fewer steps, and $\mathcal{T}(\vec{v}, \vec{v}')$ represents the BDDs of the transition relation. This operation can be performed as a single step. In this case the transition relation is said to be monolithic, and consists of a single BDD. Or the transition relation can be built as a list of small BDDs, called partitioned transition relations $\mathcal{T}_i,\ i \geq 0$, which are implicitly disjoined (asynchronous model of concurrency) or conjoined (synchronous systems). A more

detailed discussion of the implementation can be found in [Cimatti et al., 2000].

3.2.3 The model checking complexity of CTL

We state some well established theorems [Clarke et al., 2000, pp. 35–41][Baier and Katoen, 2008, pp. 355–381], which give the complexity of CTL model checking as well as the complexity of counterexample generation.

Theorem [Complexity of CTL Model Checking] *"Given a transition system $\mathcal{M}$ and CTL formula φ, there is an algorithm for the CTL model checking problem $\mathcal{M} \models \varphi$ that runs in time $O(|\varphi|.(|\mathcal{S}| + |\mathcal{T}|))$".*

Theorem [Complexity of Counterexample Generation] *"Given a transition system $\mathcal{M}$ and a CTL path formula Φ. If $\mathcal{M} \not\models A\Phi$, then a counterexample for Φ in $\mathcal{M}$ can be determined in time $O(|\mathcal{S}| + |\mathcal{T}|)$. The same holds for a witness for Φ, provided that $\mathcal{M} \models E\Phi$".*

This is the optimal result: CTL model checking is P-space complete, see for example [Schnoebelen, 2002].

3.2.4 Automata-based model checking

In this approach, the system to be verified is modelled as a finite state transition system $\mathcal{M}$ and the property to be verified is expressed as a formula φ of linear temporal logic (LTL). The first step of model checking is to translate both the model $\mathcal{M}$ and the negation of the specification $\neg\varphi$ into Büchi automata. Then the model checking problem is seen as an emptiness problem for the product of these two automata.

3.2.4.1 Büchi automata and LTL model checking

Finite-state automata accept finite words, i.e., sequences of symbols of finite length. In practice we often model concurrent systems which show infinite behaviours, which cannot be represented using finite state automata. A variant of finite-state automata

known as Büchi automata that accepts an infinite input sequence can be used to represent finite-state systems. Büchi automata have the same syntactic structure as finite state automata but they have a different acceptance condition. Let Σ be a finite alphabet. An infinite word or ω-word over Σ is simply an infinite sequence $w = a_1 a_2 \ldots$ for $a_i \in \Sigma$. Let Σ^ω denote the set of all infinite words over the alphabet Σ. Let $\mathcal{A} = (\Sigma, S, \delta, S_0, F)$ be a finite automaton, where S is the set of states, and $S_0 \subseteq S$ is the set of initial states, $\delta \subseteq S \times \Sigma \times \wp(S)$ is the transition relation, and F the set of final states. Since we are interested in the infinite behaviour of the system, $\mathcal{A}$ has the following definition of acceptance.

Definition 3.2.1. *A run over an infinite word $w = a_1 a_2 \ldots$ (for $a_i \in \Sigma$) is a sequence of states $\bar{s} = s_0 s_1 s_2 \ldots$ such that $s_0 \in S_0$ and $s_i \in \delta(s_{i-1}, a_i)$ for all $i \geq 1$. Let $infinite(\bar{s}) = \{s \mid s$ appears infinitely often on $\bar{s}\}$, then run $\bar{s}$ of $\mathcal{A}$ is said to be accepting iff $infinite(\bar{s}) \cap F \neq \emptyset$.*

The finite automaton $\mathcal{A}$ with the above acceptance condition is called a Büchi automaton. Therefore, a Büchi automaton is a finite automaton $\mathcal{A} = (\Sigma, S, \delta, S_0, F)$, and the language accepted by $\mathcal{A}$ is $L(\mathcal{A}) = \{w \mid$ there is a run $\bar{s}$ over w such that $infinite(\bar{s}) \cap F \neq \emptyset\}$. In automata-based model checking we are interested in determining whether the system $\mathcal{M}$, represented by Büchi automaton $\mathcal{A}_\mathcal{M}$, satisfies a (LTL) property specification φ, represented by another Büchi automaton $\mathcal{A}_{\neg\varphi}$. $\mathcal{A}_{\neg\varphi}$ can be automatically derived from a given LTL formula φ. The model checking problem is to check whether $\mathcal{M}$ satisfies φ iff the Büchi automaton $\mathcal{A}_\mathcal{M}$ satisfies $\mathcal{A}_{\neg\varphi}$. Now $\mathcal{A}_\mathcal{M}$ satisfies $\mathcal{A}_{\neg\varphi}$ iff $L(\mathcal{A}_\mathcal{M}) \subseteq L(\mathcal{A}_{\neg\varphi})$. Since Büchi automata are closed under complement and intersection, $L(\mathcal{A}_\mathcal{M}) \subseteq L(\mathcal{A}_{\neg\varphi})$ iff $L(\mathcal{A}_\mathcal{M}) \cap \overline{L(\mathcal{A}_{\neg\varphi})} = \emptyset$, where $\overline{L(\mathcal{A}_{\neg\varphi})}$ is the complement of $L(\mathcal{A}_{\neg\varphi})$.

3.2.4.2 Complexity of model-checking LTL using automata

The complexity of LTL model checking using automata is given by:

Theorem [Complexity of LTL model checking] *"Given a transition system $\mathcal{M}$ and LTL formula φ, there is an algorithm for the LTL model checking problem $\mathcal{M} \models \varphi$ that runs in time $O(|\mathcal{M}| \times 2^{|\varphi|})$"* [Lichtenstein and Pnueli, 1985].

The above theorem shows that the time complexity of LTL model checking algorithm is linear in the size of the model and exponential (in the worst case) in the size of the formula. Sistla and Clarke show in [Sistla and Clarke, 1985] that LTL model checking is P-space complete.

3.2.5 Discussion

3.2.5.1 Discussion of symbolic model checking

Symbolic model checking is a powerful formal specification and verification method, and it is regarded as the state-of-the-art technology for verifying finite state concurrent systems. It allows a very compact representation of states and state transition relations. It has the ability to execute at the same time all transitions enabled at the initial states and manipulate sets of states effectively. However, in model checking state space traversal is the main computational bottleneck. Despite considerable efforts, symbolic model checking techniques suffer from the state-space explosion problem. As explained above, computing reachable state sets from a given state transition graph is one of the main components of symbolic model checking. A well known problem with BDD-based reachability searches is the size of the BDDs. In many cases the size of the intermediate BDDs during image computation become very large [Burch et al., 1993], and the system blows up while computing the reachable state space.

In order to reduce BDD size, researchers have developed various algorithms which are used based on the application model during image computation, e.g., monolithic, partitioned transition relation, early quantification etc. [Burch et al., 1993]. However, "even using all these state-of-the art algorithms the size of the intermediate BDDs and the BDDs representing the reachable state set tend to behave as shown in Figure 3.4" [Bryant and Meinel, 2002].

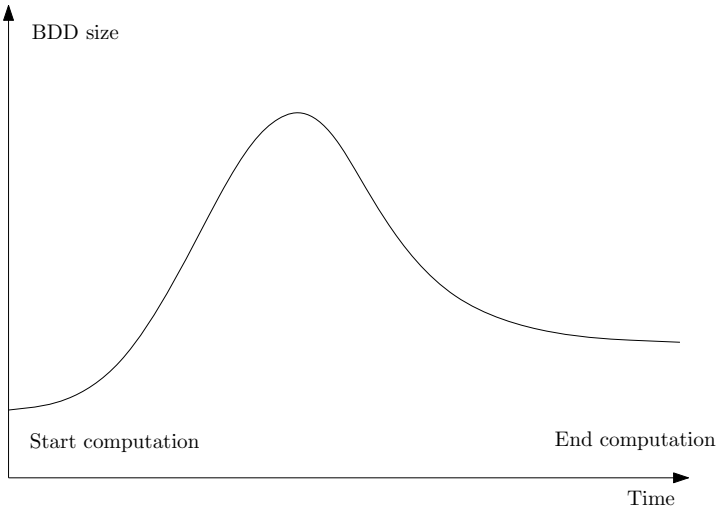


Figure 3.4: Memory consumption during image computation

3.2.5.2 Discussion of automata-based model-checking

In practice explicit-state model checking, including automata-based model checking, works efficiently if the number of reachable states is small. However, to handle large search spaces techniques exist to prune the search, for example, partial order reduction, as implemented in the SPIN model checker. It is possible in many cases to avoid constructing the entire state space of the system to be verified. This is because the states of the automaton $\mathcal{A}_{\mathcal{M}}$ are generated only when needed, while checking the emptiness of its intersection with the property automaton $\mathcal{A}_{\neg\varphi}$. Many state-of-the-art explicit-state model checking tools, including SPIN [Holzmann, 1997], the Maude LTL model checker [Eker et al., 2003], perform the verification using double depth-first-search algorithms presented in [Courcoubetis et al., 1992], after translating the system and its properties into Büchi automata.

3.2.5.3 Symbolic vs. automata-based model-checking

The choice between symbolic and explicit-state model checking may depend on the system being verified. It has been argued that symbolic model checking performs better for synchronous systems, whereas explicit-state model checking is better for asynchronous systems [Hu et al., 1994][Magazzeni, 2009, pp. 13]. However, Eisner and Peled [Eisner and Peled, 2002] have reported that symbolic model checking per-

forms better even for asynchronous systems. In this thesis, we use both symbolic and explicit-state model checking approaches.

3.2.5.4 CTL vs. LTL

We use both LTL and CTL logics to formalise desired properties of the system to be verified. The time complexity of CTL model checking algorithm is linear in the state space of the system model and the formula. In contrast, the time complexity of LTL model checking algorithm is linear in the size of the model and exponential (in the worst case) in the size of the formula. However, Vardi [Vardi, 2001] argues that: "usually the size of the formula is much smaller than the size of the transition system. So the exponential complexity in the size of the formula is not very significant in practice". Emerson and Lei show in [Emerson and Lei, 1987] that any linear logic can be extended to a branching logic that can be decided with the same complexity.

3.2.6 Model checking techniques for MAS

In recent years there has been considerable work on model checking temporal epistemic (doxastic) aspects of multi-agent systems using symbolic and explicit state (and other) techniques. In this section, we briefly review the state-of-the-art model checking techniques for the verification of multi-agent systems. This includes the use of explicit-state (as implemented, e.g., in the SPIN or JPF [Visser et al., 2003]) model checkers) and symbolic (as implemented, e.g., in the Mocha model checker) approaches for model checking multi-agent systems.

3.2.6.1 Model checking agent programs

In [Wooldridge et al., 2002], Wooldridge et al. have developed model-checking techniques for the verification of agent-based systems, where each agent has a mental state consisting of BDIs (beliefs, desires and intentions)[2]. The proposed framework intro-

[2]Rao and Georgeff [Rao and Georgeff, 1991] have developed a logical theory based on a possible worlds approach for deliberation by agents based on three mental attitudes beliefs, desires, and inten-

duces a simple imperative language MABLE and a specification language MORA, a simplified form of LORA [Wooldridge, 2000]. A MABLE system consists of a set of agents where each agent is programmed using the MABLE language. The syntax of MORA combines temporal logic and dynamic logic, and incorporates modalities for beliefs, desires and intentions. The MABLE compiler takes as input a MABLE system and properties of the system to be verified described in MORA, and generates an encoding of the system for SPIN model checker, allowing properties of the system to be verified. The description of the MABLE system translated into PROMELA (specification language of the SPIN) and the property of the system specified in MORA is translated into LTL used by SPIN for model checking. Many interesting properties that can be specified in MORA and verified them (after translating) using SPIN include for example "some agent i eventually comes to believe that agent j intends that i believes the formula φ".

In [Bordini et al., 2003], Bordini et al. have developed model-checking techniques that apply directly to multiagent programs written in AgentSpeak(F). The basic difference between MABLE and AgentSpeak(F) is that MABLE is an imperative programming language where as AgentSpeak(F) is a logic programming language. The aim of the research [Bordini et al., 2003] is to facilitate model checking of AgentSpeak(L) systems introduced by Rao in [Rao, 1996]. In order to reach that goal the first step was to introduce AgentSpeak(F), a finite sate version of AgentSpeak(L). The language AgentSpeak(F) is then used to formalise a multi-agent system expressed using Rao and Georgeff's BDI logic. The next step is translate AgentSpeak(F) programs into PROMELA and BDI specification into an LTL specification, so that the system can be model checked using the model checker SPIN. The translation process is performed automatically by a translator called CASP (Checking AgentSpeak Programs).

In [Dennis et al., 2008a] a framework for the verification of MAS is proposed that incorporates agents programmed in several programming languages. The authors have

tions. These mental attitudes represent the informational state, motivation state, and deliberative state of an agent.

developed the AIL (Agent Infrastructure Layer) toolkit so that new agent programming languages can easily be incorporated into the AJPF (an extended version of JPF) model checking architecture. The basic idea for the development of AIL was to use the semantic rules presented in [Dennis et al., 2008b], capturing all major features of common BDI languages. A number of popular BDI languages have been considered, including variant of AgentSpeak [Rao, 1996] used in $Jason$ [Bordini et al., 2007b] and $3APL$. Other agent languages that have been taken into account include Jadex [Braubach and Lamersdorf, 2005] and Concurrent METATEM [Fisher, 1993]. Properties can be specified independently of any programming language specific requirements and are given using modal temporal logic at the AIL level. Interesting properties of systems that can be specified and verified include, e.g., $\Diamond(B(i, pickup))$ ("eventually, agent i believes that the object is picked up").

3.2.6.2 Model cheking techniques for interpreted systems

Interpreted systems [Fagin et al., 1995] are considered by many as a prime example of computationally grounded models of multi-agent systems (see e.g., [Lomuscio et al., 2009]). A strand of work on model-checking techniques for MAS based on interpreted systems is carried out by various researchers, including those presented below.

In [van der Hoek and Wooldridge, 2002] a framework is proposed for model checking temporal logic of knowledge (CKL_n). CKL_n combines LTL with epistemic logic. The underlying theory of the framework is interpreted systems. Each agent i in the system is characterised by a finite set of local states L_i and a finite set of actions A_i. Actions are performed non-deterministically using a protocol $P_i : L_i \rightarrow \wp(A_i)$. A global state of a multi-agent system is represented by $\langle l_1, l_2, \ldots, l_n \rangle$, where each $l_i \in L_i$ $(1 \leq i \leq n)$ is the local state of agent i. Then $G \subseteq L_1 \times L_2 \times \ldots \times L_n$ represents the set of reachable states of the system. A run r of the system is a function from time (assumed discrete) to global states, $r : \mathbb{N} \rightarrow G$. A system model $\mathcal{R}$ is considered to be a set of runs. A pair (r, t) is a point represented by a run $r \in \mathcal{R}$ at time

$t \in \{0, 1, 2, \ldots\}$. Let $r(t) = (s_1, s_2, \ldots, s_n)$ be a global state and define $r_i(t) = s_i$ (local state). Two points (r, t) and (r', t') are indistinguishable to an agent i denoted by $(r, t) \sim_i (r', t')$ if $r_i(t) = r'_i(t')$. An interpreted system $\mathcal{I}$ is defined as a pair $(\mathcal{R}, \pi)$, where $\mathcal{R}$ is a set of runs over a set of global states and π truth assignment function, which gives the set of primitive propositions that are true at each point in $\mathcal{R}$. The relation $\sim_i$ is used to give a semantics to the knowledge modalities in CKL_n. To give a semantics to the common knowledge modality C_Γ two more relations $\sim_\Gamma^E$ and $\sim_\Gamma^C$ were introduced. If the knowledge and common knowledge modalities are omitted from the language of CKL_n then it becomes simple LTL. In case of an simple LTL formula φ interpreted at (r, t) in $\mathcal{I}$ which depends only on run r but for knowledge modalities other runs of $\mathcal{I}$ has to be considered.

The LTL satisfaction relation denoted by $\models_{LTL}$ and is defined by $\langle \mathcal{I}, (r, t) \rangle \models_{LTL} \varphi$ means that LTL formula φ is satisfied at point (r, t) in $\mathcal{I}$. The framework cannot model check directly formulae expressed in CKL_n, so CKL_n model checking is reduced to LTL model checking using the concept of a local proposition [Engelhardt et al., 1998]. A formula of the form $K_i\varphi$ (an agent i knows φ) is interpreted as : $K_i\varphi$ iff there is a proposition p local to i such that p is true, and whenever p is true, φ is also true. That is the formula $K_i\varphi$ is translated into a formula of the form $G(p \rightarrow \varphi)$ for p is an appropriate propositional formula local to agent i. The authors have shown how the properties of a bit transmission protocol system (adapted from [Meyer and van der Hoek, 1995, pp. 39–44]) can be verified using SPIN.

The problem of model checking knowledge has also been considered in [van der Meyden and Shilov, 1999]. The verified systems are modelled using the class of interpreted systems with perfect recall semantics. In this semantics an agent remembers the whole sequence of its past states. The perfect recall semantics defines a set of observation functions O_i and the perfect recall of an agent i at time t is defined by $r_i(t) = O_i(s_0).O_i(s_1) \ldots O_i(s_t)$. Agent always know the time due to it's synchronous property, which means that if $(r, t) \sim_i (r', t')$ then $t = t'$. The authors have provided

automata theoretic characterization and algorithms, and a detailed analysis of the problem of model checking. The idea is further extended [van der Meyden and Su, 2004] to provide a BDD based algorithm for the verification of synchronous systems with perfect recall. The algorithm accepts temporal epistemic formulae whose structure is of the form $X^n K_i p$, where X^n is the concatenation of n LTL next operators X and p is an atomic proposition. It has been shown that these class of formulae can be reduced to Boolean formulae using BDDs. Consequently verification of $X^n K_i p$ in synchronous interpreted systems is reduced to the verification of the equivalence of Boolean formulae. The authors have developed a system based on these ideas, and applied it to verify the dining cryptographers protocol [Chaum, 1988].

In [Penczek and Lomuscio, 2002], Penczek and Lomuscio have applied bounded model-checking techniques for the verification of MAS. The framework adapts the semantics of interpreted systems and the resulting logical formalism CTLK is a combination of CTL and an epistemic component. A computation in an interpreted system $\mathcal{I}$ is a possibly infinite sequence of global states, and a bounded computation "k-computation" is a computation of bounded length k. In a given interpreted system, a k-model is constructed by by taking all the possible runs of length k. The bounded semantics of CTLK is defined over $k\text{-}model\ M_k$. The authors have shown that for some $k \leq |M|$ (state space size of M) the validity of a formula φ of CTLK on M_k implies its validity in the standard model M and vice versa. Then it is shown that the problem of verifying a formula φ over M_k can be achieved by checking the satisfiability of a propositional Boolean formula $[M^{\varphi}]_k \wedge [\varphi]_{M_k}$. Which is a conjunction of the Boolean encoding of the model under consideration and the formula to be verified. The verified properties are, for example, $AGC\varphi$: whether a common knowledge of a fact is always true, $EG\neg C\varphi$: whether there is a path where common knowledge of a fact is not always true.

Verification of MAS via bounded model checking have also been presented in [Lomuscio et al., 2007]. In this framework, the concept of real time has been incorpo-

rated into the standard interpreted systems semantics using the timed-automata semantics [Alur et al., 1993].

3.2.6.3 Model checking resource-bounded agents

The vast majority of the literature neglects resource requirements when applying model checking techniques for the verification of multi-agent systems. However, Alechina and colleagues [Albore et al., 2006, Alechina et al., 2007] have taken some preliminary steps towards the automated verification of resource requirements of reasoning agents. In [Alechina et al., 2007], they have investigated whether an agent with a knowledge base KB, has sufficient memory to derive a given formula φ. They represent a reasoning agent as a finite state machine in which the states correspond to the formulae currently held in the reasoner's memory and the transitions between states correspond to the application of an inference rule. They have introduced a logical language in which they can formulate properties of memory-bounded reasoners, and specified and verified properties of a rule-based agent and an agent reasoning in classical logic. The properties that can be expressed include $EFB\varphi$, $EX^{\leq n}B\varphi$. In the proposed model authors recast the problem of identifying the existence of a deduction for a goal from a knowledge base for an agent as a planning problem. To check whether a reasoner has enough memory to derive a formula φ, they specify the FSM as input to the model-based planner MBP [Cimatti et al., 2003] and check whether the reasoner has a plan (a choice of memory allocations and inference rule applications), all executions of which lead to states containing φ. They have also investigated examples of trade-offs between time and memory requirements for rule-based reasoners (larger memory enables shorter derivations).

A framework for specifying and verifying systems of communicating rule-based reasoners is presented in [Alechina et al., 2006]. In order to illustrate the proposed framework, the authors have considered a very simple example consisting of two agents, and shown that if one agent asks something to another agent, then the second

agent is guaranteed to reply within a given number of inference cycles. The interesting properties that can be specified and verified include $\Box^n B_i\varphi$, where $\Box^n$ is n nesting of $\Box$. The authors have shown that such properties can be verified using existing model checking techniques, and describe an approach using the Mocha [Alur et al., 1998a] model checker.

3.2.6.4 Discussion

The verification frameworks presented in [Wooldridge et al., 2002, Bordini et al., 2003] translate the multi-agent systems specification into a SPIN specification to perform the verification. The MABLE language provides a number of agent-oriented development features: agents in MABLE have a mental state consisting of beliefs, desires and intentions, and communicate using KQML [Finin et al., 1994]-like performatives. The work [Bordini et al., 2003] introduces the main aspects of AgentSpeak(F) and its interpreter, and then addresses the use of model-checking techniques for the verification of multi-agent systems implemented in AgentSpeak(F). Both the frameworks allow system designer to formally express the system and its desired properties (as formulae of linear-time BDI logic) to be verified using model checking techniques. Bordini and colleagues [Bordini et al., 2004] extended the work on model-checking properties of agent programming languages and continued by [Bordini et al., 2006, Dennis et al., 2008a] by translating to a common underlying abstract language, Agent Infrastructure Layer , which is then translated to Java, and checked using AJPF. The architecture presented in [Dennis et al., 2008a] is much more flexible than previous approaches to model checking for agent programs. Most importantly the approach supports the verification of multi-agent systems where individual agents have been programmed in different agent languages.

The main message of [van der Hoek and Wooldridge, 2002] is that, using the concept of local propositions, the effort in model checking CKL_n formulae is reduced to LTL formulae. Thus the properties of the system can be verified using standard model

checker such as SPIN. However, the approach requires significant user intervention as the reduction method of formulae from CKL_n to LTL is manual. The main motivation of the work presented in [Penczek and Lomuscio, 2002] is to translate the formulae CTLK into propositional formulae to model check properties of systems. In bounded model checking, both the model M_k and the formula φ to be verified are translated into Boolean formulae $[M^{\varphi}]_k$ and $[\varphi]_{M_k}$ respectively. The model checking problem is then reduced to SAT-problem of verifying the satisfiability of the formula $[M^{\varphi}]_k \wedge [\varphi]_{M_k}$. The idea of [van der Meyden and Shilov, 1999, van der Meyden and Su, 2004] is also model checking formulae in the proposed logic is to reduce its equivalent Boolean formulae. However, unlike [van der Hoek and Wooldridge, 2002], [Penczek and Lomuscio, 2002] where an agent's knowledge are considered based on current observation, in [van der Meyden and Shilov, 1999, van der Meyden and Su, 2004] an agent's knowledge is based on the observations of its history. Nevertheless, while these works show a significant effort that have been made towards the verification of multi-agent systems, they adapt the classical approach of knowledge representation. Furthermore, communication mechanism is modelled via axioms that instantaneously transmit knowledge from one agent to the other. For example, $(K_i\varphi \rightarrow K_j\varphi)$—if agent i knows that φ, then agent j knows that φ instantly [Fagin et al., 1995]. However in real agent communication needs bandwidth and takes time.

The works presented in [Albore et al., 2006, Alechina et al., 2007] show how single agent systems are modelled using bounded memory logical formalisms. The frameworks [Alechina et al., 2007, Albore et al., 2006] show how to verify automatically the minimal resource(space and time) requirements to achieve a certain goal in a single agent system. Whereas in [Alechina et al., 2006], the authors have presented an interesting approach to model multi-agent interactions. The proposed framework models the communication for a particular reasoner (rule-based), the formulae of the proposed logics can be translated into the specification language of a model checker. In order to verify some interesting properties of the system automatically, the authors have used

the model checking tool Mocha. However, while these works represent a significant advance on the state-of-the-art in verifying resource bounded agents, the frameworks are unable to express the computational (memory and time) and communication resources together. Furthermore, the verification approach in [Alechina et al., 2007, Albore et al., 2006] explicitly represents the memory model as a part of the planning domain. The resulting state space explosion is therefore due in large part to the fact that a single epistemic state can be associated with several different configurations of, e.g., memory usage.

Thus, none of the approaches mentioned above allow us to express computational (memory and time) and communication resource limitations altogether, and these approaches do not allow the verification of multi-agent systems considering the interaction between different resources (time, memory and communication bandwidth). Hence there is a need to define frameworks for verifying systems considering the interaction between different resources (time, memory and communication bandwidth). In the subsequent chapters we will propose some frameworks for analysing resource requirements for systems of reasoning agents, and investigating trade-offs between multiple resource bounds.

3.3 The choice of verification approach

There are many practical advantages and disadvantages to both proof theoretic and model theoretic approaches. For instance, theorem proving techniques can deal with infinite state spaces. While some theorem provers including, e.g., PVS [Owre et al., 1992] and the Coq proof assistan [Huet et al., 2009] require human intervention, there are fully automated theorem provers (for example MSPASS [Hustadt and Schmidt, 2000] and TeMP [Hustadt et al., 2004] to cite two of many) that can handle many of the logics considered in this thesis. However, we agree with Halpern and Vardi [Halpern and Vardi, 1991] who argue that: "usually very expressive logics are used to capture the agents' behaviour while modelling multi-agent systems. Thus it is harder to prove

theorems in that logic". In contrast to the theorem proving, model checking is limited to finite-state systems. It is a completely automatic approach, and if the property being verified is violated, model checkers often produce a counterexample trace showing why the specified property is not satisfied. The counterexamples are very useful in finding subtle errors in the design of the system being verified, and it can be used effectively for the purpose of system debugging. In this thesis, we use model checking techniques to verify the properties of agent-based systems.

3.4 Model checking tools

As mentioned in the previous section, we focus on the use of model checking techniques for the verification of multi-agent systems. This section reviews some of the most popular model checking tools which are often used for the verification of multi-agent systems.

3.4.1 MCK

MCK [Gammie and van der Meyden, 2004] is a model checker for the logic of knowledge, written in Haskell. The MCK system uses interpreted systems [Fagin et al., 1995] as underlying semantics, and supports both linear and branching time temporal operators. Actions and the environment may be only partially observable at each instant in time. MCK supports several different ways of defining knowledge given a description of a multi-agent system and the observations made by the agents: observation alone; observation and clock; and perfect recall of all observations. The tool uses BDDs to represent models symbolically and the system supports several different types of temporal and epistemic specifications. In the epistemic dimension, agents may use their observations in a variety of ways to determine what they know. In the temporal dimension, the specification formulae may use either LTL or CTL. The system supports different combinations of these parameters to different degrees. The input language of MCK describes the environment in which agents interact, observation functions for

the agents to define which part of the environment each agent can observe, the agents behaviour using actions, the set of initial states, fairness constraints, and formulae to be checked.

3.4.2 VerICS

VerICS [Nabialek et al., 2004] is a model checking tool for verification of timed and multi-agent systems. The tool offers three complementary methods of verification: SAT-based Bounded Model Checking (BMC), SAT-based Unbounded Model Checking (UMC), and an on-the-fly reachability checking while constructing abstract models of systems. The input specification accepted by VerICS can be represented using a subset of Estelle [Budkowski and Dembinski, 1987], which is an ISO standard specification language designed for describing distributed systems. The tool then translates them into the common format called Intermediate Language [Doroś et al., 2002]. The tool also deals with lower level descriptions of systems such as timed automata [Alur and Dill, 1990, 1994] and Petri nets [Reisig, 1985]. Systems represented in the Intermediate Language are further translated to a set of timed automata. This translation result is then fed to the other components of VerICS for performing reachability or temporal (epistemic) logic model checking. In the case of unrestricted time constrained system, the untimed automaton generated from an Intermediate Language specification is a model of a system and it is possible to apply standard model checking algorithms to it. The properties of multi-agent systems are specified in CTLpK which is an extension of CTL with past modalities and an epistemic component.

3.4.3 MCMAS

MCMAS [Lomuscio et al., 2009] is a model checker for multi-agent systems. MCMAS permits the automatic verification of specifications that use epistemic, correctness, and cooperation modalities, in addition to the standard temporal modalities. MCMAS uses BDD based symbolic model checking algorithms, but unlike MCK, the semantics does

not assume perfect recall. The specification language of MCMAS is interpreted system programming language (ISPL), a modular language inspired by interpreted systems [Fagin et al., 1995]. The tool supports various property specification languages including CTL, epistemic operators. The properties to be verified can also be specified using ATL [Alur et al., 1998b]. Given a system model in ISPL and a formula to be verified in that model, MCMAS computes the set of states in which the formula holds and compares it to the set of reachable states. The algorithms implemented to calculate this set extends the standard fix-point boolean characterization for temporal operators [Clarke et al., 2000] to epistemic, correctness, and cooperation operators.

3.4.4 DEMO

The model checker DEMO implements the dynamic epistemic logic of [Baltag and Moss, 2004]. In this 'action model logic' the global state of the multi-agent system is represented by an epistemic model (multi-agent Kripke model), and the agent's actions are represented by an action model. An action model is also based on a multi-agent Kripke frame, but instead of carrying a valuation it has a pre-condition function which assigns a precondition to each point in the action model, which stands for an atomic action [Ditmarsch et al., 2005].

3.4.5 Mocha

Mocha [Alur et al., 1998a] is a software tool for the modular and hierarchical verification of heterogeneous systems. The input language that Mocha uses for model description is reactive modules language. Unlike simple state-transition graphs, reactive modules form a compositional model in which both states and transitions are structured. Reactive modules are built from atoms, and atoms are built from variables which are the elementary particles of systems. Each specification consists of one or more modules and they are composed in parallel. Mocha implements enumerative, as well as symbolic, state-exploration algorithms and both checkers have the capability

to produce error traces. It supports three kinds of simulation, namely, random simulation, manual simulation, and game simulation. Two versions of Mocha are available: cMocha and jMocha. The property specification language of Mocha is ATL [Alur et al., 1998b] which includes CTL.

3.4.6 NuSMV

NuSMV [Cimatti et al., 2000] is a symbolic model checker which is a reimplementation of SMV[McMillan, 1992]. It implements symbolic model checking techniques for CTL and bounded model checking techniques for LTL. The specification language of NuSMV permits the definition of the temporal model in an expressive, compact and modular way. NuSMV applies symbolic techniques based on BDDs or propositional satisfiability (SAT) solvers to efficiently perform verification over large state spaces. NuSMV allows for the representation of synchronous and asynchronous finite state systems, and for the analysis of specifications expressed in CTL and LTL, using BDD-based and SAT-based model checking techniques. Heuristics are available for achieving efficiency and partially controlling the state explosion. The interaction with the user can be carried on with a textual interface, as well as in batch mode.

3.4.7 SPIN

SPIN [Eker et al., 2003] is designed for analysing the logical consistency of concurrent or distributed asynchronous software systems. SPIN verification models are focused on proving the correctness of process interactions, and they attempt to abstract as much as possible from internal sequential computations. The system models are described in a modelling language called PROMELA (a Process Meta Language), which helps to find good abstraction of system designs. SPIN supports two principal modes of operation such as simulation and verification. SPIN uses finite automata based model checking. The verification procedure is based on the reachability analysis of the automata, using an optimized depth-first-search or breadth-first-search graph traversal method.

3.4.8 Maude LTL model checker

Maude LTL model checker [Eker et al., 2003] supports on-the-fly explicit-state model checking of concurrent systems. The specification language of Maude LTL model checker is rewrite theories, which is based on the mathematical theory of rewriting logic [Meseguer, 1990, 1992]. Maude LTL model checker can model check systems whose states involve data in data types of infinite cardinality. Data types could be any algebraic data types. The only assumption is that the set of states reachable from a given initial state is finite. LTL model checking is performed by constructing a Büchi automaton from the negation of the property formula and the specified system, and lazily searching the synchronous product for a reachable accepting cycle using double depth-first algorithm presented in [Holzmann et al., 1996].

3.4.9 The choice of model checker

The above mentioned model checking tools, among others, have been extensively used for automatic verification of multi-agent systems. However it is difficult to decide which model checker is the best to use for verification of MAS. Note that since belief operators in our model to be verified are interpreted syntactically, we do not need to use a model-checker for temporal epistemic logic, e.g., MCMAS [Lomuscio et al., 2009], MCK [Gammie and van der Meyden, 2004] or others. In our research we use Mocha, NuSMV, and the Maude LTL model checker. We use the Mocha symbolic model checker due to the ease with which we can specify concurrently executing agents in reactive modules, the description language used by Mocha. NuSMV is also a state-of-the-art symbolic model checker, which has been used for verifying several problems corresponding to interesting scenarios from real-world applications [NuS]. In this thesis we consider one of the example scenarios, where agents reason using first-order rules incorporating some reasoning strategies. In order to verify properties of such systems we use Maude LTL model checker. The specification language of the Maude LTL model checker supports any algebraic data types, this simplifies modelling of the

agents' (first-order) rules and reasoning strategies.

Chapter 4

Verifying resolution-based systems

In this chapter, we present a framework for verifying systems composed of resolution-based reasoning agents, where the resources each agent is prepared to commit to a goal (time, memory and communication bandwidth) are bounded. The framework allows us to reason about and verify tradeoffs between time, memory and communication in systems of distributed reasoning agents. We consider a typical problem for distributed reasoning agents, which we can easily parameterise to increase or decrease the problem size. We then show how model checking techniques can be used to verify that the agents can achieve a goal only if they are prepared to commit certain time, memory and communication resources. We also present an analysis of the problem and its encoding complexity in terms of state space size and branching factor.

4.1 Distributed reasoners

We define the shape of a proof in terms of the maximum space requirement at any step in the proof and the number of inference steps it contains. The lower bound on space for a given problem is then the least maximum space requirement of any proof, and the lower bound on time is the least number of inference steps of any proof. In general, a minimum space proof and a minimum time proof will be different (have different shapes). Bounding the space available for a proof will typically increase the number

of inference steps required and bounding the number of steps will increase the space required. For example, a proof which requires only the minimum amount of space may require rederivation of intermediate results.

We define the bounds on a reasoning agent in terms of its available resources expressed in terms of memory, time and communication. We assume that the memory required for a particular proof can be taken to be its space requirement (e.g., the number of formulae that must be simultaneously held in memory), and the time required of a proof is taken to be the number of inference steps necessary to solve the problem. The communication requirement of a proof is taken to be the number of messages exchanged with other agents.

For a particular agent solving a particular problem, the space available for any given proof is ultimately bounded by the size of the agent's memory and the number of inference steps is bounded by the time available to the agent, e.g., by a response time guarantee offered by the agent, or simply the point in time at which the solution to the problem becomes irrelevant. The question then arises of whether a proof can be found which falls within the resource envelope defined by the agent's resource bounds.

For a single agent which processes a single goal at a time, the lower bounds on space for the goal determines the minimum amount of memory the agent must have if it is to solve the problem (given unlimited time); and the lower bound on time determines the time the agent must commit to solving the problem (given unlimited memory). In the general case in which the agent is attending to multiple goals simultaneously, the memory and time bounds may be given not by the environment, but by the need to share the available resources between multiple tasks. For example, the agent may need to share memory between multiple concurrent tasks and/or devote no more than a given proportion of CPU to a given task. In both cases, the agent designer may be interested in tradeoffs between resource bounds; for example, whether more timely responses can be provided by pursuing fewer tasks in parallel (thereby making more memory available to each task) or whether more tasks can be pursued in parallel if each task is

allowed to take longer.

In the distributed setting we distinguish between symmetric problem distributions, where all agents have the same premises, and asymmetric problem distributions where different premises may be assigned to different agents. We also distinguish between homogeneous reasoners (when all agents have the same rules of inference and resource bounds) and heterogeneous reasoners, (when different agents have different rules of inference and/or resource bounds).

Distribution does not necessarily change the shape (maximum space requirement and number of inference steps) of a proof. However, in a distributed setting the tradeoffs between memory and time bounds are complicated by communication. Unlike memory and time, communication has no direct counterpart in the proof. However like memory, communication can be substituted for time (e.g., if part of the proof is carried out by another agent), and, like time, it can be substituted for memory (e.g., if a lemma is communicated by another agent rather than having to be remembered). In the distributed setting, each agent has a minimum memory bound which is determined by its inference rules and which may be smaller than the minimum space requirement for the problem. If the memory bound for all agents taken individually is less than the minimum space requirement for the problem, then the communication bound must be greater than zero. If the memory bound for all agents taken together is less than the minimum space requirement for the problem, then the problem is insoluble for any communication bound. With a symmetric problem distribution, if the memory bound for at least one agent is greater than the minimum space requirement for the problem, the minimum communication bound is zero (with unbounded time). If the problem distribution is asymmetric, i.e., not all agents have all the premises, then the lower bound on communication may again be non-zero, if a necessary inference step requires premises from more than one agent.

In the next section, we present measures of space, time and communication for distributed reasoning agents which allow us to make these tradeoffs precise.

4.2 Measuring resources

We consider a distributed system consisting of n_{Ag} reasoning agents. Each agent i of the system has a set of inference rules R_i (for example, R_i could contain conjunction introduction and modus ponens, or it could contain just a single rule of resolution) and a set of premises or a knowledge base KB_i. Here we consider a system of reasoning agents which reason using resolution. However, in [Alechina et al., 2008a] we present a general framework which can be applied to different kinds of reasoners. For a single agent, the notion of a derivation, or a proof of a formula G from KB_i is standard, and the time and space complexity of proofs are well studied [Haken, 1984]. Our model of space complexity is based on [Alekhnovich et al., 2002]. We view the process of producing a proof of G from KB_i as a sequence of configurations or states of a reasoner, starting from an empty configuration, and producing the next configuration by one of the following operations:

- **Read** copies a formula from KB_i into the current configuration, possibly overwriting non-deterministically a formula from the previous configuration;
- **Infer** applies a rule from R_i to formulae in the current configuration, possibly overwriting non-deterministically a formula from the previous configuration.

The sequence of configurations constitutes a proof of G if G appears in the last configuration. Time complexity corresponds to the length of the sequence, and space complexity to the size of configurations.[1] The size of a configuration can be measured either in terms of the number of formulae appearing in the configuration or in terms of the number of symbols required to represent the configuration. We take the size of a configuration to be the maximal number of formulae, where counting formulae results in non-trivial space complexity [Esteban and Torán, 1999]. Table 4.1 illustrates the

[1] We deviate from [Alekhnovich et al., 2002] in that we do not have an explicit **Erase** operation, preferring to incorporate erasing (overwriting) in the **Read** and **Infer** operations. This obviously results in shorter proofs; however including an explicit erase operation gives proofs which are no more than twice as long as our proofs if we don't require the last configuration to contain only the goal formula.

(non-trivial) space complexity of resolution proofs in terms of the number of formulae in a configuration. The example, which is due to [Esteban and Torán, 1999], shows the derivation of an empty clause by resolution from the set of all possible clauses of the form

$$\sim A_1 \vee \sim A_2 \vee \ldots \vee \sim A_n$$

where, $\sim A_i$ is either A_i or $\neg A_i$, for $n = 2$. This is known as tree-like resolution, whose clauses are all possible combinations of literals with the restriction that each variable appears once in each clause. Its space usage is 3 and the length of the proof is 8.

#	Configuration	Operation
1	$\{\,\}$	
2	$\{A_1 \vee A_2\}$	**Read**
3	$\{A_1 \vee A_2, \neg A_1 \vee A_2\}$	**Read**
4	$\{A_1 \vee A_2, \neg A_1 \vee A_2, A_2\}$	**Infer**
5	$\{A_1 \vee \neg A_2, \neg A_1 \vee A_2, A_2\}$	**Read**
6	$\{A_1 \vee \neg A_2, \neg A_1 \vee \neg A_2, A_2\}$	**Read**
7	$\{A_1 \vee \neg A_2, \neg A_2, A_2\}$	**Infer**
8	$\{\emptyset, \neg A_2, A_2\}$	**Infer**

Table 4.1: Example derivation using resolution

In the multi-agent case, when several reasoners can communicate to derive a common goal, an additional resource of interest is how many messages the reasoners must exchange in order to derive the goal. In the distributed setting, we assume that each agent has its own set of premises and inference rules and its own configuration, and that the reasoning of the agents proceeds in lock step. In addition to **Read** and **Infer**, each reasoner can perform two extra operations:

- **Idle** which leaves its configuration unchanged;
- **Copy** if agent i has a formula φ in its current configuration, then agent j can copy it to its next configuration (possibly overwriting non-deterministically a formula from the previous configuration).

The goal formula is derived if it occurs in the configuration of one of the agents. Our model of communication complexity is based on [Yao, 1979], except that we count the number of formulae exchanged by the agents rather than the number of bits exchanged. The communication complexity of a joint derivation is then the (total) number of **Copy** operations in the derivation.

	Agent 1		Agent 2	
#	Configuration	Operation	Configuration	Operation
1	$\{\}$		$\{\}$	
2	$\{A_1 \vee A_2\}$	**Read**	$\{A_1 \vee \neg A_2\}$	**Read**
3	$\{A_1 \vee A_2, \neg A_1 \vee A_2\}$	**Read**	$\{\neg A_1 \vee \neg A_2, A_1 \vee \neg A_2\}$	**Read**
4	$\{A_1 \vee A_2, A_2\}$	**Infer**	$\{\neg A_2, A_1 \vee \neg A_2\}$	**Infer**
5	$\{A_1 \vee \neg A_2, A_2\}$	**Read**	$\{\neg A_2, A_2\}$	**Copy**
6	$\{A_1, A_2\}$	**Infer**	$\{\{\}, A_2\}$	**Infer**

Table 4.2: Example derivation using resolution with two agents

In general, in a distributed setting, trade-offs are possible between the number of messages exchanged and the space (size of a single agent's configuration) and time required for a derivation. The total space used (the total number of formulae in all agent's configurations) clearly cannot be less than the minimal configuration size required by a single reasoner to derive the goal formula from the union of all knowledge bases using resolution, however this can be distributed between the agents in different ways, resulting in different numbers of exchanged messages. Similarly, if parts of a derivation can be performed in parallel, the total derivation will be shorter, though communication of the partial results will increase the communication complexity. As an illustration, Table 4.2 shows one possible distribution of the resolution example in Table 4.1. As can be seen, two communicating agents can solve the problem faster than a single agent. We are assuming here both the agents having the same knowledge base. It is shown in Table 4.2 that agent 1 derives formula A_2 at time step 4 and agent 2 copies it to its next configuration.

4.3 Property specification

Let us consider the example derivation shown above in Table 4.2. The resource requirements for the system to derive the goal clause $\emptyset$ are memory bound of 2 for each agent, the communication bound is 0 for the first agent and 1 for the second, and the time bound is 6 inference (time) steps. We can prove that $start \rightarrow EX^{\leq 6}[B_1\emptyset \vee B_2\emptyset]$ (i.e., from the start state, the agents can derive the empty clause in 6 timesteps), where B_i is a belief operator (discussed in the next section) for each agent i. To obtain the actual derivation we can also attempt to verify the negation of a formula, for example $AG\neg B_i\emptyset$ (for $i = 1, 2$)—the counterexample trace will show how the system reaches the state where $\emptyset$ is proved. In the following, we briefly describe a temporal doxastic logic which can be used to reason about the system.

4.4 Logical formalism

To reason about systems of distributed reasoning agents we use BMCL-CTL developed by Nga and Alechina, a temporal doxastic logic which allows us to describe a set of reasoning agents with bounds on memory and on the number of messages they can exchange. We are primarily interested in the automated verification aspects of such systems. However, in this section, we briefly discuss the syntax and semantics of BMCL-CTL, a detailed description can be found in [Alechina et al., 2009a].

The language of the logic contains belief operators B_i, for each agent i. We interpret $B_i\alpha$ syntactically (as a property of a formula φ, rather than of a proposition denoted by φ). $B_i\alpha$ is true if the formula α is in agent i's memory. This is inevitable since we consider resource-limited reasoning agents, and we cannot assume that the agents can instantaneously identify logically equivalent formulae. For the same reason, we do not interpret beliefs using an accessibility relation, since this would cause beliefs to be immediately closed under logical consequence. We also do not consider nested belief operators because we do not model agents reasoning about each other's

beliefs. However it is possible to model agents that reason using positive introspection in a similar way, see for example [Alechina et al., 2009b].

We consider a set of agents $Ag = \{1, 2, \ldots, n_{Ag}\}$ that reason using resolution. For simplicity, we assume that they agree on a finite set P of propositional variables.[2] Since each agent uses resolution for reasoning, we assume that all formulae of the internal language of the agents are in the form of *clauses*. For convenience, we define a clause as a set of *literals* in which a literal is a propositional variable or its negation. Then the set of literals is defined as $LP = \{p, \neg p \mid p \in P\}$. If l is a literal, then $\neg l$ is $\neg p$ if l is a propositional variable p, and p if l is of the form $\neg p$. Let Ω be the set of all possible clauses over LP, i.e., $\Omega = \wp(LP)$. Note that Ω is finite. The only rule of inference that each agent has is the resolution rule which is defined as follows:

$$\frac{l \in \alpha \qquad \neg l \in \beta}{(\alpha \setminus \{l\}) \cup (\beta \setminus \{\neg l\})} \text{ Res}$$

which states that if there are two clauses α and β such that one contains a literal l and the other contains $\neg l$, then we can derive a new clause $(\alpha \setminus \{l\}) \cup (\beta \setminus \{\neg l\})$. Each agent i has a memory of size $n_M(i)$ where one unit of memory corresponds to the ability to store an arbitrary clause. Each agent i has a knowledge base or a set of premises $KB_i \subseteq \Omega$ and can read clauses from KB_i by performing **Read** action. The communication ability of the agents is expressed by the **Copy** action which copies a clause from another agent's memory. The limit on each agent's communication ability is $n_C(i)$: in any valid run of the system, agent i can perform at most $n_C(i)$ **Copy** actions.

4.4.1 Syntax of BMCL-CTL

The syntax of BMCL-CTL is defined inductively as follows:

- $\top$ is a well-formed formula (wff) of BMCL-CTL;

[2] This assumption can easily be relaxed, so that only some propositional variables are shared.

- if α is a clause, then $B_i\alpha$ is a wff of BMCL-CTL for all $i \in Ag$;
- $c_i = n$ is a wff of BMCL-CTL, for all $i \in Ag$ and $n \in \mathbb{N}$;
- if φ and ψ are wff, then so are $\neg\varphi$, $\varphi \wedge \psi$;
- if φ and ψ are wff, then so are $EX\varphi$, $E(\varphi U\psi)$, and $A(\varphi U\psi)$.

Classical abbreviations for $\vee$, $\rightarrow$, $\leftrightarrow$ and $\perp$ are defined as usual. The language contains both temporal and doxastic modalities. For the temporal part of BMCL-CTL, we have CTL, a branching time temporal logic. Intuitively, CTL describes infinite trees, or all possible runs of the system, over discrete time. In the temporal logic part of the language, X stands for next step, U for until, A for 'on all paths' and E for 'on some path'. We also use abbreviations for other usual temporal operators AX, EF, AF, EG, and EG in which F stands for 'some time in the future' and G for 'always from now'. The doxastic part of the language consists of belief modalities $B_i\alpha$. For convenience, we define the following sets:

$$B_i\Omega = \{B_i\alpha \mid \alpha \in \Omega\},\ B\Omega = \bigcup_{i \in AG} B_i\Omega,$$

$$CP_i = \{c_i = n \mid 0 \leq n \leq n_C(i)\}, \text{ and } CP = \bigcup_{i \in AG} CP_i.$$

4.4.2 Semantics of BMCL-CTL

The semantics of BMCL-CTL is defined by BMCL-CTL transition systems. A BMCL-CTL transition system $M = (S, R, V)$ is defined as follows:

- S is a non-empty set of states;
- $R \subseteq S \times S$ is a total binary relation, that is for all $s \in S$, there exists $s' \in S$ such that $(s, s') \in R$;
- $V : S \times Ag \rightarrow \wp(\Omega \cup CP)$; we define the 'belief part' of the assignment $V^B(s, i) = V(s, i) \backslash CP$ and the communication counter part $V^C(s, i) = V(s, i) \cap CP$. V satisfies the following conditions:

i) $|V^C(s,i)| = 1$ for all $s \in S$ and $i \in Ag$;

ii) If $(s,t) \in R$ and $c_i = n \in V(s,i)$ and $c_i = m \in V(t,i)$ then $n \leq m$.

For each model $M = (S, R, V)$, a path in M is a sequence of states $(s_0, s_1, \ldots)$ in which $(s_k, s_{k+1}) \in R$ for all $k \geq 0$. The truth of a BMCL-CTL formula at a state $s \in S$ of a model $M = (S, R, V)$ is defined inductively as follows:

- $M, s \models B_i\alpha$ iff $\alpha \in V(s,i)$,
- $M, s \models c_i = n$ iff $c_i = n \in V(s,i)$,
- $M, s \models \neg\varphi$ iff $M, s \not\models \varphi$,
- $M, s \models \varphi \wedge \psi$ iff $M, s \models \varphi$ and $M, s \models \psi$,
- $M, s \models EX\phi$ iff there exists $s' \in S$ such that $(s, s') \in R$ and $M, s' \models \varphi$,
- $M, s \models E(\varphi U \psi)$ iff there exists a path $(s_0, s_1, \ldots, s_n, \ldots)$ in M with $s = s_0$ and $n \geq 0$ such that $M, s_k \models \varphi$ for all $0 \leq k \leq n-1$ and $M, s_n \models \psi$,
- $M, s \models A(\varphi U \psi)$ iff for all paths $(s_0, s_1, \ldots)$ in M with $s = s_0$, there exists $n \geq 0$ such that $M, s_k \models \varphi$ for all $0 \leq k \leq n-1$ and $M, s_n \models \psi$,

Now we describe conditions on the models. The first set of conditions refers to the accessibility relation R. The intuition behind the conditions is that R corresponds to the agents executing actions $\langle a_1, \ldots, a_{n_{Ag}} \rangle$ in parallel, where action a_i is a possible action (transition) for the agent i in a given state. The actions an agent i can perform are: $Read_{i,\alpha,\beta}$ (reading a clause α from the knowledge base and erasing β), $Res_{i,\alpha_1,\alpha_2,l,\beta}$ (resolving α_1 and α_2 on l and erasing β), $Copy_{i,\alpha,\beta}$ (copying α from another agent and erasing β), and $Idle_i$ (doing nothing), where $\alpha, \alpha_1, \alpha_2, \beta \in \Omega$ and $l \in LP$. Intuitively, β is an arbitrary clause which gets overwritten if it is in the agent's memory. If the agent's memory is full ($|V^B(s,i)| = n_M(i)$), then we require that β has to be in $V^B(s,i)$. Not all actions are possible in any given state. For example, to perform

a resolution step from state s, the agent has to have two resolvable clauses in s. The message counter of each agent i starts with the value 0 and is incremented every time i copies a clause. When the value of the counter becomes equal to $n_C(i)$, i cannot execute the **Copy** action any more. Let us denote the set of all possible actions by agent i in state s by $R_i(s)$. Below is the definition of $R_i(s)$:

Definition 4.4.1 (Available actions)**.** *For every state s and agent i,*

1. $Read_{i,\alpha,\beta} \in R_i(s)$ *iff* $\alpha \in KB_i$ *and* $\beta \in \Omega$*, or if* $|V^B(s,i)| = n_M(i)$ *then* $\beta \in V^B(s,i)$*,*

2. $Res_{i,\alpha_1,\alpha_2,L,\beta} \in R_i(s)$ *iff* $\alpha_1, \alpha_2 \in \Omega$*,* $l \in \alpha_1$*,* $\neg l \in \alpha_2$*,* $\alpha_1, \alpha_2 \in V(s,i)$*,* $\alpha = (\alpha_1 \setminus \{l\}) \cup (\alpha_2 \setminus \{\neg l\})$*;* β *is as before,*

3. $Copy_{i,\alpha,\beta} \in R_i(s)$ *iff there exists* $j \neq i$ *such that* $\alpha \in V(s,j)$ *and* $c_i = n \in V(s,i)$ *for some* $n < n_C(i)$*;* β *is as before,*

4. $Idle_i$ *is always in* $R_i(s)$*.*

Now we define effects of actions on the agent's state, i.e., the assignment $V(s,i)$.

Definition 4.4.2 (Effects of actions)**.** *For each $i \in Ag$, the result of performing an action a in state s is defined if $a \in R_i(s)$ and has the following effect on the assignment of clauses to i in the successor state t:*

1. *if a is $Read_{i,\alpha,\beta}$:* $V(t,i) = V(s,i) \cup \{\alpha\} \setminus \{\beta\}$*,*

2. *if a is $Res_{i,\alpha_1,\alpha_2,l,\beta}$:* $V(t,i) = V(s,i) \cup \{\alpha\} \setminus \{\beta\}$ *where* $\alpha = (\alpha_1 \setminus \{l\}) \cup (\alpha_2 \setminus \{\neg l\})$*,*

3. *if a is $Copy_{i,\alpha,\beta}$,* $c_i = n \in V(s,i)$ *for some* n*:* $V(t,i) = V(s,i) \cup \{\alpha, c_i = (n+1)\} \setminus \{\beta, c_i = n\}$*,*

4. *if a is $Idle_i$:* $V(t,i) = V(s,i)$*.*

Definition 4.4.3. *BMCM-CTL*$(KB_1, \ldots, KB_{n_{Ag}}, n_M, n_C)$ *is the set of models* $M = (S, R, V)$ *such that:*

1. *For every s and t, $R(s,t)$ iff for some tuple of actions $\langle a_1, \ldots, a_{n_{Ag}} \rangle$, $a_i \in R_i(s)$ and the assignment in t satisfies the effects of a_i for every i in $\{1, \ldots, n_{Ag}\}$,*

2. *For every s and a tuple of actions $\langle a_1, \ldots, a_{n_{Ag}} \rangle$, if $a_i \in R_i(s)$ for every i in $\{1, \ldots, n_{Ag}\}$, then there exists $t \in S$ such that $R(s,t)$ and t satisfies the effects of a_i for every i in $\{1, \ldots, n_{Ag}\}$,*

3. *The bound on each agent's memory is set by the following constraint on the mapping V:*

$$|V^B(s,i)| \leq n_M(i) \text{ for all } s \in S \text{ and } i \in Ag.$$

Note that the bound $n_C(i)$ on each agent i's communication ability (no branch contains more than $n_C(i)$ **Copy** actions by agent i) follows from the fact that $Copy_i$ is only enabled if i has performed fewer than $n_C(i)$ **Copy** actions in the past.

4.5 Verifying resource-bounds

It is straightforward to encode a BMCM-CTL model using a standard model checker, and to verify resource bounds using existing model checking techniques. For the examples, originally presented in [Alechina et al., 2009a], we have used the model checking tool Mocha. The specification language of Mocha is ATL, which includes CTL. We can express properties such as 'agent i may derive belief α in n steps' as $EX^{\leq n}\ tr(B_i\alpha)$ where $tr(B_i\alpha)$ is a suitable encoding of the fact that a clause α is present in the agent's memory (a detailed encoding is presented in § 4.8). To obtain the actual derivation we can verify the negation of a formula, for example $AG\ \neg tr(B_i\alpha)$, and use the counterexample trace generated by the model checker to show how the system reaches the state where α is proved.

Consider a single agent (agent 1) whose knowledge base contains all clauses of the form $\sim A_1 \vee\ \sim A_2$ where $\sim A_i$ is either A_i or $\neg A_i$, and which has the goal of

# Agents	Problem distribution	Memory	Communication	Time	Found a proof
1	Symmetric	2	–	–	No
1	Symmetric	3	–	8	Yes
2	Symmetric	2, 2	1, 0	6	Yes
2	Symmetric	3, 3	1, 0	6	Yes
2	Symmetric	3, 3	0, 0	8	Yes
2	Symmetric	2, 1	1, 1	9	Yes
2	Asymmetric	2, 2	2, 1	7	Yes
2	Asymmetric	3, 3	2, 1	7	Yes
2	Asymmetric	3, 1	1, 0	8	Yes

Table 4.3: Experimental results using two propositional variables

deriving the empty clause. We can express the property that agent 1 will derive the empty clause at some point in the future as $EF\ B_1\emptyset$. Using the model checker, we can show that deriving the empty clause requires a memory bound of 3 and 8 time steps (see Table 4.1). We can also show that these space and time bounds are minimal for a single agent; i.e., increasing the space bound does not result in a shorter proof. With two agents and a symmetric problem distribution (i.e., each agent has all the premises $\sim A_1 \vee \sim A_2$), we can show that a memory bound of 2 (i.e., the minimum required for resolution) and a communication bound of 1 gives a proof of 6 steps (see Table 4.2). Reducing the communication bound to 0 results in no proof, as, with a memory bound of 2 for each agent, at least one clause must be communicated from one agent to the other. Increasing the space bound to 3 (for each agent) does not shorten the proof, though it does allow the communication bound to be reduced to 0 at the cost of increasing the proof length to 8 (i.e., the single agent case). Reducing the total space bound to 3 (i.e., 2 for one agent and 1 for the other, equivalent to the single agent case) increases the number of steps required to find a proof to 9 and the communication bound to 1 for each agent. In effect, one agent functions as a cache for a clause required later in the proof, and this clause must be copied in both directions. If the problem distribution is asymmetric, e.g., if one agent has premises $A_1 \vee A_2$ and $\neg A_1 \vee \neg A_2$ and the other has premises $\neg A_1 \vee A_2$ and $A_1 \vee \neg A_2$, then with a memory bound of 2 for each agent, we can show that the time bound is 7, and the communication bound is 2 for the first agent and 1 for the second. Increasing the

memory bound for each agent to 3 does not reduce the time bound. However the memory bound can be reduced to 1 and the communication bound reduced to 1 for one agent and 0 for the other, if the time bound is increased to 8 (again this is equivalent to the single agent case, except that one agent copies the clause it lacks from the other rather than reading it). These tradeoffs are summarised in Table 4.3. Increasing the size of the problem increases the number of possible tradeoffs, but similar patterns can be seen to the 2-variable case. For example, if the agent's knowledge base contain all clauses of the form $\sim A_1 \vee \sim A_2 \vee \sim A_3$, then a single agent requires a memory bound of 4 and the time bound is 16 steps to achieve the goal. In comparison, two agents, each with a memory bound of 2, require 13 steps and 4 messages to derive the goal.

These examples serve to illustrate the interaction between memory, time and communication bounds, and between the resource distribution and the problem distribution. However, while these techniques work for small numbers of agents, they are unlikely to scale to large-scale systems. For example, using Mocha and the encoding above, we are unable to verify in reasonable time a single agent system whose knowledge base contain all clauses of the form $\sim A_1 \vee \sim A_2 \vee \sim A_3 \vee \sim A_4$ where $\sim A_i$ is either A_i or $\neg A_i$, and which has the goal of deriving the empty clause. In the following, we analyse the problem and its encoding complexity to better understand the scalability issues.

4.6 Analysis of the problem complexity

In this section we present an analysis of the complexity of reasoning in a distributed system for the tree-like resolution example introduced in § 4.2, in terms of its state space size and branching factor. We make the following assumptions.

(i) Only a single copy of each clause is allowed to be present in the agent's working memory, i.e., at any given time all clauses present in the working memory are distinct.

(ii) Overwrite of a memory cell will take place only after all the memory cells are

occupied.

(iii) If an agent's working memory contains a clause α, then α will not be read again if the agent's knowledge base contains α and two clauses β and γ present in the agent's working memory will not be resolved if α is the resolvent.

(iv) An agent can copy a clause α from an other agent's memory only if α is not present in its own working memory.

(v) An agent will not generate a tautology.

We also assume that the agents share a finite set of propositional variables $P = \{A_1, A_2, \ldots, A_n\}$ and each agent's knowledge base KB can contain clauses from the set of all possible clauses of the form $\sim A_1 \vee \sim A_2 \vee \ldots \vee \sim A_n$ where, $\sim A_i$ is either A_i or $\neg A_i$. Then the number of clauses in each agent's knowledge base can be at most 2^n. Let F be the set of all (tautology free) clauses which can be constructed from P. Then we can easily observe that, if P is $\{A_1\}$ then F is $\{A_1, \neg A_1, \emptyset\}$, if P is $\{A_1, A_2\}$ then F is $\{A_1 \vee A_2, \neg A_1 \vee A_2, A_1 \vee \neg A_2, \neg A_1 \vee \neg A_2, A_1, \neg A_1, A_2, \neg A_2, \emptyset\}$ and so on. Therefore, for a given P with n propositional variables, the size of the set F is 3^n. This comes from the fact that each variable appears in a clause either as a positive or negative form or it may be absent. Note that $KB \subseteq F$. We call a clause which is an element of agent's knowledge base a KB-clause.

In [Esteban and Torán, 1999] it is shown that any resolution refutation of such a tree-like example with n propositional variables requires at least space $n + 1$. Therefore in order to derive the empty clause $\emptyset$ from a given knowledge base of size 2^n the minimum memory size required by a system is $m = n + 1$ cells. For example, Table 4.1 shows that a single agent system with two propositional variables requires three memory cells to derive the empty clause $\emptyset$. In the multi-agent case with two propositional variables, to derive the empty clause $\emptyset$ at least one agent requires two memory cells and the combined space requirements is at least three memory cells. If one of them contains only a single memory cell, then it will be used as a cache by the

other agent as one can easily observe from the fact that a single memory cell agent can't infer anything.

4.6.1 An analysis of the state space

In symbolic model checking, there are two different measures of the size of a given model. One measure is the number of bits i.e., the number of Boolean variables required to represent a state. This measure provides information about the size of a single state in the model. The other measure is the number of reachable states of the system i.e., the set of states which are reachable from the initial state(s) of the system, known as the state space size of the system. A system which can be encoded using $b \in \mathbb{N}$ Boolean variables, can have at most 2^b reachable states. A tight upper bound on the size of this set can often be determined by analysing the system. We provide here the number of reachable states for the tree-like resolution example.

In the single-agent case, let $n \in \mathbb{N}$ be the number of propositional variables, $k = |KB| = 2^n$ be the size of the knowledge base, $p = |F| = 3^n$ be the size of the set of tautology free clauses, and $m = n + 1$ be the number of memory cells required for a derivation. Then the number of reachable states $\mathcal{N}$ is given by the following equation:

$$\mathcal{N} = (\sum_{i=0}^{m-1} {}^{k}C_i) + {}^{p}C_m \tag{4.1}$$

where the notation ${}^{n}C_r$ stands for "n choose r". The above equation allows us to calculate how the clauses from the set F can be chosen in different ways to represent all possible configurations (reachable states) of the agent. The first term $\sum_{i=0}^{m-1} {}^{k}C_i = {}^{k}C_0 + {}^{k}C_1 + \ldots + {}^{k}C_{m-1}$ on the right hand side of Equation 4.1 represents the configurations as follows. The first term of the summation is ${}^{k}C_0 = {}^{2^n}C_0$ which evaluates to 1, which is the initial configuration of the agent when none of the 2^n clauses from the knowledge base has been read. The second term of the summation is ${}^{k}C_1 = {}^{2^n}C_1$ which evaluates to 2^n, which are the 2^n configurations of the agent when one of the 2^n clauses from the knowledge base has been read non-deterministically. It

continues until the agent can read $(m-1)$ clauses from its knowledge base. The second term ${}^{p}C_{m}$ on the right hand side of Equation 4.1 represents all those configurations when m clauses are chosen out of 3^n clauses from the set F. Thus Equation 4.1 gives the (exact) number of reachable states of the system.

Let us consider the case of a multi-agent system, consisting of the parallel composition of n_{Ag} reasoning agents $\mathcal{A} = \{1, 2, \ldots, n_{Ag}\}$. Intuitively, the parallel composition contains all possible states and transitions that can be reached by making simultaneous transitions by each agent in the system. If $\mathcal{N}_i$ denotes the number of reachable states of agent i, then the number of reachable states $\mathcal{N}'$ of the multi-agent is obtained by their parallel composition.

$$\mathcal{N}' = \prod_{i=1}^{n_{Ag}} \mathcal{N}_i \tag{4.2}$$

We observe that $\mathcal{N}'$ is exponential in the number of agents: the parallel composition of n_{Ag} agents of state space size $\mathcal{N}$ each gives $n_{Ag}^{\mathcal{N}}$ states. Note that in the multi-agent case, if m_i denotes the memory size of agent i, then the minimum value of $\sum_{i=1}^{n_{Ag}} m_i$ must be $n+1$, and the value of at least one of the m_is must be 2 otherwise the system cannot infer anything. The following theorems give the asymptotic upper bounds on the set of reachable states.

Theorem 4.6.1. *Let ReS be a single-agent resolution-based system whose knowledge base contains all clauses of the form $\sim A_1 \vee \sim A_2 \vee \ldots \vee \sim A_n$ where, $\sim A_i$ is either A_i or $\neg A_i$, and which has the goal of deriving the empty clause. Then the upper bound on the set of reachable states of ReS is of order $O(3^{n^2})$.*

Proof. Recall that Eqn. 4.1 $(\sum_{i=0}^{m-1} {}^{k}C_i) + {}^{p}C_m$ gives the number of reachable states of the system. By substituting the values of m, k and p in terms of n we can define a function $f : \mathbb{N} \to \mathbb{N}$ by

$$f(n) = \sum_{i=0}^{n} {}^{2^n}C_i + {}^{3^n}C_{n+1}$$

Simplifying $f(n)$ gives us the following:

$$\begin{aligned}
f(n) &= \sum_{i=0}^{n} {}^{2^n}C_i + {}^{3^n}C_{n+1} \\
&= \sum_{i=0}^{n} \frac{2^n!}{i! \times (2^n - i)!} + \frac{3^n!}{(n+1)! \times (3^n - (n+1))!} \\
&= \sum_{i=0}^{n} \frac{2^n \times (2^n - 1) \times \ldots \times (2^n - (i-1))}{i!} + \frac{3^n \times (3^n - 1) \times \ldots \times (3^n - n)}{(n+1)!} \\
&\leq \sum_{i=0}^{n} \frac{2^{n \times i}}{i!} + \frac{3^{n \times (n+1)}}{(n+1)!}, \forall k \in \mathbb{N} \cdot 2^n - k \leq 2^n \text{ and } 3^n - k \leq 3^n \\
&= \sum_{i=0}^{n} \frac{2^{n \times i}}{i!} + 3^{n^2} \times \frac{3^n}{(n+1)!}
\end{aligned}$$

Hence, the function $f(n)$ has order $O(3^{n^2})$. □

Theorem 4.6.2. *Let ReM be a multi-agent resolution-based system consisting of n_{Ag} agents where each agent's knowledge base KB_i can contain clauses from the set of all possible clauses of the form $\sim A_1 \vee \sim A_2 \vee \ldots \vee \sim A_n$ where, $\sim A_i$ is either A_i or $\neg A_i$, and the agents have the goal of deriving the empty clause. Then the upper bound on the set of reachable states of ReM is of order $O(3^{n_{Ag} \cdot n^2})$.*

Proof. The proof is immediate from theorem 4.6.1 and the multiplication rule of Big-O complexity theory. □

4.6.2 An analysis of the branching factor

Given a state space graph of a system, the branching factor of a given state s is determined by the number of possible legal moves that the system make from s. Let $P = \{A_1, A_2, \ldots, A_n\}$ be a finite set of propositional variables. Let ReS be a single-agent resolution-based system whose knowledge base contains all clauses of the form $\sim A_1 \vee \sim A_2 \vee \ldots \vee \sim A_n$ where, $\sim A_i$ is either A_i or $\neg A_i$, and which has the goal of deriving the empty clause. Then ReS requires $m = n + 1$ memory cells in order to find a proof and the size of it's knowledge base is $k = |KB| = 2^n$. Let F be the set of all (tautology free) clauses which can be constructed from P. Then, for a given

P with n propositional variables, the size of the set F is 3^n. Note that $KB \subseteq F$. Let us consider a configuration $s \equiv \langle \alpha_1 \ \ \alpha_2 \ \ \ldots \ \ \alpha_m \rangle$ of the state space of ReS such that all the memory cells are occupied, where $\alpha_i \in F$ for all $i \in \{1, 2, \ldots, m\}$. At s the agent can perform at most ${}^mC_2$ resolution actions. This follows from the following observations: if any two clauses α_i and α_j can be resolved on variable A_i then they cannot be resolved on any other variable A_j $(i \neq j)$ otherwise the resolvent will be a tautology. That is resolving any two clauses that can be resolved over more than one variable always results in a tautology. Therefore, at an arbitrary state in the state space of ReS, the agent can perform at most ${}^mC_2$ resolution steps. The branching factor at s due to resolution action is $m \cdot {}^mC_2$. This is because an agent can perform ${}^mC_2$ resolution actions and put the resolvents non-deterministically into its m memory cells. Now, if $\alpha_i \notin KB$ for all $i \in \{1, 2, \ldots, m\}$, then at s the agent can perform k read actions. Therefore, the branching factor at s due to read action is $m \cdot k$. This is because the agent can read k clauses from its knowledge base and put them non-deterministically into its m memory cells. Then the worst case branching factor $\mathcal{B}$ of the search space of ReS is given by the following equation:

$$\mathcal{B} = m \cdot k + m \cdot {}^mC_2 \tag{4.3}$$

The following theorems give the asymptotic branching factors of the search space of a tree-like resolution based system.

Theorem 4.6.3. *Let ReS be a single-agent resolution-based system whose knowledge base contains all clauses of the form* $\sim A_1 \vee \sim A_2 \vee \ldots \vee \sim A_n$ *where,* $\sim A_i$ *is either* A_i *or* $\neg A_i$*, and which has the goal of deriving the empty clause. Then the worst case branching factor of the search space of ReS is of order* $O(n \cdot 2^{n+1})$.

$Proof$. Recall that $(m \cdot k + m \cdot {}^mC_2)$ gives the the worst case branching factor of the search space of ReS. By substituting the values of m and k in terms of n we can

define a function $f : \mathbb{N} \to \mathbb{N}$ by

$$f(n) = (n+1) \cdot 2^n + (n+1) \cdot \frac{(n+1) \cdot n}{2!}$$

Simplifying $f(n)$ gives us the following:

$$\begin{aligned} f(n) &= (n+1) \cdot 2^n + (n+1) \cdot \frac{(n+1) \cdot n}{2!} \\ &= n \cdot 2^n + 2^n + \frac{1}{2}(n^3 + 2n^2 + n) \\ &\leq n \cdot 2^n + n \cdot 2^n, \exists n_0 \in \mathbb{N} \cdot \forall n \geq n_0 \cdot 2^n + \frac{1}{2}(n^3 + 2n^2 + n) \leq n \cdot 2^n \\ &= n \cdot 2^{n+1} \end{aligned}$$

Hence, the function $f(n)$ has order $O(n \cdot 2^{n+1})$. □

Corollary 4.6.4. *Let ReM be a multi-agent resolution-based system consisting of n_{Ag} agents where each agent's knowledge base KB_i can contain clauses from the set of all possible clauses of the form $\sim A_1 \vee \sim A_2 \vee \ldots \vee \sim A_n$ where, $\sim A_i$ is either A_i or $\neg A_i$, and the agents have the goal of deriving the empty clause. Let each agent in the system have a knowledge base of size 2^n and memory bound of $n+1$ cells; then, the worst case branching factor of the search space of ReM is of order $O(n^{n_{Ag}} \cdot 2^{n_{Ag} \cdot (n+1)})$.*

Proof. From theorem 4.6.3, for each agent i we can define a function $f_i : \mathbb{N} \to \mathbb{N}$ by:

$$f_i(n) = (n+1) \cdot (n+1) + (n+1) \cdot 2^n + (n+1) \cdot \frac{(n+1) \cdot n}{2!}.$$

In the above defined function the additional $(n+1) \cdot (n+1)$ is due to the **Copy** action, i.e., in the worst case an agent can copy $(n+1)$ formulae from another agent's memory and put them non-deterministically into any of its $(n+1)$ memory cells. We

can now simplify $f(n)$ which gives us the following:

$$\begin{aligned} f(n) &= (n+1)\cdot 2^n + (n+1)\cdot \frac{(n+1)\cdot n}{2!} + (n+1)\cdot(n+1) \\ &= n\cdot 2^n + 2^n + \frac{1}{2}(n^3+2n^2+n) + n^2 + 2n + 1 \\ &= n\cdot 2^n + 2^n + \frac{1}{2}(n^3+4n^2+5n+2) \\ &\leq n\cdot 2^n + n\cdot 2^n, \exists n_0 \in \mathbb{N}\cdot \forall n \geq n_0 \cdot 2^n + \frac{1}{2}(n^3+4n^2+5n+2) \leq n\cdot 2^n \\ &= n\cdot 2^{n+1} \end{aligned}$$

Therefore, the proof of the corollary is immediate from the multiplication rule of the Big-O complexity theory. □

Theorem 4.6.5. *Let ReM be a multi-agent resolution-based system consisting of n_{Ag} agents where each agent's knowledge base KB_i can contain clauses from the set of all possible clauses of the form $\sim A_1 \vee \sim A_2 \vee \ldots \vee \sim A_n$ where, $\sim A_i$ is either A_i or $\neg A_i$, and the agents have the goal of deriving the empty clause. Let k_i and m_i be the size of the knowledge base and memory bound for each agent i in the system, respectively, such that $m_i \geq 2$ for all $i \in \{1, 2, \ldots, n_{Ag}\}$. Then the worst case branching factor of the search space of ReM is of order*

$$O(\prod_{i=1}^{n_{Ag}}(m_i \cdot m_i + m_i \cdot k_i + m_i \cdot {}^{m_i}C_2)).$$

Proof. The proof is immediate from theorem 4.6.3. The additional $m_i \cdot m_i$ is due to the **Copy** action, i.e., in the worst case an agent can copy m_i formulae from another agent's memory and put them non-deterministically into any of its m_i memory cells. □

4.7 Analysis of the encoding complexity

The analysis in the previous section gives theoretical lower bounds for the tree-like resolution example. However for practical verification, the problem must be encoded in the states of a model checker, and this typically entails some overhead. In this section we analyse the complexity of the tree-like resolution example for two alternative model-checker encodings: a positional encoding which tries to minimise the number of bits required to encode each state, and a non-positional encoding which requires more bits to encode each state but which gives a symmetry reduced state space. We analyse the state space size and branching factor of the problem for both encodings.

4.7.1 Positional encoding complexity

The states of the system correspond to an assignment of values to state variables in the model-checker. One possible way in which the state variables representing an agent's memory can be organized is as a collection of "cells", each holding at most one clause. Each cell can be represented by a bit vector of length $\delta = 2 \times |P|$, for example, when P contains the propositional variables A_1 and A_2 with index positions 0 and 1 respectively, the clause $A_1 \vee \neg A_2$ would be represented by two bitvectors: "10" for the positive literals and "01" for the negative literals. In this encoding the position of a memory cell is fixed and hence, it is known that, which memory cell contains which clause. Therefore, in a system consisting of n_{Ag} agents, $n_{Ag} \cdot m \cdot 2 \cdot |P|$ bits are required to represent a state when there are $|P|$ variables and each agent has m memory cells.

Reading a premise simply sets the bit vectors representing an arbitrary cell in agent i's memory to the appropriate values for the clause. Resolution can be implemented using simple bit operations on cells containing values representing clauses α and β, with the results being assigned to an arbitrary cell in agent i's memory. Communication, i.e., a **Copy** action, can be implemented by copying the values representing a clause α from a cell of agent j to an arbitrary cell of agent i. To express the communication bound, we can use a counter for each agent which is incremented each time a

copy action is performed by the agent. After the counter for agent i reaches $n_C(i)$, the Copy action is disabled. Note that, since the position of a memory cell is fixed in this encoding, the state space of the system may contain a number of symmetric states. For instance, the states $\langle A_1, A_1 \vee A_2, \neg A_2 \rangle$ and $\langle \neg A_2, A_1 \vee A_2, A_1 \rangle$ are symmetric. They contain the same set of formulae and one can be obtained from the other by permuting indices.

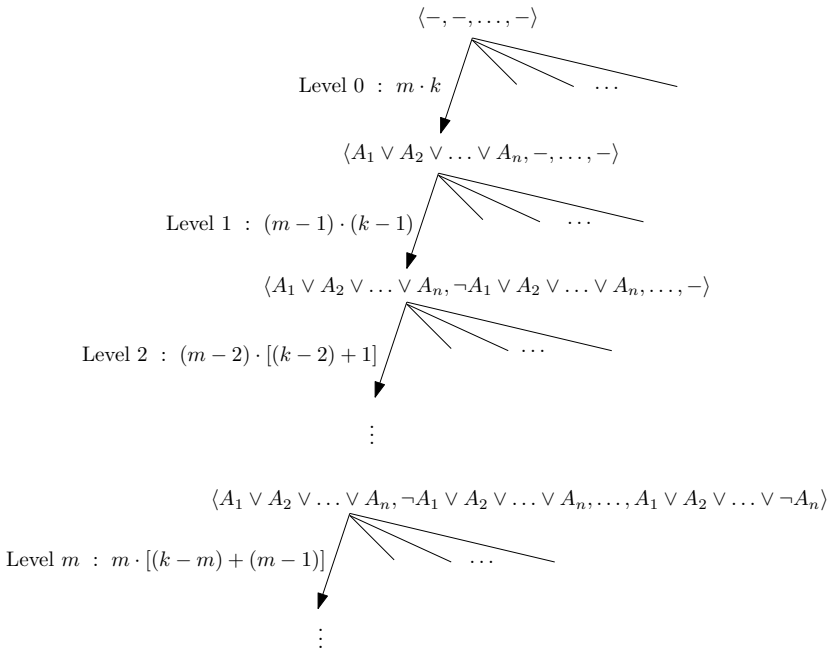


Figure 4.1: A single agent positional state branching factor model

4.7.1.1 Positional encoding analysis for a single agent system

A worst case branching factor scenario of a single agent system is shown in Figure 4.1. The initial state of the system is represented by the memory cells being empty. Since, each memory cell may contain k possible clauses that the agent can read from its knowledge base, the branching factor of the initial state is $m \cdot k$. It is easy to observe that the branching factor of any state at level 1 is $(m-1) \cdot (k-1)$, because agent can read the remaining $(k-1)$ clauses from its knowledge base and put it $(m-1)$ ways into the rest $(m-1)$ empty memory cells. At level 2, the agent can read the remaining $(k-2)$ clauses from its knowledge base and put it into $(m-2)$ ways into the remaining $(m-2)$ empty memory cells, in this state agent may also perform a resolution action.

Thus, the branching factor of a state at level 2 is $(m-2)\cdot[(k-2)+1]$. In the same way when all the memory cells at level m are fully occupied, the worst case branching factor of a given state in this level would be $m\cdot[(k-m)+(m-1)]$. This is due to the fact that agent can read its remaining $(k-m)$ clauses and put it non-deterministically to any of its m memory cells, it can also perform $(m-1)$ possible resolution actions and put the resolvents non-deterministically to any of its m memory cells. For instance, let us consider $P=\{A_1,A_2\}$ and $KB=\{A_1\vee A_2,\neg A_1\vee A_2,A_1\vee\neg A_2,\neg A_1\vee\neg A_2\}$. As shown earlier, in this case the agent requires $m=3$ memory cells to derive the empty clause. Now when the agent has read three clauses from its KB, all the three memory cells are occupied. One possible configuration of the state space could be, for example, $s\equiv\langle A_1\vee A_2,\neg A_1\vee A_2,A_1\vee\neg A_2\rangle$. At s the agent can read the remaining clause $\neg A_1\vee\neg A_2$ of its KB into its memory, it can also perform two resolution actions such as resolving the clauses $A_1\vee A_2$ and $\neg A_1\vee A_2$ or $A_1\vee A_2$ and $A_1\vee\neg A_2$. Note that at s the agent cannot perform any more resolution actions otherwise the resolvent of the two clauses will be a tautology. Similarly, when considering an arbitrary configuration of the state space, which contains any three clauses from F, the agent cannot resolve them in more than two ways. Thus the worst case branching factor at s is $3\cdot[(4-3)+(3-1)]$. In the same way, for an arbitrary number of propositional variables, it can be shown that the worst case branching factor at level m would be $m\cdot[(k-m)+(m-1)]$. After level m, in the state space, there exists a state

$$\langle A_1\vee A_2,A_1\vee\neg A_2,\ldots,\neg A_1\vee\neg A_3\vee\ldots\vee A_n,\neg A_1\vee\neg A_3\vee\ldots\vee\neg A_n\rangle$$

where the branching factor $\mathcal{B}$ is $m\cdot k+m\cdot{}^{m}C_2$. This is because the agent can read k clauses from its knowledge base and put them non-deterministically into its m memory cells, and also agent can perform ${}^{m}C_2$ resolution actions and put the resolvents non-deterministically into its m memory cells. The branching factor $\mathcal{B}$ can be expressed in terms of n as $[(n+1)\cdot 2^n+(n+1)\cdot{}^{n+1}C_2]$ which is of order $O(n\cdot 2^{n+1})$, which agrees with the result of Theorem 4.6.3. However, in this encoding the position of a memory cell is fixed, therefore the reachable state space contains a large number of symmetric

states. Apart from the initial state, every other state in the state space will be generated $m!$ times all of which give the same information. Therefore, the upper bound on the set of reachable states will be $\mathcal{N} \times (m!)$, where $\mathcal{N}$ is the reachable state space size of the problem. Thus the state space size of the positional encoding is $m!$ times bigger than the reachable state space size of the problem.

4.7.1.2 Positional encoding analysis for a multi-agent system

A worst case branching factor scenario of a multi-agent system for $n_{Ag} = 2$ and n propositional variables is shown in Figure 4.2. In this case it is assumed that both the agents have m memory cells and the same knowledge base of size $k = 2^n$. The branching factor at the initial state when all memory cells are empty is $m \cdot k \times m \cdot k$, which results from the fact that both agents can read any of their k KB-clauses and put it any of its m memory cells non-deterministically. At the next level, in addition to read actions, the agents can communicate with each other by performing **Copy** actions. Thus the branching factor of a given state in this level will be $((m-1)\cdot[(k-1)+1]) \times ((m-1)\cdot[(k-1)+1])$, where each agent can read its remaining $(k-1)$ KB-clauses and also perform a **Copy** action and put the copied clause non-deterministically into any of its $m-1$ memory cells. At level m when all the memory cells are occupied, an agent can read $(k-m)$ KB-clauses, it can perform $(m-1)$ resolution actions, and it can also perform a **Copy** action. Therefore the branching factor of a given state in this level would be $(m \cdot [(k-m)+(m-1)+1]) \times (m \cdot [(k-m)+(m-1)+1])$. As in the single-agent case there exists a state in the state space where the worst case branching factor is of order $O([m\cdot m + m\cdot k + m\cdot {}^mC_2] \times [m\cdot m + m\cdot k + m\cdot {}^mC_2])$. The additional $m \cdot m$ is due to the **Copy** action, i.e., in the worst case an agent can copy m formulae from another agent's memory and put them non-deterministically into any of its m memory cells. In this scenario the reachable state space also contains a number of symmetric states. The upper bound on the set of reachable states will be $(\mathcal{N} \times m!) \times (\mathcal{N} \times m!)$, which is greater than the reachable state space size of the problem. This model can be extended to arbitrary number of agents with different size

of memory cells and knowledge bases. In this case, the worst case branching factor will be of order $O(\prod_{i=1}^{n_{Ag}}(m_i \cdot m_i + m_i \cdot k_i + m_i \cdot {}^{m_i}C_2))$ and the upper bound on the set of reachable states will be $\prod_{i=1}^{n_{Ag}}(\mathcal{N}_i \times m_i!)$.

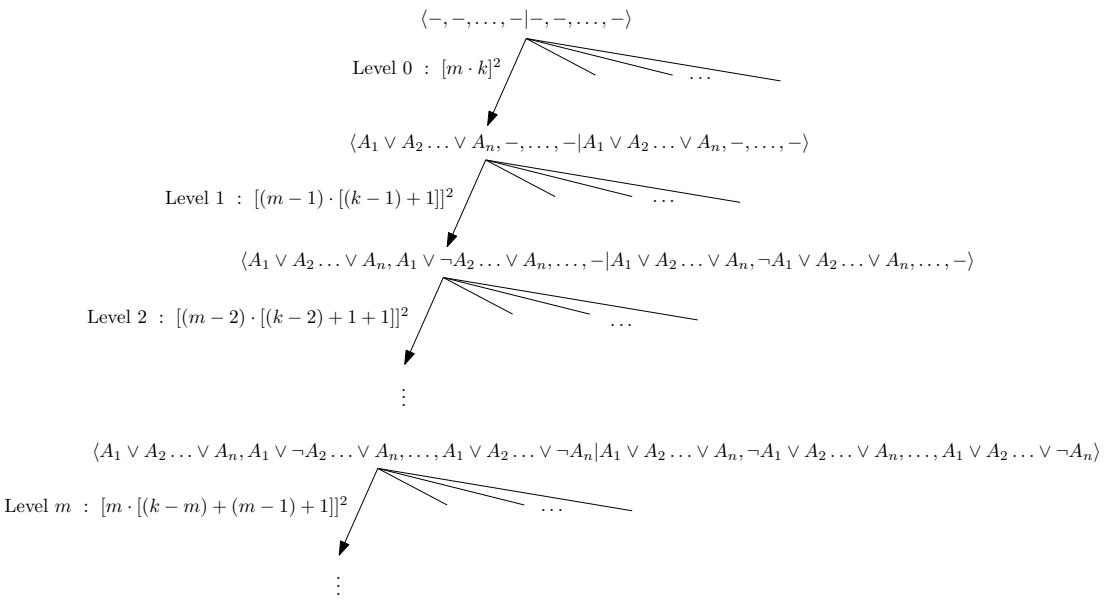


Figure 4.2: A multi-agent positional state branching factor model

4.7.2 Non-positional encoding complexity

The above analyses of positional encoding show that the encoded state space of the system is bigger than the reachable state space size of the problem. This can be avoided by a non positional encoding, where the position of a memory cell is not fixed. In this encoding, each clause of F is represented by a Boolean variable. Therefore, in a system consisting of $n_{Ag}(\geq 1)$ agents, $n_{Ag} \cdot |F|$ i.e., $n_{Ag} \cdot 3^n$ Boolean variables are required to encode each state. In this encoding we use the term *valid transition* to mean that two resolvable clauses never produce a tautology.

Let us consider a simple system consisting of a single agent with one propositional variable. The set of all possible clauses is $F = \{A_1, \neg A_1, \emptyset\}$. Therefore, three Boolean variables are required to encode the system. A state of the system is the valuation of its Boolean variables. The initial state of the system is represented by making all the

Boolean variables as false. In fact, this simple system never generates a tautology, and only possible resolution step is between the clauses A_1 and $\neg A_1$ which produces the empty clause $\emptyset$. At any given state at most m Boolean variables can be true, where m is the number of memory cells. In this encoding, the position of a memory cell is not fixed, so we do not need to care about where the clauses are stored in memory. Only the maximum number of Boolean variables which are to be true in a given state is taken into account.

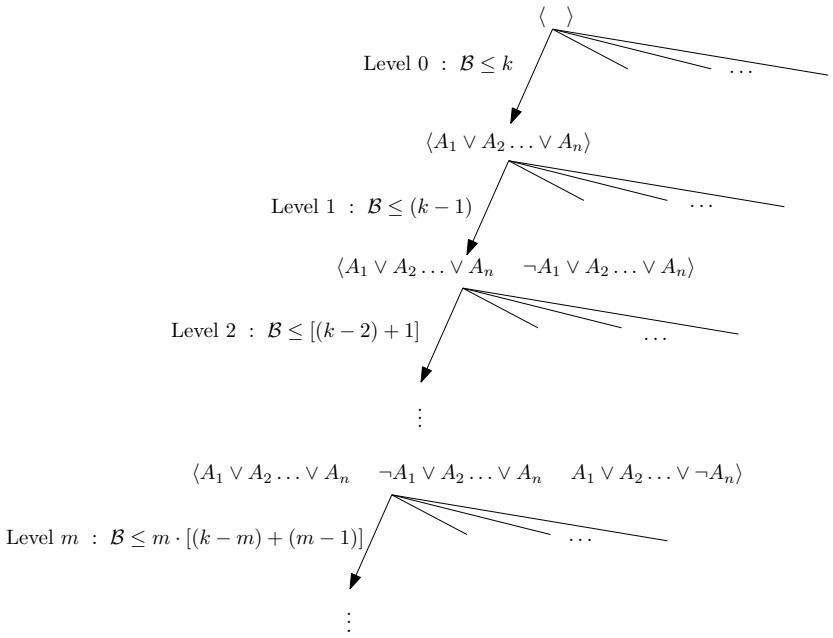


Figure 4.3: A single agent non-positional state branching factor model

4.7.2.1 Non-positional encoding analysis for a single-agent system

A possible worst case branching factor scenario of a single agent system is shown in Figure 4.3. In the initial state all memory cells are empty. Since, each memory cell may contain k possible clauses that the agent can read from its knowledge base, the branching factor of the initial state is k i.e., any of the k clauses can be true. It is easy to observe that the branching factor of any state at level 1 is $(k-1)$, because any of the remaining $(k-1)$ clauses from knowledge base can be true. At level 2, the agent can read the remaining $(k-2)$ clauses from its knowledge base as well as perform a resolution action. Thus, the branching factor of a given state at level 2 is $[(k-2)+1]$.

In the same way when all the memory cells at level m are occupied, the worst case branching factor of a given state in this level would be $m \cdot [(k-m)+(m-1)]$. This is due to the fact that agent can read its remaining $(k-m)$ clauses from its knowledge base, it can also perform $(m-1)$ possible resolution actions and overwrite any of the m clauses present in the memory. Since all the memory cells are occupied, the worst case branching factor of a state at level m or higher will be greater then or equal to $m \cdot [(k-m)+(m-1)]$. After level m, in the state space, there exists a state

$$\langle A_1 \vee A_2 \;\; A_1 \vee \neg A_2 \;\; \ldots \;\; \neg A_1 \vee \neg A_3 \vee \ldots \vee A_n \;\; \neg A_1 \vee \neg A_3 \vee \ldots \vee \neg A_n \rangle$$

where the branching factor $\mathcal{B}$ is $m \cdot k + m \cdot {}^{m}C_2$. This is because an agent can read k clauses from its knowledge base and put them non-deterministically into its m memory cells, and can also perform ${}^{m}C_2$ resolution actions and put the resolvents non-deterministically into its m memory cells. The branching factor $\mathcal{B}$ can be expressed in terms of n as $[(n+1) \cdot 2^n + (n+1) \cdot {}^{n+1}C_2]$ which is of order $O(n \cdot 2^{n+1})$, which agrees with the result of Theorem 4.6.3. Furthermore, in this encoding the position of a memory cell is not fixed, so there are no symmetric states and the reachable state space size would be $\mathcal{N}$ which is same as the problem state space size.

4.7.2.2 Non-positional encoding analysis for a multi-agent system

Let us consider a multi-agent system where $n_{Ag} = 2$ and there are n propositional variables. Assume that both the agents have m memory cells and same knowledge base of size $k = 2^n$. The branching factor of the initial state when all memory cells are empty is $k \times k$. This follows from the fact that both agents can read any of their k KB-clauses. At the next level, in addition to read actions the agents can communicate with each other by performing a **Copy** action. Thus, the branching factor of a state at this level will be $((k-1)+1) \times ((k-1)+1)$ i.e., $k \times k$. This is because each agent can read $(k-1)$ KB-clauses and can also perform a **Copy** action. At level m when all memory cells are occupied, each agent can read $(k-m)$ KB-clauses, they can perform $(m-1)$ resolution actions, and can also perform a **Copy** action. Thus the

branching factor of a given state at this level or higher would be greater than or equal to $(m\cdot[(k-m)+(m-1)+1])\times(m\cdot[(k-m)+(m-1)+1])$. As above (in the single agent scenario), in this scenario there exists a state in the state space where the worst case branching factor is of order $O([m\cdot m+m\cdot k+m\cdot {}^{m}C_2]\times[m\cdot m+m\cdot k+m\cdot {}^{m}C_2])$. The additional $m \cdot m$ is due to the **Copy** action, i.e., in the worst case an agent can copy m formulae from another agent's memory and put them non-deterministically into any of its m memory cells. Therefore, the worst case branching factor of the reachable state space follows Theorem 4.6.5, and the upper bound on the set of reachable states will be $\mathcal{N} \times \mathcal{N}$. This model can be extended to arbitrary number of agents with different size of memory cells and knowledge bases. In that case, the worst case branching factor will be of order $O(\prod_{i=1}^{n_{Ag}}(m_i \cdot m_i + m_i \cdot k_i + m_i \cdot {}^{m_i}C_2))$ and the upper bound on the set of reachable states will be $\prod_{i=1}^{n_{Ag}} \mathcal{N}_i$.

4.8 Experimental evaluation

In this section, we investigate the impact of different encodings for varying sizes of the same problem introduced in § 4.2. We report the reachable state space size, maximal BBD (MDD[3]) size of a particular iteration of image computation, and the CPU time (in seconds), required to verify properties of the system. In the largest problem that could be verified using conventional model checking techniques, each agent's knowledge base contain all clauses of the form $\sim A_1 \vee \sim A_2 \vee \sim A_3$ where $\sim A_i$ is either A_i or $\neg A_i$, and the agents have the goal of deriving the empty clause. The properties that were verified are of the form $AG\neg\emptyset$. These type of properties are useful to verify the existence of derivations. When $AG\neg\emptyset$ is false, upon analysing the counterexample trace generated by the model checker we can show how the system reaches the state

[3]As we have seen (cf. § 3.2.2.1) that each Boolean function $f : B^n \rightarrow B$ can be represented by a BDD, BDDs can be extended to represent functions $f : B^n \rightarrow \{0, \ldots, k \;\; 1\}$ and the resulting graph is called as multi-terminal BDDs. In turn, multi-terminal BDDs can be extended to multi-value decision diagrams known as MDDs which represent functions of the form $f : \{0, \ldots, k \;\; 1\}^n \rightarrow \{0, \ldots, k \;\; 1\}$. Unlike BDDs which has two outgoing edges for each internal nodes, in MDDs each internal node has k outgoing edges. The efficient operations which can be performed on BDDs can also be carried out on MDDs [Srinivasan et al., 1990].

where $\emptyset$ is proved and the resource requirements for such systems by looking at the values of the counter variables. All the experiments reported here were performed on an Intel Pentium 4 CPU 3.20GHz machine with 2GB of RAM under CentOS release 4.8.

4.8.1 Positional encoding using Mocha

In this section, we present experimental results using positional encoding based on the Mocha model checker. States of the system correspond to an assignment of values to state variables in the model-checker. The state variables representing an agent's memory are organised as a collection of 'cells', each holding at most one clause. For an agent i with memory bound $n_M(i)$, there are $n_M(i)$ cells. Each cell is represented by a pair of bitvectors, each of length $\delta = |P|$, representing the positive and negative literals in the clause in some standard order (e.g., lexicographic order). For example, if P contains the propositional variables A_1, A_2 and A_3 with index positions 0, 1 and 2 respectively, the clause $A_1 \vee \neg A_3$ would be represented by two bitvectors: "100" for the positive literals and "001" for the negative literals. This gives reasonably compact states. A positional Mocha encoding for a single agent tree-like resolution example with two propositional variables is given in Appendix D.

Actions by each agent such as reading a premise, resolution and communication with other agents are represented by Mocha *atoms* which describe the initial condition and transition relation for a group of related state variables. Reading a premise ($Read_{i,\alpha,\beta}$) simply sets the bitvectors representing an arbitrary cell in agent i's memory to the appropriate values for the clause α. Resolution ($Res_{i,\alpha_1,\alpha_2,l,\beta}$) is implemented using simple bit operations on cells containing values representing α_1 and α_2, with the results being assigned to an arbitrary cell in agent i's memory. Communication ($Copy_{i,\alpha,\beta}$) is implemented by copying the values representing α from a cell of agent j to an arbitrary cell of agent i. To express the communication bound, we use a counter for each agent which is incremented each time a copy action is performed by the agent.

After the counter for agent i reaches $n_C(i)$, the $Copy_{i,\alpha,\beta}$ action is disabled.

Mocha supports hierarchical modelling through composition of *modules*. A module is a collection of atoms and a specification of which of the state variables updated by those atoms are visible from outside the module. In our encoding, each agent is represented by a module. A particular distributed reasoning system is then simply a parallel composition of the appropriate agent modules. We verify the properties of the form $AG\ \neg tr(B_i\alpha)$, and use the counterexample trace generated by the model checker to show how the system reaches the state where α is proved. In the $AG\ \neg tr(B_i\alpha)$, $tr(B_i\alpha)$ is a suitable encoding of the fact that a clause α is present in the agent's memory, either as a disjunction of possible values of cell vectors or as a special boolean variable which becomes true when one of the cells contains a particular value. For example, if α is the empty clause, then both of the vectors of state variables representing one of agent i's cells should contain all 0s. (In practice, the situation is slightly more complex, as we need to check that a memory cell which contains all 0s at the current step was actually used in the proof, i.e., it contained a literal at the previous step.)

# Ag.	# Var.	Mem.	# Reachable states	# Reachable states (sym_search)	Max. MDDs	Max. MDDs (sym_search)	CPU time	CPU time (sym_search)
1	2	3	613	1009	554	629	0.1	0.1
2	2	2,2	5177	17921	977	2115	0.7	1
1	3	4	417201	835041	61986	92605	321	498
2	3	2,2	415933	1.65821e+06	42552	121223	252	709

Table 4.4: Mocha positional encoding

Mocha supports on-the-fly model checking which is performed by verifying, at each step of the reachability analysis, whether the formula is satisfied in states reached so far. As soon as the property is violated in a state the algorithm will terminate by producing a counter example resulting in the state where the property is violated. We can obtain the complete reachable state space information using Mocha's sym_search command. In Table 4.4 we provide the state apace size, BDD size, and CPU time information obtained from the counterexample trace and using sym_search command.

4.8.2 Non-positional encoding using Mocha

In this section, we present experimental results using non-positional encoding and the Mocha model checker. The encoding is based on a list of possible (useful) clauses and the appropriate transitions. The encoding follows the assumptions which have been made in § 4.6. Each clause of the set of all possible clauses F is represented as a Boolean variable. In order to implement memory bounds, message counters, clause adding and overwriting operations, we use counter variables of range type and Mocha's event variables. Actions by each agent such as reading a premise, resolving two clauses, and communication with other agents are represented by Mocha atoms which describe the initial condition and transition relation for a group of related state variables. A non-positional Mocha encoding for a single agent tree-like resolution example with two propositional variables is given in Appendix E. Table 4.5 summarises Mocha's runtime information based on non-positional encoding.

# Ag.	# Var.	Mem.	# Reachable states	# Reachable states (sym_search)	Max. MDDs	Max. MDDs (sym_search)	CPU time	CPU time (sym_search)
1	2	3	85	95	633	807	0.1	0.1
2	2	2,2	975	1313	2739	2828	0.8	2
1	3	4	15013	17643	49110	51954	44	48
2	3	2,2	99888	108716	377465	315364	1250	1527

Table 4.5: Mocha non-positional encoding

The experimental results show that positional encoding gives relatively better results (in terms of CPU time) than that of a non-positional encoding, some of the possible reasons for this are explained in § 4.8.3.

4.8.2.1 Experiments using NuSMV

In the previous section, we have presented some experimental results for the problem introduced in § 4.2 using Mocha which suggest that scalability may be an issue. In this section, we present some experimental results for the same problem using NuSMV 2.4.3. This is to know whether a better performance and scalability can be achieved using a different model checker. We consider a positional encoding that follows the assumptions which have been made in § 4.6. We use a positional encoding because it

gave better results in the Mocha experiments. The system is encoded in NuSMV using a similar approach to the Mocha encoding. The state variables representing an agent's memory are organised as a collection of 'cells', each holding at most one clause. Each cell is represented by a bitvector, each of length $\delta = 2 \times |P|$, representing the positive and negative literals in the clause in some standard order (e.g., lexicographic order). Actions by each agent such as reading a premise, resolution are represented using init and next which describe the initial condition and transition relation for a group of related state variables. These are implemented using simple bit operations. The specification language of NuSMV is designed to allow for the description of finite state systems and it supports both synchronous and asynchronous transitions. An SMV program consists of one or more module declarations including a *main* module. A module declaration is an encapsulated collection of declarations, constraints and specifications. A module can contain instances of other modules. In this encoding we implement the memory cells and specify read and resolve operations as an asynchronous network of non-deterministic processes. Among all the modules instantiated with the process keyword, one is non-deterministically chosen, and the assignment statements declared in that process are executed in parallel. As an example, a NuSMV encoding for a single agent tree-like resolution example with two propositional variables is given in Appendix F.

# Agents	# Var.	Mem.	# Reachable states	Max. BDDs (intermediate product)	CPU time
1	2	3	1305	1924	0.3
1	3	4	874097	1126430	413

Table 4.6: NuSMV positional encoding

NuSMV has an interactive mode through which the user can activate the various computation steps as system commands with different options. For example, the user can provide options (monolithic, partitioned etc.) on how to partition the model during reachable state space computation. The final BDD size for each step in the traversal is identical for both partitioned and monolithic methods. As far as the intermediate BDD sizes, the partitioned relation reports the sizes after each subset of quantifications are

done. On the other hand, with monolithic, the BDD size explosion happens inside BDD manager during a single operation, so it is not reported by NuSMV. The default option that NuSMV uses is the partitioned transition relation, and the information reported here is based on the default option. The results are summarised in Table 4.6. We found no significant differences in performance between Mocha and NuSMV.

4.8.3 Analysis of experimental results

The state space size and CPU time required by the model checkers for various problem sizes are summarised in Tables 4.4, 4.5 & 4.6. From the results we observe that the state space size reported by the model checkers in case of positional encodings for a single agent system are slightly larger than the theoretical analysis. This is due to use of some additional Boolean variables in the encodings. On the other hand, in the non-positional encoding for a single agent system the state space size reported by Mocha (the only model checker where a non-positional encoding was used) is approximately the same as the actual state space size of the problem. In this encoding we have used additional "event" variables which do not contribute to the state space size. In both the encodings, for multi-agent systems, the state space size reported by Mocha (the only model checker that was used) are slightly larger than the theoretical analysis. This is because for multi-agent systems some communication counter variables (an integer type) are also required to count the number of messages exchanged. Thus in a multi-agent system consisting of $n_{Ag}(> 1)$ agents, n_{Ag} additional counter variables are required one for each agent. We need $\log_2 r$ bits for a counter of range $r \in \mathbb{N}$. We further observe that, the maximal BDD (MDD) size during image computation may reach 1126430.

An important point that we can see from the Table 4.4 and Table 4.5 that the CPU time required by Mocha for non-positional encoding with two agents and three propositional variables is much larger than that of a positional encoding. This may be because in the non-positional encoding we had to use additional $2 \cdot 3^n$ event variables

and a counter variable of range type in order to implement the memory bound, clause adding and overwriting operations. Although the event variables do not contribute to the state space size, at each iteration in the image computation steps their presence is necessary and hence increases the size of the intermediate MDD sizes which causes the slowdown of the computation.

In NuSMV, image computation methods are implemented in different ways such as Monolithic, Threshold, and IWLS95 [Ranjan et al., 1995]. The Monolithic option is based on no partitioning at all. We have tried different methods, however, we did not find any significant differences in terms of BDD nodes or CPU time in order to perform the reachability analysis. The final BDD size for each step in the traversal is identical for both partitioned and monolithic methods.

In symbolic model checking, variable ordering plays a major role to achieve better performance. However, in our example model, for both the model checkers, variable ordering does not improve performance significantly.

In addition, we have also compared our resolution encodings with two other standard models from the NuSMV2.4.3 distribution package: "Shuttle Digital Autopilot engines out (3E/O) contingency guidance requirements (SDA)" and "the model of the MSI protocol with transient states (MSI)". The SDA model has also been studied in [Cimatti et al., 2000] as a reference example, and was used for a comparison of NuSMV performance with the original CMU SMV [McMillan, 1992]. These example models can be model checked quite efficiently. The state space size of these models are much greater than that of our resolution-based models, however, their maximal BDD sizes during image computation are much smaller (because of small branching factors) (see Table 4.7). The number of reachable states of a single-agent tree-like resolution based system with four propositional variables is 25624113, which is less than that of SAD and MSI. However, we were unable to verify properties of such a system using both the model checkers. In fact its intermediate BDD size reaches 35333396 at an early stage (at the sixth iteration) of image computation and the system blows up. This

is because the worst case branching factor of the system at this stage itself is greater than or equal to 75.

Model	Property	# Reachable states	# Max. BDDs (intermediate product)	CPU time
SDA	CTL AG(cg.finished -> output_k != 0)	2.10443e+14	2245	0.7
MSI	CTL AG ! (n0.c.modified & n1.c.modified & (n0.c.tag = n1.c.tag))	3.65528e+07	33693	9

Table 4.7: Reference model from the NuSMV2.4.3 distribution package

We observe that both the model checkers spent much of the verification time during reachable state space computation. Each iteration of the reachability analysis takes a large amount of time to complete execution. This is mostly due to the large branching factor of the model. Also, the system verification time increases exponentially with the number of propositional variables. This is because increasing the number of propositional variables increases the branching factor as well as the depth of the solution.

In the next chapter, we propose a framework for analysing resource requirements for systems of reasoning agents which reason using rules.

Chapter 5

Verifying rule-based systems

In this chapter, we consider the verification of system behaviour and resource requirements for distributed rule-based agents (i.e., agents which reason using rules). More specifically, we consider distributed problem solving in systems of communicating rule-based agents, and ask how much time (measured as the number of rule firings) and message exchanges does it take the system to find a solution. Using a synthetic but realistic example system of rule-based reasoning agents which allows the size of the problem and the distribution of knowledge among the reasoners to be varied, we show the Mocha model checker can be used to encode and verify properties of systems of distributed rule-based agents. We present preliminary results which highlight complex tradeoffs between time and communication bounds. We analyse the complexity of the problem and its model checking encoding in terms of state space size and branching factor. Finally, we argue (based on complexity analysis and experimental results) that reasonably sized problem instances are unlikely to be tractable for a standard model checker without steps to reduce the branching factor of the state space.

5.1 Rule-based systems

An important class of AI reasoning systems is rule-based. Rule-based systems are rapidly becoming an important component of mainstream computing technologies, for

example in business process modelling, the semantic web, sensor networks etc. However, while rules provide a flexible way of implementing such systems, the resulting system behaviour and the resources required to realise it can be difficult to predict. These problems become even more challenging in the case of *distributed rule-based systems*, where the system being designed or analysed consists of several communicating rule-based programs that exchange information via messages, e.g., a semantic web application or a sensor network. A communicated fact (or sensor reading) may be added asynchronously to the state of a rule-based system while the system is running, potentially triggering a new strand of computation which executes in parallel with current processing. To be able to provide response time guarantees for such systems, it is important to know how long each rule-based system's reasoning is going to take. In other situations, for example a rule-based system running on a PDA or other mobile device, the number of messages exchanged may be a critical factor. In this section, we present the basic structure of rule-based systems.

5.1.1 Structure of rule-based systems

A rule-based system consists of a *rule-base*; an *inference engine*; and a *working memory*. In some applications, a *user interface* may be present through which input and output signals are received and sent, however, it is not necessarily a part of the basic reasoning process. The architecture of a rule-based system is depicted in Figure 5.1 [Negnevitsky, 2005]. The rule-based system uses a very simple technique. Starting with a rule-base, which contains all of the appropriate knowledge encoded into IF-THEN rules for a given problem, and a working memory contains a set of facts which represent the initial state of the system.

It repeatedly executes an inference cycle consisting of three phases: are the *match* phase, the *select* phase and the *execute* phase. The *match* phase compares the conditions (IF) of all rules to working memory. A match for every condition in a rule constitutes an instantiation of that rule. A rule may have more than one instantiation.

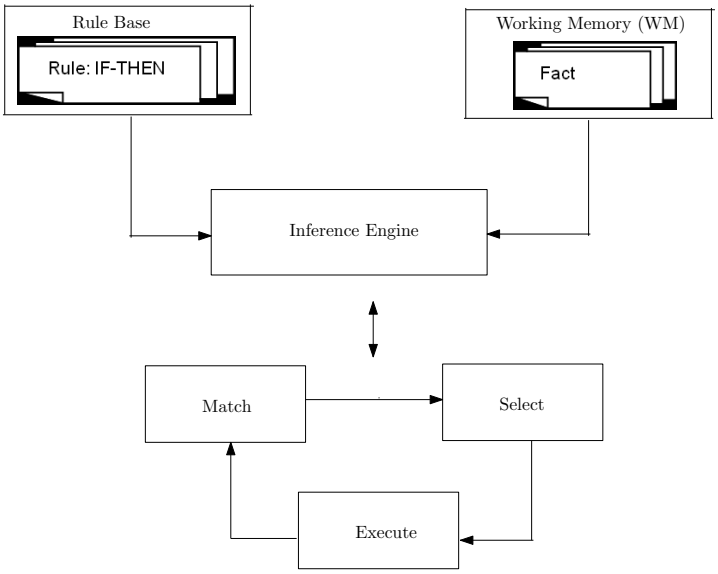


Figure 5.1: Structure of rule-based systems

The set of all rule instantiations collectively form a set, called the *conflict set*, which is passed through the *select* phase. In the *select* phase a *reasoning strategy* (or a conflict resolution strategy) determines a single instantiation, all instantiations or a subset of conflict set, which is passed to the execute phase. In the absence of an explicit reasoning strategy, all the instantiations are selected for execution. The *execute* phase then performs the actions of those instantiations passed specified in its THEN clause. These actions can modify the working memory, for example newly generated facts can be added to the working memory, some old facts can be deleted from the working memory or do anything else specified by the system designer. The cycle begins again with the *match* phase and the process continues until no more rules can be matched and all agents have an empty conflict set.

In this chapter, we do not consider explicit conflict resolution strategies, hence at each cycle in the *select* phase all the instantiations are selected for execution.

5.1.2 Basic form of rules

Rules in a rule-based system have the following general form:

$$rule : \langle \langle IF \rangle \rightarrow \langle THEN \rangle \rangle$$

where $\langle IF \rangle$ is a logical formula represented by a set of conditions which define when the rule can be applied, and $\langle THEN \rangle$ is the consequence of applying the rule; it can also be a logical formula, action or decision. The most common form of rule has more than one conditions ($|IF| \geq 1$) and a single conclusion ($|THEN| = 1$):

$$\langle P_1 \wedge P_2 \wedge \ldots \wedge P_{n-1} \rightarrow P_n \rangle \tag{5.1}$$

where $P_1, P_2, \ldots, P_{n-1}$ are atomic formulae[1] of some language, for example, propositional logic or first-order logic, and P_n can also be an atomic formula of the same language or an action or a decision. A rule of the form above, which has one or more conditions and a single conclusion, is known as Horn clause rule. For simplicity, all the variables occurring in a formula P_i (when P_i is an atomic formula of first-order logic) are assumed to be universally quantified. Furthermore, it is assumed that all the variables occurring in the right-hand side atomic formula P_n must also appear in some of the left-hand side atomic formulae $P_1, P_2, \ldots, P_{n-1}$.

In this chapter, we consider rule-based systems with propositional Horn clause rules. However, rule-based systems with first-order Horn clause rules are considered later in this thesis (cf. Chapter 6).

5.2 Systems of communicating rule-based reasoners

In this section, we describe the systems of communicating rule-based agents which we investigate. We assume that the system consists of n_{Ag} individual rule-based systems or agents, where $n_{Ag} \geq 1$. Each agent is identified by a value in $\{1, \ldots, n_{Ag}\}$, and we use variables i and j over $\{1, \ldots, n_{Ag}\}$ to refer to agents. Each agent i has a *program*, consisting of propositional Horn clause rules, and a working memory, which contains facts (propositions). The restriction to propositional rules is not critical: if the rules do not contain functional symbols and we can assume a fixed finite set of

[1] An atomic formula is a formula that contains no logical connectives nor quantifiers.

constant symbols, then any set of first-order Horn clauses and facts can be encoded as propositional formulae. If an agent i has a rule $A_1 \wedge A_2 \wedge \ldots \wedge A_n \rightarrow B$, the facts $A_1, \ldots, A_n$ are in i's working memory and B is not in i's working memory in state s, then i can fire the rule, adding B to i's working memory in the successor state s'.

In addition to firing rules, agents can exchange messages regarding facts currently in their working memory. The exchange of information between agents is modelled as an abstract *Copy* operation as before. An agent can also perform an Idle operation (do nothing). Furthermore, each agent performs a single action at each step.

Time	Agent 1	Agent 2
t_0	$\{A_1, A_2, A_3, A_4\}$	$\{A_5, A_6, A_7, A_8\}$
operation:	RuleB2	RuleB4
t_1	$\{A_1, A_2, A_3, A_4, B_2\}$	$\{A_5, A_6, A_7, A_8, B_4\}$
operation:	RuleB1	RuleB3
t_2	$\{A_1, A_2, A_3, A_4, B_1, B_2\}$	$\{A_5, A_6, A_7, A_8, B_3, B_4\}$
operation:	RuleC1	RuleC2
t_3	$\{A_1, A_2, A_3, A_4, B_1, B_2, C_1\}$	$\{A_5, A_6, A_7, A_8, B_3, B_4, C_2\}$
operation:	Idle	Copy (C_1 from Agent 1)
t_4	$\{A_1, A_2, A_3, A_4, B_1, B_2, C_1\}$	$\{A_5, A_6, A_7, A_8, B_3, B_4, C_1, C_2\}$
operation:	Idle	RuleD1
t_5	$\{A_1, A_2, A_3, A_4, B_1, B_2, C_1\}$	$\{A_5, A_6, A_7, A_8, B_3, B_4, C_1, C_2, D_1\}$

Table 5.1: Example: derivation with two agents

A problem is considered to be solved if one of the agents has derived the goal. The time taken to solve the problem is taken to be the total number of steps by the whole system (agents firing their rules or copying facts in parallel, at most one operation executed by each agent at every step). This abstracts away from the cost of rule matching etc. This assumption is made for simplicity and a single 'tick' can be replaced with a numerical value reflecting real time taken by the system to fire a rule (worst case or average). The amount of communication required to solve the problem is taken to be the total number of copy operations performed by all agents. Note that the only agent which incurs the communication cost is the agent which performs the copy. As with our model of time, the assumptions regarding communication are made for simplicity; it is straightforward to modify the definition of communication so that, e.g., the 'cost'

of communication is paid by both agents, communication takes more than one tick of time, and communication is non-deterministic.

The execution of a distributed rule-based system can be modelled as a state transition system where states correspond to combined states of agents (set of facts in each agent's working memory) and transitions correspond to agents performing actions in parallel, where each agent's action is either a single rule firing, a copy action, or an idle action.

As an example, consider a system of two agents, 1 and 2. The agents share the same set of rules is as follows.

$$\textbf{RuleB1}\ A_1 \wedge A_2 \rightarrow B_1 \qquad \textbf{RuleB2}\ A_3 \wedge A_4 \rightarrow B_2$$

$$\textbf{RuleB3}\ A_5 \wedge A_6 \rightarrow B_3 \qquad \textbf{RuleB4}\ A_7 \wedge A_8 \rightarrow B_4$$

$$\textbf{RuleC1}\ B_1 \wedge B_2 \rightarrow C_1 \qquad \textbf{RuleC2}\ B_3 \wedge B_4 \rightarrow C_2$$

$$\textbf{RuleD1}\ C_1 \wedge C_2 \rightarrow D_1$$

The goal is to derive D_1. Table 5.1 gives a simple example of a run of the system starting from a state where agent 1 has A_1, A_2, A_3 and A_4 in its working memory, and agent 2 has A_5, A_6, A_7, A_8. In this example, the agents require one Copy operation and five time steps to derive the goal. (In fact, this is an optimal use of resources for this problem, as verified using Mocha, see § 5.7). We will use variations on this synthetic 'binary tree' problem, in which the A_is are the leaves and the goal is the root of the tree, as examples, depicted in Figure 5.2. This problem is typical of a class of distributed reasoning problems and can be easily parameterised by the number of leaf facts and the distribution of facts and rules among the agents. For example, a larger system can be generated using 16 'leaf' facts $A_1, \ldots, A_{16}$, adding extra rules to derive B_5 from A_9 and A_{10}, etc., and a new goal E_1 derivable from D_1 and D_2. We will refer to this as a

'16 leaf example'.

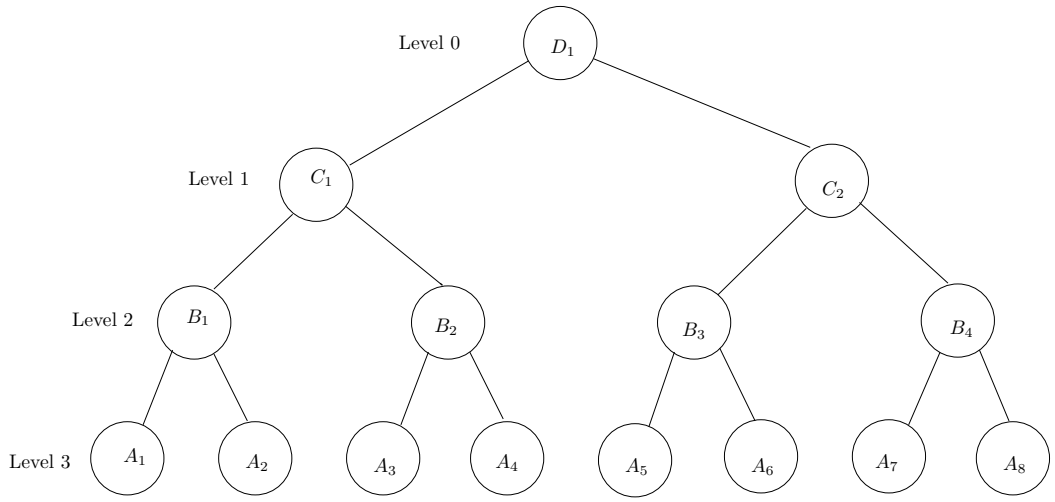


Figure 5.2: Binary tree example

5.3 Property specification

Let us consider the '8 leaf example' discussed above, and a run of the system to derive the goal formula D_1 which is shown in Table 5.1. The resource requirements for the system to derive the goal formula D_1 are one copy operation and 5 time steps. We can prove that $start \rightarrow EX^5\,[B_1D_1 \vee B_2D_1]$ (i.e., from the start state, the agents can derive the formula D_1 in 5 steps), where B_i is a belief operator (discussed in the next section) for each agent i. This is a very simple case; however, if we increase the problem size and distribute leaf nodes to the agents in various patterns, the verification task would be hard to do by hand. Therefore it is more convenient to use an automatic method to verify them. In order to verify these properties automatically we use symbolic model checking tools, which will be discussed shortly. To obtain the actual derivation we can also attempt to verify the negation of a formula, for example $AG\neg B_iD_1$ (for $i = 1, 2$)—the counterexample trace will show how the system reaches the state where the goal formula D_1 is derived. In the following, we briefly describe a temporal doxastic logic that can be used to reason about the system.

If formulae are not deleted once they are in the agent's memory, in order to verify

that an agent has derived φ within n timesteps, it is sufficient to check whether φ is in the agent's memory at the nth step, so we can use $EX^n B\varphi$ to verify whether $EX^{\leq n} B\varphi$ holds.

5.4 Logical formalism

To reason about systems of distributed rule-based reasoning agents we use $\mathcal{L}_{CRB}$ developed by Nga and Alechina, a temporal doxastic logic which allows us to describe a set of reasoning agents with bounds on time and on communication. In this section, we briefly describe the syntax and semantics of $\mathcal{L}_{CRB}$, a detailed description can be found in [Alechina et al., 2008b]. The language of $\mathcal{L}_{CRB}$ is an extension of CTL^* [Clarke et al., 2000, pp. 27–30], and contains a belief operator for each agent and communication modalities. All the properties of interest are expressible in CTL, but CTL^* is used to make the completeness proof easier.

We begin by defining an internal language for each agent. This language includes all possible formulae that the agent can store in its working memory. Let $\mathcal{A} = \{1, \ldots, n_{Ag}\}$ be the set of all agents, and $\mathcal{P}$ a finite common alphabet of facts. Let Π be a finite set of rules of the form $P_1, \ldots, P_n \rightarrow P$, where $n \geq 0$, $P_i, P \in \mathcal{P}$ for all $i \in \{1, \ldots, n\}$ and $P_i \neq P_j$ for all $i \neq j$. For convenience, we use the notation $pre(\rho)$ where $\rho \in \Pi$ for the set of premises of ρ and $con(\rho)$ for the conclusion of ρ. For example, if $\rho = P_1, \ldots, P_n \rightarrow P$, then $pre(\rho) = \{P_1, \ldots, P_n\}$ and $con(\rho) = P$. The internal language IL, then, includes all the facts $P \in \mathcal{P}$ and rules $\rho \in \Pi$. We denote the set of all formulae of IL by $\Omega = \mathcal{P} \cup \Pi$. Note that Ω is finite. The communication ability of the agents is expressed by the **Copy** action which copies a fact from another agent's memory. The limit on each agent's communication ability is $n_C(i)$: in any valid run of the system, agent i can perform at most $n_C(i)$ **Copy** actions.

5.4.1 Syntax of $\mathcal{L}_{CRB}$

The syntax of $\mathcal{L}_{CRB}$ includes the temporal operators of CTL* and is defined inductively as follows:

- $\top$ (tautology) and $start$ (a propositional variable which is only true at the initial moment of time) are well-formed formulae (wff) of $\mathcal{L}_{CRB}$,
- $cp_i^{=n}$ (which states that the value of agent i's communication counter is n) is a wff of $\mathcal{L}_{CRB}$ for all $0 \leq n \leq n_C(i)$ and $i \in \mathcal{A}$,
- $B_i P$ (agent i believes P) and $B_i \rho$ (agent i believes ρ) are wffs of $\mathcal{L}_{CRB}$ for any $P \in \mathcal{P}$, $\rho \in \Pi$ and $i \in \mathcal{A}$,
- If φ and ψ are wffs of $\mathcal{L}_{CRB}$, then so are $\neg\varphi$ and $\varphi \wedge \psi$,
- If φ and ψ are wffs of $\mathcal{L}_{CRB}$, then so are $X\varphi$ (in the next state φ), $\varphi U \psi$ (φ holds until ψ), $A\varphi$ (on all paths φ).

Other classical abbreviations for $\bot$, $\vee$, $\rightarrow$, $\leftrightarrow$, and temporal operations:

$F\varphi \equiv \top U \varphi$ (at some point in the future φ) and $G\varphi \equiv \neg F \neg \varphi$ (at all points in the future φ), and $E\varphi \equiv \neg A \neg \varphi$ (on some path φ) are defined as usual. For convenience, we also introduce the following abbreviations: $CP_i = \{cp_i^{=n} \mid 0 \leq n \leq n_C(i)\}$ and $CP = \bigcup_{i \in \mathcal{A}} CP_i$.

5.4.2 Semantics of $\mathcal{L}_{CRB}$

The semantics of $\mathcal{L}_{CRB}$ is defined by $\mathcal{L}_{CRB}$ transition systems which are based on ω-tree structures. Let (T, R) be a pair where T is a set and R is a binary relation on T. (T, R) is a ω-tree frame iff the following conditions are satisfied.

1. T is a non-empty set.
2. R is total, i.e. for all $t \in T$, there exists $s \in T$ such that tRs.

3. Let $<$ be the strict transitive closure of R, namely $\{(s,t) \in T \times T \mid \exists n \geq 0, t_0 = s, \ldots, t_n = t \in T$ such that $t_i R t_{i+1} \forall 0 \leq i \leq n-1\}$.

4. For all $t \in T$, the past $\{s \in T \mid s < t\}$ is linearly ordered by $<$.

5. There is a smallest element called the root, which is denoted by t_0.

6. Each maximal linearly $<$- ordered subset of T is order-isomorphic to the natural numbers.

A branch of (T, R) is an ω-sequence $(t_0, t_1, \ldots)$ such that t_0 is the root and $t_i R t_{i+1}$ for all $i \geq 0$. We denote $B(T, R)$ to be the set of all branches of (T, R). For a branch $\sigma \in B(T, R)$, σ_i denotes the element t_i of σ and $\sigma_{\leq i}$ is the prefix $(t_0, t_1, \ldots, t_i)$ of σ.

A $\mathcal{L}_{CRB}$ transition system M is defined as a triple (T, R, V) where:

- (T, R) is a ω-tree frame,
- $V : T \times \mathcal{A} \rightarrow \wp(\Omega \cup CP)$ such that for all $s \in T$ and $i \in \mathcal{A}$: $V(s,i) = Q \cup \{cp_i^{=n}\}$ for some $Q \in \wp(\Omega)$ and $cp_i^{=n} \in CP_i$. We denote $V^*(s,i) = V(s,i) \setminus CP_i$.

The truth of a $\mathcal{L}_{CRB}$ formula at a point n of a path $\sigma \in B(T, R)$ is defined inductively as follows:

- $M, \sigma, n \models \top$,
- $M, \sigma, n \models start$ iff $n = 0$,
- $M, \sigma, n \models B_i \alpha$ iff $\alpha \in V(s,i)$,
- $M, \sigma, n \models cp_i^{=m}$ iff $cp_i^{=m} \in V(s,i)$,
- $M, \sigma, n \models \neg\varphi$ iff $M, \sigma, n \not\models \varphi$,
- $M, \sigma, n \models \varphi \wedge \psi$ iff $M, \sigma, n \models \varphi$ and $M, \sigma, n \models \psi$,
- $M, \sigma, n \models X\varphi$ iff $M, \sigma, n+1 \models \varphi$,

- $M, \sigma, n \models \varphi U \psi$ iff $\exists m \geq n$ such that $\forall k \in [n, m)$ $M, \sigma, k \models \varphi$ and $M, \sigma, m \models \psi$,
- $M, \sigma, n \models A\varphi$ iff $\forall \sigma' \in B(T, R)$ such that $\sigma'_{\leq n} = \sigma_{\leq n}$, $M, \sigma', n \models \varphi$.

The models of $\mathcal{L}_{CRB}$ satisfy a set of constraints on the accessibility relation. Intuitively, each R is composed of an n_A-tuple of agents' actions performed in parallel. We will next define precisely the set of actions that each agent can perform. They are $Rule_{i,\rho}$, $Copy_{i,\alpha}$ and $Idle_i$ where $i \in \mathcal{A}$, $\rho \in \Pi$ and $\alpha \in \Omega$. $Rule_{i,\rho}$ is the action of an agent i firing ρ; $Copy_{i,\alpha}$ the action of copying α from another agent and $Idle_i$ is when agent i does nothing and moves to the next state.

We set constraints on the set of models such that the two following conditions are satisfied: (i) any transition between two states of the model corresponds to the effect of actions done by all agents in $\mathcal{A}$ and (ii) for any action of an agent in $\mathcal{A}$ that is applicable at a state s of the model, then there exists another state s' and a transition from s to s' which corresponds to the effect of the action. To formalise those two conditions, we have the following definitions.

Definition 5.4.1. *Let (T, R, V) be a tree model. The set of effective transitions R_a for an action a is defined as a subset of R and satisfies the following conditions, for all $(s, t) \in R$*

1. *$(s, t) \in R_{Rule_{i,\rho}}$ iff $\rho \in V(s, i)$, $V(s, i) \supseteq pre(\rho)$, $con(\rho) \notin V(s, i)$ and $V(t, i) = V(s, i) \cup \{con(\rho)\}$.* This condition says that s and t are connected by agent i's *rule-fired* transition if the following is true: ρ is a rule of i, $V(s, i)$ contains all premises of ρ but not its conclusion and the conclusion of ρ is added to the next state t of i.

2. *$(s, t) \in R_{Copy_{i,\alpha}}$ iff $\alpha \in V(s, j)$ for some $j \in \mathcal{A}$ and $j \neq i$, $cp_i^{=n} \in V(s, i)$ such that $n < n_C(i)$, $\alpha \notin V(s, i)$ and $V(t, i) = V(s, i) \setminus \{cp_i^{=n}\} \cup \{cp_i^{=n+1}\} \cup \{\alpha\}$.* In this condition, s and t are connected by a $Copy$ transition of agent i iff i has copied so far at most $n_C(i) - 1$ messages from other agents, at s, i does not have

α in its working memory while another agent j does and at the next state t, α is added into the working memory of i and its message counter is increased by one.

3. $(s,t) \in R_{Idle_i}$ *iff* $V(t,i) = V(s,i)$. The $Idle$ transition does not change the state.

Below, we specify when an action is applicable. Note that we only enable deriving a formula if this formula is not already in the agent's working memory.

Definition 5.4.2. *Let (T,R,V) be a tree model. The set $Act_{s,i}$ of applicable actions that an agent i can perform at a state $s \in T$ is defined as follows:*

1. $Rule_{i,\rho} \in Act_{s,i}$ *iff* $\rho \in V(s,i)$, $pre(\rho) \subseteq V(s,i)$ *and* $con(\rho) \notin V(s,i)$.
2. $Copy_{i,\alpha} \in Act_{s,i}$ *iff* $n < n_C(i)$ *where* n *is from* $cp_i^{=n} \in V(s,i)$, $\alpha \notin V(s,i)$, $\alpha \in V(s,j)$ *for some* $j \in \mathcal{A}$.
3. *It is always the case that* $Idle_i \in Act_{s,i}$.

Finally, the definition of the set of models corresponding to a system of rule-based reasoners is given below:

Definition 5.4.3. *$M(n_C)$ is the set of models (T,R,V) which satisfies the following conditions:*

1. $cp_i^{=0} \in V(t_0,i)$ *where* t_0 *is the root of* (T,R) *for all* $i \in \mathcal{A}$.
2. $R = \bigcup_{\forall a} R_a$.
3. *For all* $s \in T$, $a_i \in Act_{s,i}$, *there exists* $t \in T$ *such that* $(s,t) \in R_{a_i}$ *for all* $i \in \mathcal{A}$.

5.5 Analysis of the problem complexity

In this section, we present an analysis of the complexity of the binary-tree problem. We analyse the problem complexity in terms of its state space size and branching factor. Let us consider the binary-tree example depicted in Figure 5.2. It is easy to compute

the set of all possible configurations for a single-agent system, when all leaf facts are present in its initial working memory, and the agent has the following set of rules:

$$\textbf{RuleB1}\ A_1 \wedge A_2 \rightarrow B_1 \qquad \textbf{RuleB2}\ A_3 \wedge A_4 \rightarrow B_2$$

$$\textbf{RuleB3}\ A_5 \wedge A_6 \rightarrow B_3 \qquad \textbf{RuleB4}\ A_7 \wedge A_8 \rightarrow B_4$$

$$\textbf{RuleC1}\ B_1 \wedge B_2 \rightarrow C_1 \qquad \textbf{RuleC2}\ B_3 \wedge B_4 \rightarrow C_2$$

$$\textbf{RuleD1}\ C_1 \wedge C_2 \rightarrow D_1$$

Figure 5.3 shows the set of all possible configurations (numbered 1 to 26) and the corresponding state transition graph of the system. Let $S = \{1, 2, \ldots, 26\}$ be the set of all possible configurations of the system. Let $S_3 = \{1\}$, $S_2 = \{2, 3 \ldots, 16\}$, $S_1 = \{17, 18, \ldots, 25\}$, and $S_0 = \{26\}$. Then $S = \bigcup_{l=0}^{3} S_l$. The singleton set S_3 contains the initial configuration of the state space, in this initial configuration all leaf facts are true (present in the initial working memory). The set S_2 contains all those configurations which are A_i's (all the leaf facts) in concatenation with all possible combinations of B_i's. That is S_2 contains those configurations of the state space which represent all possible ways B_i may be present in the agent's working memory. Similarly, the set S_1 contains those configurations of the state space which represent all possible ways C_i may be present in the agent's working memory, and the set S_0 contains the configuration of the state space which represents D_1's presence in the agent's working memory. For ease of illustration, we assume that there is an S_l corresponding to each Level l ($0 \leq l \leq 3$) of the tree depicted in Figure 5.2. Let N_l denote the cardinality of the set S_l ($0 \leq l \leq 3$). Then the value of N_l can be calculated as follows:

$$\begin{aligned} N_3 &= {}^{8}C_8 \\ &= 1. \end{aligned}$$

The above expression gives the number of initial configuration of the state space: when all the 8 leaf facts are present in the working memory; i.e., 8 elements are chosen from a set of size 8.

$$N_2 = {}^4C_1 \cdot {}^8C_8 + {}^4C_2 \cdot {}^8C_8 + {}^4C_3 \cdot {}^8C_8 + {}^4C_4 \cdot {}^8C_8$$
$$= 4 + 6 + 4 + 1$$
$$= 15.$$

The above expression gives the number of all those configurations which are A_i's (all the leaf facts ${}^8C_8$) in concatenation with all possible combinations of B_i's. Note that we keep ${}^8C_8$ in the expression to come up with a pattern otherwise we can simply replace it by 1.

$$N_1 = ({}^2C_1 \cdot [\textstyle\sum_{i=0}^{2} {}^2C_i] \cdot {}^8C_8) + ({}^2C_2 \cdot {}^4C_4 \cdot {}^8C_8)$$
$$= (2 \cdot [1 + 2 + 1]) + 1$$
$$= 9.$$

The above expression gives the number of all those configurations which represent all possible ways C_i's may be present in the agent's working memory. The first term $({}^2C_1 \cdot [\sum_{i=0}^{2} {}^2C_i] \cdot {}^8C_8)$ on the right-hand side of the expression computes the presence of either C_1 or C_2 in different ways. The presence of C_i's in the working memory depends on the presence of B_i's. In order to ensure the presence of C_1 (or C_2) in the working memory, the B_i's may be present in $[\sum_{i=0}^{2} {}^2C_i]$ different ways, where the upper range 2 of the sum and the constant 2 of the combination operator are calculated based on the B_i's level in the tree. We keep ${}^8C_8$ instead of 1 for the same reason as stated above. The second term ${}^2C_2 \cdot {}^4C_4 \cdot {}^8C_8$ on the right-hand side of the expression computes the presence of both C_1 and C_2 in different ways. There is only one way they can be present together in the working memory: when all the B_i's and in turn all the A_i's are present.

$$N_0 = {}^1C_1 \cdot {}^2C_2 \cdot {}^4C_4 \cdot {}^8C_8$$
$$= 1.$$

The above expression gives the number of configuration(s) which represent all possible way the root node of the tree can be present in the working memory.

There is only one way this can happen when all other nodes of the tree are already present in the working memory.

Then the number of reachable states $\mathcal{N}$ is:

$$\mathcal{N} = N_0 + N_1 + N_2 + N_3 = 1 + 9 + 15 + 1 = 26$$

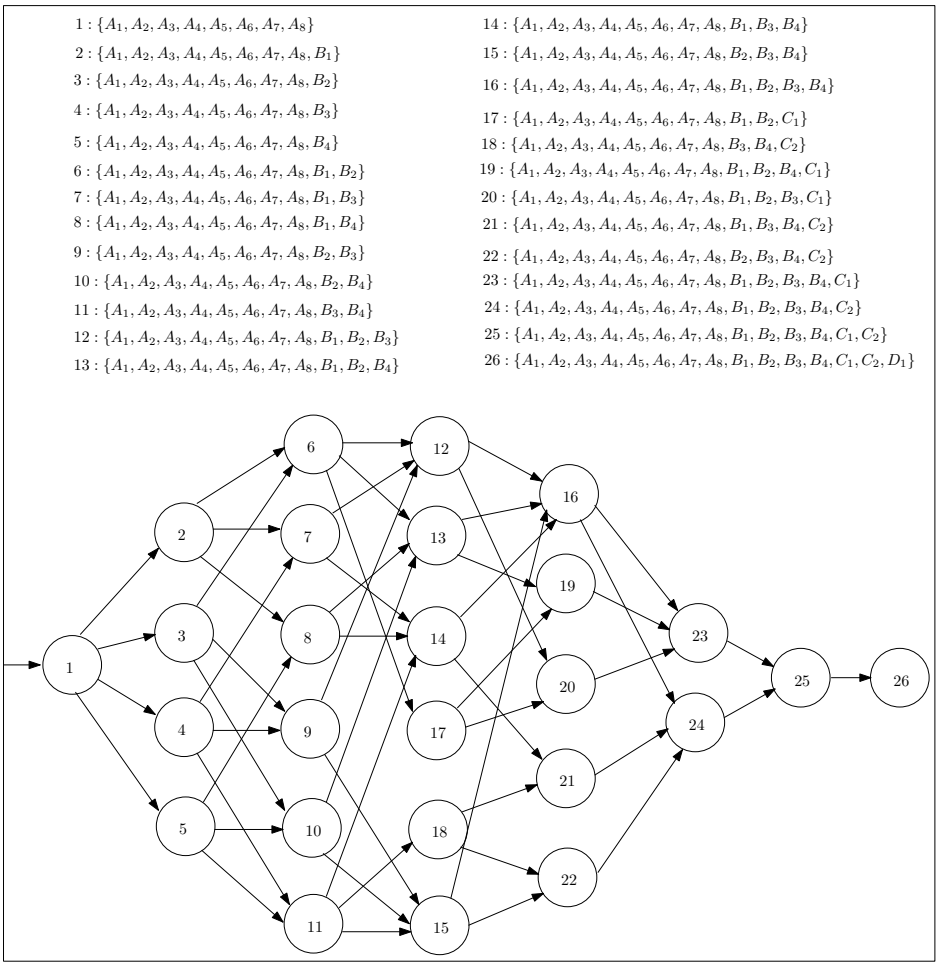


Figure 5.3: State transition graph for '8 leaf example'

Now we generalise the idea discussed above for an 'n leaf example' (see Figure 5.4). Without loss of generality we assume that the tree is a perfect binary tree.[2] Therefore, in a tree with height h has 2^h leaf nodes and the total number of nodes in the tree is $2^{h+1} - 1$. The number of nodes n_j at level j is determined by the expression $n_j = 2^j$, for $0 \leq j \leq h$. Therefore, a tree with n leaf nodes, at level h the number of

[2]A full binary tree is a tree in which every internal node has two children. A *perfect binary tree* is a full binary tree in which all leaves are at the same level [Preiss, 1999].

nodes is $n_h = 2^h = n$, at level $h-1$ the number of nodes is $n_{h-1} = 2^{h-1} = 2^h/2 = n/2$ and so on. We use the following notation to represent the nodes of the tree at each level.

Level : 0 $\quad X_1^0$

Level : 1 $\quad X_1^1, X_2^1$

$\vdots$

Level : $h-1$ $\quad X_1^{h-1}, X_2^{h-1}, \ldots, X_{n_{h-1}}^{h-1}$

Level : h $\quad X_1^h, X_2^h, \ldots, X_{n_h}^h$

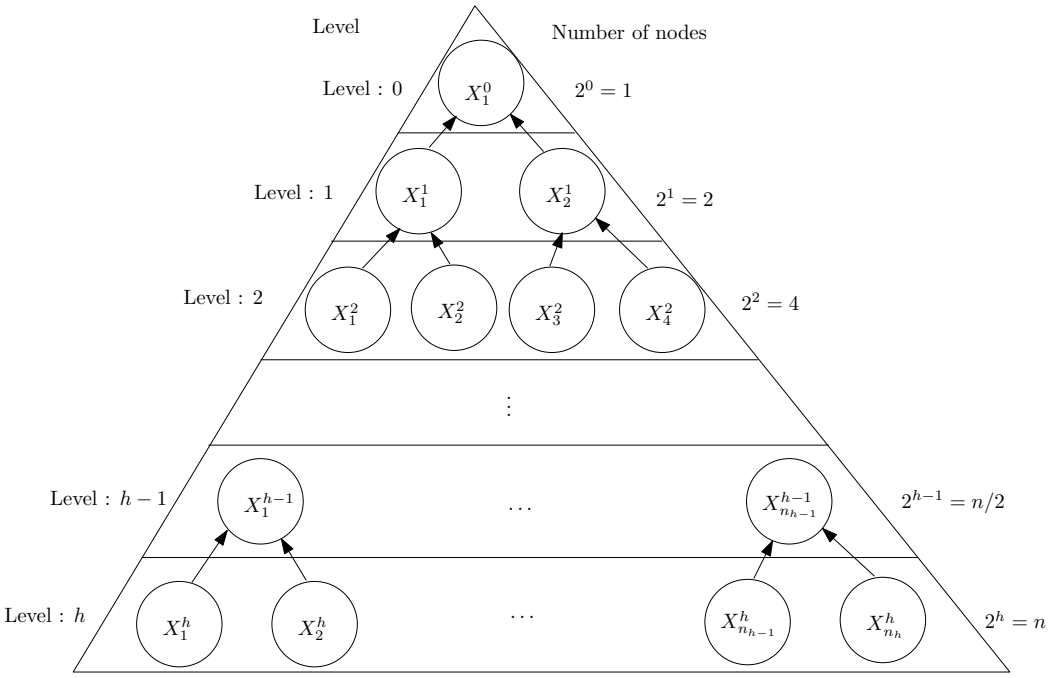


Figure 5.4: Levels and the corresponding nodes of the tree

We assume that all leaf facts $\{X_1^h, X_2^h, \ldots, X_{n_h}^h\}$ are true initially (present in the initial working memory). For $0 \leq l \leq h$, let S_l represent the set of configuration(s) corresponding to each Level l, and N_l denote the cardinality of the set S_l. Then the value of N_l (for $0 \leq l \leq h$) can be calculated using the following expressions:

$N_0 = \prod_{j=0}^{h} f(j)$, where $f(j) = {}^{n_j}C_{n_j}$

$N_1 = {}^{n_1}C_1 \cdot [\prod_{j=2}^{h-1} f(j)] \cdot {}^{n_h}C_{n_h} + \prod_{k=1}^{h} g(k)$, where $f(j) = \sum_{i=0}^{n_j - 2^{j-1}} {}^{n_j-2^{j-1}}C_i$ and $g(k) = {}^{n_k}C_{n_k}$

$N_2 = {}^{n_2}C_1 \cdot [\prod_{j=3}^{h-1} f_1(j)] \cdot {}^{n_h}C_{n_h} + {}^{n_2}C_2 \cdot [\prod_{j=3}^{h-1} f_2(j)] \cdot {}^{n_h}C_{n_h} + {}^{n_2}C_3 \cdot [\prod_{j=3}^{h-1} f_3(j)] \cdot {}^{n_h}C_{n_h} + \prod_{k=2}^{h} g(k)$

, where $f_1(j) = \sum_{i=0}^{n_j - 1\cdot 2^{j-2}} {}^{n_j-1\cdot 2^{j-2}}C_i$, $f_2(j) = \sum_{i=0}^{n_j - 2\cdot 2^{j-2}} {}^{n_j-2\cdot 2^{j-2}}C_i$, $f_3(j) = \sum_{i=0}^{n_j - 3\cdot 2^{j-2}} {}^{n_j-3\cdot 2^{j-2}}C_i$ and $g(k) = {}^{n_k}C_{n_k}$

$\vdots$

$N_{h-2} = {}^{n_{h-2}}C_1 \cdot [\sum_{i=0}^{n_{h-1}-2^1} {}^{n_{h-1}-2^1}C_i] \cdot {}^{n_h}C_{n_h} + {}^{n_{h-2}}C_2 \cdot [\sum_{i=0}^{n_{h-1}-2^2} {}^{n_{h-1}-2^2}C_i] \cdot {}^{n_h}C_{n_h} + \ldots +$
${}^{n_{h-2}}C_{n_{h-2}} \cdot {}^{n_{h-1}}C_{n_{h-1}} \cdot {}^{n_h}C_{n_h}$

$N_{h-1} = {}^{n_{h-1}}C_1 \cdot {}^{n_h}C_{n_h} + {}^{n_{h-1}}C_2 \cdot {}^{n_h}C_{n_h} + \ldots + {}^{n_{h-1}}C_{n_{h-1}} \cdot {}^{n_h}C_{n_h}$

$N_h = {}^{n_h}C_{n_h}$

Let $\mathcal{N}$ denote the number of reachable states of a single-agent system. The expression 5.2 defines the value of $\mathcal{N}$ as follows:

$$\mathcal{N} = \sum_{l=0}^{h} N_l \tag{5.2}$$

Let us consider the case of a multi-agent system, consisting of the parallel composition of n_{Ag} reasoning agents $\mathcal{A} = \{1, 2, \ldots, n_{Ag}\}$. If $\mathcal{N}_i$ denotes the number of reachable states of agent i, then the number of reachable states $\mathcal{N}'$ of the multi-agent is obtained by their parallel composition.

$$\mathcal{N}' = \prod_{i=1}^{n_{Ag}} \mathcal{N}_i \tag{5.3}$$

5.5.1 Asymptotic upper bound on the state space size

The following theorems provide the upper bounds on the set of reachable states of the systems using Big-O notation.

Theorem 5.5.1. *Let RuS be a single-agent rule-based system corresponding to an 'n leaf example'. Then the upper bound on the set of reachable states of RuS is of order*

$O(n \cdot 2^{\frac{n(n+2)}{4}})$.

Proof. see Appendix A for a proof. □

Theorem 5.5.2. *Let RuM be a multi-agent rule-based system consisting of n_{Ag} agents which share the same set of rules of an 'n leaf example'. Then the upper bound on the set of reachable states of RuM is of order $O(n^{n_{Ag}} \cdot 2^{n_{Ag} \cdot (\frac{n(n+2)}{4})})$.*

Proof. The proof is immediate from theorem 5.5.1 and the multiplication rule of Big-O complexity theory. □

5.5.2 The branching factor of the problem

The worst case branching factor of the search space of a rule-based system corresponding to an "n leaf example" is determined by the following theorems.

Theorem 5.5.3. *Let RuS be a single-agent rule-based system corresponding to an 'n leaf example'. Then the worst case branching factor of the search space of ReS is of order $O(\frac{n}{2})$.*

Proof. see Appendix B for a proof. □

Theorem 5.5.4. *Let RuM be a multi-agent rule-based system consisting of two agents which share the same set of rules of an 'n leaf example'. Then the worst case branching factor of the search space of RuM is of order $O(n^2)$.*

Proof. see Appendix C for a proof. □

5.6 Analysis of the encoding complexity

In symbolic model checking, in order to encode a $\mathcal{L}_{CRB}$ model each proposition of $\mathcal{P}$ can be represented by a Boolean variable. Thus $|\mathcal{P}|$ Boolean variables are required to encode the system. Therefore, the encoding complexity of an "n leaf example" will be same as its problem complexity (in terms of state space size and branching factor).

5.7 Model checking rule-based systems

In this experiment, we used the Mocha model checker in order to verify the properties of the system. The state of the system is described by a set of state variables and each system state corresponds to an assignment of values to the variables. The presence or absence of each fact in the working memory of an agent is represented by a boolean state variable $a_i A_j$ which represents the fact that agent i believes fact A_j. The initial values of these variables determines the initial distribution of facts between agents.[3] In the experiments reported below, all derived (non-leaf) variables were initialized to *false*, and only the allocation of leaves to each agent was varied.

5.7.1 Mocha encoding

The actions of firing a rule, copying a fact from another agent and idling were encoded as a Mocha *atom* which describes the initial condition and transition relation for a group of related state variables. Inference is implemented by marking the consequent of a rule as present in working memory at the next cycle if all of the antecedents of the rule are present in working memory at the current cycle. A rule is only enabled if its consequent is not already present in working memory at the current cycle. Communication is implemented by copying the value representing the presence of a fact in the working memory of another agent at the current cycle to the corresponding state variable in the agent performing the copy at the next cycle. Copying is only enabled if the fact to be copied is not already in the working memory of the agent performing the copy. To express the communication bound, we use a counter for each agent which is incremented each time a copy action is performed by the agent. In the experiments, we assumed that all rules are believed by all agents in the initial state, and did not implement copying of rules. However, this can be done in a straightforward way by adding an extra boolean variable to the premises of each rule, and implementing copying a

[3] We can also leave the initial allocation of facts undetermined, and allow the model checker to find an allocation which satisfies some property, e.g., that there is a proof which takes less than 7 steps. However for the experiments reported here, we specified the initial assignment of facts to agents.

rule as copying this variable. To allow an agent to idle at any cycle, the atoms which update working memory in each agent are declared to be *lazy*.

At each update round, Mocha non-deterministically chooses between the enabled rules and copy operations and idling.

5.7.2 Specifying system properties in Mocha

We can express a $\mathcal{L}_{CRB}$ formula such as $EX^n B_i\alpha$ (agent i may derive belief α in n steps) in the specification language of Mocha as $EX^n\ tr(B_i\alpha)$, where EX^n is EX repeated n times, and $tr(B_i\alpha)$ is a state variable encoding of the fact that α is present in the agent's working memory (e.g. $tr(B_i\alpha) = a_i A_j$ if $\alpha = A_j$). To obtain the actual derivation, we can verify an invariant which states that $tr(B_i\alpha)$ is never true, and use the counterexample trace to show how the system reaches the state where α is proved. To bound the number of messages used, we can include a bound on the value of the message counter of one or more agents in the property to be verified. For example, $EX^n\ (tr(B_i\alpha) \wedge tr(cp_i^{=0} \vee cp_i^{=1}))$, where $tr(cp_i^{=0} \vee cp_i^{=1})$ is translated to the statement $a_i\_counter < 2$, bounds the number of messages used by agent i to be at most 1.

5.7.3 Experimental results

In this section we give the results of experiments, originally presented in [Alechina et al., 2008b] for different sizes of the binary tree example and different distributions of leaves between the agents. The experiments were designed to investigate trade-offs between the number of steps and the number of messages exchanged (a shorter derivation with more messages or a longer derivation with fewer messages).

First, as a 'base case' and also to get an idea of the size of examples which can be model-checked in a reasonable time using our Mocha encoding, we ran experiments with just one agent, varying the size of the tree. The results are shown in Table 5.2. As one would expect, the number of steps equals to the total number of rules in the example. While for our binary tree example the results are unsurprising, in a less uniform

rule-based system such a result may be difficult to establish by a simple inspection of rules.

Case	# leaves	# steps
1.	8	7
2.	16	15
3.	32	31
4.	64	63
5.	128	127

Table 5.2: Resource requirements for one agent

Case	Agent 1	Agent 2	# steps	#Messages 1	#Messages 2
1.	$A_1 - A_8$		7	-	-
2.	$A_1 - A_7$	A_8	6	0	3
3.	$A_1 - A_7$	A_8	6	1	2
4.	$A_1 - A_7$	A_8	7	1	1
5.	$A_1 - A_7$	A_8	8	1	0
6.	$A_1 - A_6$	A_7, A_8	6	0	2
7.	$A_1 - A_6$	A_7, A_8	6	1	1
8.	$A_1 - A_6$	A_7, A_8	7	1	0
9.	$A_1 - A_4$	$A_5 - A_8$	5	1	0
10.	A_1, A_3, A_5, A_7	A_2, A_4, A_6, A_8	7	2	3
11.	A_1, A_3, A_5, A_7	A_2, A_4, A_6, A_8	11	0	4

Table 5.3: Resource requirements for optimal derivation in 8 leaves cases

We then investigated different distributions of leaf facts between the agents. Table 5.3 shows the number of derivation steps and the number of messages for each agent for varying distributions of 8 leaves. Note that there are several optimal (non-dominated) derivations for the same initial distribution of leaves between the agents. For example, when agent 1 has all the leaves apart from A_8, and agent 2 has A_8, the obvious solution is case 5, where agent 1 copies A_8 from agent 2, and then derives the goal in 7 steps, as in case 1. This derivation requires 8 time steps and one message. However, the agents can solve the problem in fewer steps by exchanging more messages. For example, case 2 describes the situation when agent 2 copies A_7 from agent 1, while agent 1 derives B_3 (step 1). Then agent 2 derives B_4 while agent 1 derives B_2 (step 2). Then agent 2 copies B_3 from agent 1, while agent 1 derives B_1 (step 3). At the next step agent 1 derives C_1 and agent 2 derives C_2 (step 4). Then agent 2 copies

C_1 from agent 1 (step 5) and agent 1 idles; finally at step 6 agent 2 derives D_1. The effect of the bound on messages varies with the distribution, as can be seen in cases 10 and 11: if agent 1 has all the odd leaves and agent 2 all the even leaves, then to derive the goal either requires 7 steps and 5 messages, or 11 steps and 4 messages.

Case	Agent 1	Agent 2	# steps	# msg 1	# msg 2
1.	$A_1 - A_{16}$		15	-	-
2.	$A_1 - A_{15}$	A_{16}	12	0	6
3.	$A_1 - A_{15}$	A_{16}	12	1	4
4.	$A_1 - A_{15}$	A_{16}	13	1	3
5.	$A_1 - A_{15}$	A_{16}	14	1	2
6.	$A_1 - A_{15}$	A_{16}	15	1	1
7.	$A_1 - A_{15}$	A_{16}	16	1	0
8.	$A_1 - A_{14}$	A_{15}, A_{16}	11	0	5
9.	$A_1 - A_{14}$	A_{15}, A_{16}	11	1	4
10.	$A_1 - A_{14}$	A_{15}, A_{16}	12	1	3
11.	$A_1 - A_{14}$	A_{15}, A_{16}	13	1	2
12.	$A_1 - A_{14}$	A_{15}, A_{16}	14	1	1
13.	$A_1 - A_{14}$	A_{15}, A_{16}	15	1	0
14.	$A_1 - A_{12}$	$A_{13}, A_{14}, A_{15}, A_{16}$	11	0	4
15.	$A_1 - A_{12}$	$A_{13}, A_{14}, A_{15}, A_{16}$	11	1	2
16.	$A_1 - A_{12}$	$A_{13}, A_{14}, A_{15}, A_{16}$	12	1	1
17.	$A_1 - A_{12}$	$A_{13}, A_{14}, A_{15}, A_{16}$	13	1	0
18.	$A_1 - A_3, A_5 - A_7, A_9 - A_{11}, A_{13} - A_{15}$	A_4, A_8, A_{12}, A_{16}	13	2	6
19.	$A_1 - A_3, A_5 - A_7, A_9 - A_{11}, A_{13} - A_{15}$	A_4, A_8, A_{12}, A_{16}	19	4	0
20.	$A_1, A_3, A_5, A_7, A_9, A_{11}, A_{13}, A_{15}$	$A_2, A_4, A_6, A_8, A_{12}, A_{14}, A_{16}$	13	4	5
21.	$A_1, A_3, A_5, A_7, A_9, A_{11}, A_{13}, A_{15}$	$A_2, A_4, A_6, A_8, A_{12}, A_{14}, A_{16}$	23	0	8

Table 5.4: Resource requirements for optimal derivation in 16 leaves cases

Similar trade-offs are apparent for a problem with 16 leaves, as shown in Table 5.4. However in this case there are a larger number of possible distributions of leaves, and, in general, more trade-offs for each distribution. For example, when one of the agents has all the leaves but one, we again have the obvious solution where agent 1 copies the missing leaf and derives the goal on its own, which takes 16 steps and 1 message (case 7). In addition there are 15, 14, 13 and 12 step derivations, where the shorter the derivation the more messages the agents have to exchange (cases $2 - 7$). We also see interesting trade-offs when agent 2 has two leaves (cases $8 - 13$) or four leaves in the same subtree (cases $14 - 17$). When agent 1 has 3 leaves in each subtree and agent 4 the fourth leaf in each subtree, there is again an obvious derivation in which agent 1 copies the 4 missing leaves and completes the derivation in 19 steps and 4 copy operations, and a more interesting one which takes 13 steps and the agents exchange more messages (agent 2 copies 3 leaves to complete a part of the proof, and then copies variables from higher up in the tree). The difference is also more marked in the 'odd

and even' case (cases 20 and 21), where agent 1 has all the odd leaves and agent 2 all the even leaves, where increasing the message bound by 1 reduces the length of the proof by 10 steps.

5.8 Analysis of experimental results

All the experiments reported in the previous section were performed on an Intel Pentium 4 CPU 3.20GHz machine with 2GB of RAM under CentOS release 4.8. In Table 5.5, we present some runtime system information produced by Mocha when verifying properties of the binary tree example and different distributions of leaves between the agents. That includes the state space size, maximal MDD size of a particular iteration during image computation, and the CPU time (in seconds). When a system property is violated Mocha produces a counter example trace that includes final MDD size of each step of the reachable state space computation. In this experiment, for a single agent, in order to verify the invariant property of the form $AG\neg\varphi$ (for example φ is the root node), model checker has to explore the entire reachable state space. This is because for a single agent system, to derive the root node, the system has to fire all the rules of the system. However, for a multi-agent system in order to verify invariant properties, model checker does not need to explore the entire reachable state space. This is because each agent does not necessarily have to fire all its rules. For instance, one agent can receive facts from other agent in the system. In Table 5.5, we have provided the complete reachable state space information for multi-agent cases using the Mocha's sym_search command, these are mentioned within second brackets. The distribution $(n/2, n/2)$ of leaf facts between agents indicates that the first $n/2$ leaf facts $A_1, A_2, \ldots, A_{n/2}$ are assigned to one agent and the other $n/2$ leaf facts $A_{n/2+1}, A_{n/2+2}, \ldots, A_n$ are assigned to the other agent in the system. Similarly, the distribution $(odd, even)$ of leaf facts indicates that the odd position node facts are assigned to one agent and the even position node facts are assigned to the other agent in the system.

# Ag.	# Leaves	Dist.	# Reach. states	# Reach. states (sym_search)	# Max. MDDs	# Max. MDDs (sym_search)	CPU time	CPU time (sym_search)
1	8	-	26	-	22	-	0.4	-
2	8	(4,4)	41943	336156	3108	3874	4	5
2	8	(odd,even)	55278	145511	3447	4636	7	8
1	16	-	784	-	173	-	1	-
2	16	(8,8)	8.6667e+08	2.34705e+10	131179	321423	469	3429
2	16	(odd,even)	7.52994e+08	3.64244e+09	189419	286196	613	2267
1	32	-	458330	-	1141	-	3	-
1	64	-	2.10066e+11	-	4655	-	251	-
1	128	-	4.41279e+22	-	38897	-	6472	-

Table 5.5: State space and CPU time produced by Mocha

The results show that the state space size produced by Mocha is a close approximation of the problem state space size. We observe that Mocha spent much of the verification time during reachable state space computation. Table 5.5 shows that the maximal MDD size of an intermediate product in a particular iteration during image computation is quite large, e.g., the size reaches up to $8.6667e + 08$. As in the resolution example, model checking performance heavily depends not only on the number of states and Boolean variables used in a model but also on the branching factor of the model. A large branching factor causes the slowdown of the overall model checking process. It also depends upon the solution depth. For example, a single agent 'n leaf' rule based system requires $(n - 1)$ iterations during reachable state space computation in order to reach the fixed point. The results suggest that the scalability issue of the models coming from the large branching factor and the solution depth of the problem.

To address the problem of scalability, in the next chapter we propose a framework for verifying systems of rule-based agents which uses explicit strategies and abstraction.

Chapter 6

A scalable verification framework for MAS

In the preceding chapters, we have described frameworks for the explicit modelling of computational (time, memory) and communication resources for distributed reasoning systems. We have seen that reasoning occurs in time and the agents can achieve a goal only if they are prepared to commit certain time, memory and communication resources. We have seen how properties of systems of distributed reasoning agents, such as existence of derivations with given bounds on memory, communication, and the number of inference steps, can be verified automatically. However, while these techniques work for small numbers of agents, we saw in chapters 4 and 5, they are unlikely to scale to large-scale systems. To address the problem of scalability, in this chapter we propose a framework for verifying systems of rule-based agents which uses explicit strategies and abstraction. The framework allows the use of abstract specifications consisting of LTL formulae to specify some of the agents in the system.

6.1 Verification framework

We would like to be able to verify properties of systems consisting of arbitrary numbers of complex communicating reasoners. However our experience has indicated that verifying such large, complex reasoning systems is infeasible with current model checking technologies.

The most straightforward approach to defining the global state of a multi-agent sys-

tem is as a (parallel) composition of the local states of the agents (cf. Chapters 4 & 5). At each step in the evolution of the system, each agent chooses from a set of possible actions (we assume that an agent can always perform an 'idle' action which does not change its state). The actions selected by the agents are then performed in parallel and the system advances to the next state. In a multi-agent system composed of n_{Ag} (≥ 1) agents, if each agent i can choose between performing at most a (≥ 1) actions, then the system as a whole can move in $a^{n_{Ag}}$ different ways from a given state at a given point in time. Along with state space size, model checking performance is heavily dependent on the branching factor of states in the reachable state space and the solution depth of a given problem. In general, the model checking algorithm for reachability analysis performs a breadth-first exploration of the state transition graph. When checking invariant (safety) properties, the model-checker will either determine that no states violate the invariant by exploring the entire state space, or will find a state violating the invariant and produce a counter-example.[1] However, even with state-of-the-art BDD-based model-checkers, memory exhaustion can occur when computing the reachable state space due to the large size of the intermediate BDDs (because of the high branching factor).

To overcome this problem, we propose two approaches: the use of abstract specifications to model the behaviour of some of the agents in the system, and exploiting information about the reasoning strategy adopted by the agents.

In model checking, when verifying large system, the model that describes the system must be designed as a compromise between the model precision and its state space size. Our approach to the verification of systems of communicating reasoners starts from the assumption that the detailed behaviour of only a small number of agents (perhaps only a single agent) is of interest to the system designer, and the remaining agents in the system can be considered at a high level of abstraction. This is largely true in practice, when different agents situated in different locations work indepen-

[1] Even with on-the-fly model-checking [Holzmann, 1996], the model checker has to explore the state space at least until the solution depth.

dently except for the exchange of messages. Their complete computational behaviour is therefore hidden from each other, and in general where the system designer may have little or no direct control over (or even knowledge of) the internal behaviour of some of the agents in the system. The external behaviours of such agents can be adequately captured by specifications in an appropriate temporal doxastic logic, i.e., their external behaviour can be represented by sets of temporal doxastic formulae, e.g., formulae of the form $X^{\leq n}\varphi$ describe agents which produce a certain message or input to the system within n time steps. Here φ can be, e.g., B_i $\mathsf{Ask}(i, j, P)$, B_i $\mathsf{Tell}(i, j, P)$, or B_i P.

In our framework, we assume that an agent in the system is either completely concrete or completely abstract. The representation of agents in the system are divided into two classes based on their behavioural specification, depicted in Figure 6.1. The system designer may have complete control over the internal behaviour of some agents in the system. The concrete agents class contains those agents. The remaining agents belong to the abstract agents class. In this step the designer identifies which agents (s)he needs to design for what classes. The designer also determines the number of agents he needs to place in each class and their possible interactions. An agent can interact with one or more agents in the system, but not necessarily every agent interacts with every other agent in the system. For simplicity, we assume that communication is error-free and takes one tick of time. The designer can consider the following different possible levels of system information in order to design and verify system properties.

1. The system designer may have detailed design information about the internal behaviour of some agents in the system including the initial facts in their working memories, their rules and the reasoning strategy. The remaining agents in the system are modelled using temporal doxastic formulae.

2. The system designer may have information of all the agents in the system including the initial facts in their working memories, their rules but no information at all about their reasoning strategy. This design gives the worst case model which

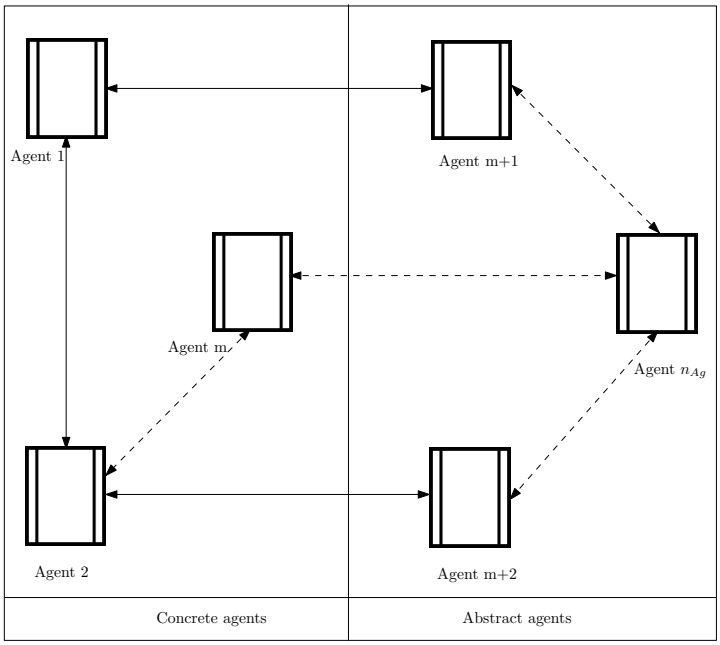


Figure 6.1: System behavioural specification

we have already taken into account in our previous work.

3. The system designer may have detailed information of all the agents in the system including the initial facts in their working memories, their rules and the reasoning strategy.

In the following sections, we describe in more detail how we model the concrete and abstract agents.

6.2 Communicating reasoners

We extend the model of distributed reasoners presented in Chapter 5. A distributed reasoning system consists of n_{Ag} (≥ 1) individual reasoners or *agents*. Each agent is identified by a value in $\{1, 2, \ldots, n_{Ag}\}$ and we use variables i and j over $\{1, 2, \ldots, n_{Ag}\}$ to refer to agents. An agent in the system is either concrete or abstract. Each concrete agent has a program, consisting of first-order Horn clause rules with negation-as-failure allowed in the premises[2], and a working memory, which contains facts (ground atomic

[2]Rules are of the form $P_1 \wedge \ldots \wedge P_n \rightarrow P$ where P is an atomic formula and P_i are atomic formulae or atomic formulae preceded by the negation as failure operator.

formulae) representing the initial state of the system. The introduction of first-order Horn clause rules and negation as failure increase the expressiveness of the framework, and makes it easier to model complex real world problems. The behaviour of each abstract agent is represented in terms of a set of temporal doxastic formulae. That is abstract specifications are given as LTL formulae which describe the external behaviour of agents, and allow their temporal behaviour (the response time behaviour of the agent), to be compactly modelled. The agents (concrete and abstract) execute synchronously. We assume that each agent executes in a separate process and that agents communicate via message passing. We further assume that each agent can communicate with multiple agents in the system at the same time.

6.3 Concrete agents

The behavioural specification of a concrete agent that the system designer can specify is depicted in Figure 6.2. The two main components of rule-based agents are the knowledge base (KB) which contains a set of condition-action rules and the working memory (WM) which contains a set of facts that constitute the current (local) state of the system. Another major component of a rule-based system is the inference engine which reasons over rules when the application is executed. The inference engine may have some reasoning strategies to handle cases when multiple rule instances are eligible to fire.

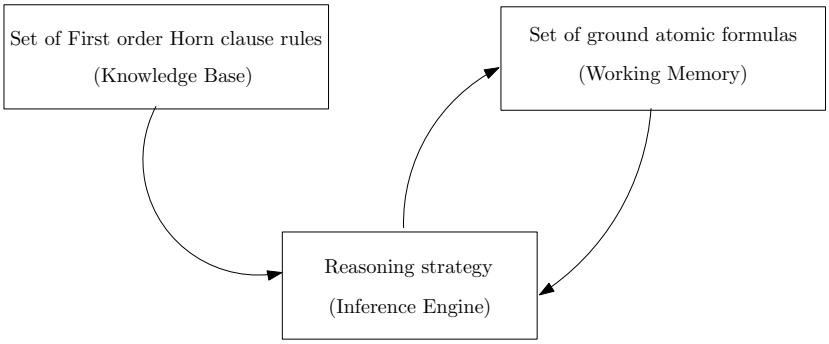


Figure 6.2: Individual concrete agent

6.3.1 Rules and facts

The syntax of rules that the system designer can specify has the following BNF:

$$
\begin{aligned}
&\mathit{Rule} \ ::= \ \langle \ \mathit{Priority} \ : \ \mathit{Patterns} \rightarrow \mathit{Pattern} \ \rangle \\
&\mathit{Patterns} \ ::= \ \mathit{Pattern}(\wedge \mathit{Pattern})^* \\
&\mathit{Pattern} \ ::= \ \mathit{Predicate}(\mathit{Terms}) \\
&\qquad\qquad \mid \mathit{Naf}(\mathit{Predicate}(\mathit{Terms})) \\
&\qquad\qquad \mid \mathit{Ask}(i, j, \mathit{Predicate}(\mathit{Terms})) \\
&\qquad\qquad \mid \mathit{Tell}(i, j, \mathit{Predicate}(\mathit{Terms})) \\
&\mathit{Priority} \ ::= \ N_{\geq 0} \\
&N_{\geq 0} \ ::= \ 0 \mid 1 \mid 2 \mid \ldots \\
&i \ ::= \ 1 \mid 2 \mid \ldots \mid n_{Ag} \\
&j \ ::= \ 1 \mid 2 \mid \ldots \mid n_{Ag} \\
&\mathit{Predicate} \ ::= \ \mathit{Identifier} \\
&\mathit{Terms} \ ::= \ \mathit{Term}(, \mathit{Term})^* \\
&\mathit{Term} \ ::= \ \mathit{Constant} \mid \mathit{Variable} \mid \mathit{Function} \\
&\mathit{Function} \ ::= \ \mathit{Identifier}(\mathit{Terms}) \\
&\mathit{Variable} \ ::= \ \mathit{Identifier} \\
&\mathit{Constant} \ ::= \ \mathit{Identifier} \\
&\mathit{Identifier} \ ::= \ \mathit{Letter}(\mathit{Letter} \mid \mathit{Digit})^* \\
&\mathit{Letter} \ ::= \ A \mid B \mid \ldots \mid Z \mid a \mid b \mid \ldots \mid z \\
&\mathit{Digit} \ ::= \ 0 \mid 1 \mid \ldots \mid 9
\end{aligned}
$$

That is, rules of a concrete agent have the following form:

$$\langle \ n \ : \ P_1 \wedge P_2 \wedge \ldots \wedge P_n \rightarrow P \ \rangle$$

where n is a constant that represents the priority of the rule. For communication, we assume a simple query-response scheme based on asynchronous message passing. Each agent's rules may contain two distinguished communication primitives: $\mathsf{Ask}(i, j, P)$, and $\mathsf{Tell}(i, j, P)$, where i and j are agents and P is an atomic formula not containing an Ask or a Tell. $\mathsf{Ask}(i, j, P)$ means 'i asks j whether P is the case' and $\mathsf{Tell}(i, j, P)$ means 'i tells j that P' ($i \neq j$). The positions in which the Ask and Tell primitives may appear in a rule depends on which agent's program the rule belongs to. Agent i may have an Ask or a Tell with arguments (i, j, P) in the consequent of a rule, e.g.,:

$$\langle \ n \ : \ P_1 \wedge P_2 \wedge \ldots \wedge P_n \rightarrow \ \mathsf{Ask}(i, j, P) \ \rangle$$

Whereas agent j may have an Ask or a Tell with arguments (i, j, P) in the antecedent of the rule. For example:

$$\langle\ n\ :\ \mathsf{Tell}(i, j, P) \rightarrow P\ \rangle$$

is a well-formed rule for agent j that causes it to believe i when i informs it that P is the case. No other occurrences of Ask or Tell are allowed. When a rule has either an Ask or a Tell as its consequent, we call it a communication rule. All other rules are known as deduction rules. These include rules with Asks and Tells in the antecedent as well as rules containing neither an Ask nor a Tell.

Firing a communication rule instance with the consequent $\mathsf{Ask}(i, j, P)$ adds the pattern $\mathsf{Ask}(i, j, P)$ both to the working memory of i and of j. Intuitively, i has a record that it asked j whether P is the case, and j has a record of being asked by i whether P is the case. Similarly, if the consequent of a communication rule instance is of the form $\mathsf{Tell}(i, j, P)$, then the corresponding pattern $\mathsf{Tell}(i, j, P)$ is added to the working memories of both the agents i and j. The set of facts are ground atomic formulae.

6.3.2 Reasoning strategy

At each cycle, each agent matches (unifies) the conditions of its rules against the contents of its working memory. The conditions of a rule are evaluated using the closed world assumption (i.e., $Naf(P)$ evaluates to true if P is not in working memory). A match for every condition of a rule constitutes an instance of that rule (a rule may have more than one instance). The set of all rule instances for an agent form the agent's *conflict set*. Each agent then chooses a subset of rule instances from the conflict set to be applied. Applying a rule adds the consequent of the rule as a new fact to the agent's working memory or sends a message to another agent. The cycle begins again with the match phase and the process continues until no more rules can be matched and all agents have an empty conflict set.

We assume that each concrete agent has a *reasoning strategy* (or conflict resolution strategy) which determines the order in which rules are applied when more than one rule matches the contents of the agent's working memory. The choice of reasoning strategy is important in determining the capabilities of the agent. For example, different reasoning strategies may determine how quickly/efficiently an answer to a query can be derived, or even whether an answer can be produced at all. The reasoning strategy is also important in determining trade-offs between the resources required to process a query. For example, if multiple queries arrive at about the same time, processing them sequentially may reduce the memory required at the cost of increasing the worst case response time for queries. Conversely, processing the queries in parallel may reduce the worst case response time at the cost of increasing the peak memory usage.

To allow the implementation of reasoning strategies, each pattern is associated with a time stamp which records the cycle at which the pattern was added to working memory. Rule priorities and fact time stamps can be used to determine which rule instance(s) are selected from the conflict set for execution. For example, a rule instance with the highest priority may be selected, or a rule instance may be selected whose antecedent patterns are associated with highest time stamp etc. The framework (and the TVRBA tool presented in Chapter 7) supports a set of standard conflict resolution strategies often used in rule-based systems including: rule ordering, depth, breadth, simplicity, and complexity. The internal configurations of the rules follow the syntax given below:

$$\langle\, n \;:\; [\, t_1 \;:\; P_1] \wedge [\, t_2 \;:\; P_2] \wedge \ldots \wedge [\, t_n \;:\; P_n] \rightarrow [\, t \;:\; P] \,\rangle$$

where the placeholders t_i's and t represent time stamps of patterns. When a rule instance of the above rule is fired, its consequent pattern P will be added to the working memory with time stamp $t = t' + 1$, i.e., t will be replaced by $t' + 1$, where t' is the current cycle time of the system.

Let $R_I = \{r \mid r \text{ is a rule instance}\}$ and $F_P = \{p \mid p \text{ is a TPattern}\}$ denote the set of rule instances and set of time patterns (every pattern has an associated timestamp

assigned to it), respectively. We define two partial orders $\leq_r$ and $\leq_f$ over R_I and F_P, respectively, as follows:

1. for any two rule instances $r, r' \in R_I$ where

$$r \equiv \langle\, n_1 \,:\, [\, t_1 \,:\, P_1] \wedge [\, t_2 \,:\, P_2] \wedge \ldots \wedge [\, t_n \,:\, P_n] \rightarrow [\, t \,:\, P] \,\rangle$$

$$r' \equiv \langle\, n_2 \,:\, [\, t'_1 \,:\, P'_1] \wedge [\, t'_2 \,:\, P'_2] \wedge \ldots \wedge [\, t'_m \,:\, P'_m] \rightarrow [\, t' \,:\, P'] \,\rangle$$

   we say that $r \leq_r r'$ (rule instance r' has priority over the rule instance r) iff $n_1 \leq n_2$, where $\leq$ is the standard less-than-or-equal relation on the set of non-negative integers $N_{\geq 0}$.

2. for any two time patterns $p, p' \in F_P$ where $p \equiv [t_1 \,:\, P_1]$ and $p' \equiv [t_2 \,:\, P_2]$, we say that $p \leq_f p'$ (fact P_2 has greater timestamp than the fact P_1) iff $t_1 \leq t_2$.

This information is used by each strategy. The system designer can specify the following standard conflict resolution strategies (based on those provided in [Culbert, 2007, Friedman-Hill, 2008, Tzafestas et al., 1989]).

1. **Rule ordering strategy** Select one rule instance from the conflict set that has the highest priority. If there are mutiple rule instances with the same priority, the rule instance to be executed is selected non-deterministically.

2. **Depth strategy** If the conflict set contains multiple rule instances with the highest priority, a rule instance whose antecedent patterns are associated with the highest timestamp is executed. If there are multiple rule instances whose antecedent patterns are associated with the highest timestamp, the rule instance to be executed is selected non-deterministically.

3. **Breadth strategy** If the conflict set contains multiple rule instances with the highest priority, a rule instance whose antecedent patterns are associated with the lowest timestamp is executed. If there are multiple rule instances whose antecedent patterns are associated with the lowest timestamp, the rule instance to be executed is selected non-deterministically.

4. **Specificity strategy (simplicity)** If the conflict set contains multiple rule instances with the highest priority, a rule instance with the smallest number of conditions is executed. If there are multiple rule instances with the smallest number of conditions, the rule instance to be executed is selected non-deterministically.

5. **Specificity strategy (complexity)** If the conflict set contains multiple rule instances with the highest priority, a rule instance with the largest number of conditions is executed. If there are multiple rule instances with the largest number of conditions, the rule instance to be executed is selected non-deterministically.

Different agents in the system may use different types of reasoning strategy.

6.4 Abstract agents

When verifying response time guarantees of the 'focal' agent(s), the concrete representation of 'peripheral' agents can be replaced by an abstract specification of their external (communication) behaviour, so long as the abstract specification results in behaviour that is indistinguishable from the original concrete representation for the purposes of verification, i.e., it produces queries and responds to queries within specified bounds. All other details of an abstract agent's internal behaviour are omitted.

The decision regarding which agents to abstract and how their external behaviour should be specified rests with the system designer. Specifications of the external (observable) behaviour of abstract agents may be derived from, e.g., assumed characteristics of as-yet-unimplemented parts of the system, assumptions regarding the behaviour of parts of the overall system the designer does not control (e.g., quality of service guarantees offered by an existing web service) or from the prior verification of the behaviour of other (concrete) agents in the system.

An abstract agent consists of a working memory and a behavioural specification. The behaviour of abstract agents is specified using the temporal logic LTL extended

with belief operators. The general form of the formulae used to represent the external behaviour of an abstract agent i is given below:

$$
\begin{aligned}
\rho &::= X^{\leq n}\varphi_1 \mid G(\varphi_2 \rightarrow X^{\leq n}\varphi_3) \\
\varphi_1 &::= B_i\ \mathsf{Ask}(i,j,P) \\
&\quad \mid B_i\ \mathsf{Tell}(i,j,P) \\
&\quad \mid B_i\ \mathsf{Ask}(j,i,P) \\
&\quad \mid B_i\ \mathsf{Tell}(j,i,P) \\
&\quad \mid B_i\ P \\
\varphi_2 &::= B_i\ \mathsf{Ask}(j,i,P) \\
\varphi_3 &::= B_i\ \mathsf{Tell}(i,j,P)
\end{aligned}
$$

where X is the next step temporal operator, $X^{\leq n}$ is a sequence of n X operators, G is the temporal 'in all future states' operator, and B_i for each agent i is a syntactic doxastic operator used to specify agent i's 'beliefs' or the contents of its working memory. Formulae of the form $X^{\leq n}\varphi_1$ describe agents which produce a certain message or input to the system within n time steps. When φ_1 is of the form $B_i\ \mathsf{Ask}(i,j,P)$ or $B_i\ \mathsf{Tell}(i,j,P)$ these two cases result in communication with the other agent as follows: when the beliefs appear (as an Ask or a Tell) in the abstract agent i's working memory, they are also copied to agent j's working memory at the next cycle. Formulae of the form $B_i\ P$ represent the fact that beliefs other than Ask and Tell may also appear in the abstract agent i's working memory within n time steps. This is not critical to how abstract agents interact with communication, however it describes agent i's own behaviour.

The $G(\varphi_2 \rightarrow X^{\leq n}\varphi_3)$ formulae describe agents which are always guaranteed to reply to a request for information within n timesteps. The abstract agent i interacts with communication as follows: if t is the timestamp when abstract agent i came to believe formula $\mathsf{Ask}(j,i,P)$ (agent j asked for P), then the formula $\mathsf{Tell}(i,j,P)$ must

appear in the working memory of agent i within $t + n$ steps. The formula $\mathsf{Tell}(i, j, P)$ is then copied to agent j's working memory at the next cycle. Note that we do not need the full language of LTL (for example, the Until operator) in order to specify abstract agents.

6.5 Example

To illustrate the use of the proposed framework, let us consider an example system consisting of two agents: where one is concrete, the other is abstract. We consider the following two different scenarions.

Scenario 1**:**

Agent 1 which is a concrete agent has the following set of rules:

$$Rule1 \quad < 1 \ : \ P \rightarrow \mathsf{Ask}(1, 2, Q) >$$

$$Rule2 \quad < \ 2 \ : \mathsf{Tell}(2, 1, Q) \rightarrow Q >$$

$$Rule3 \quad < \ 3 \ : \ P \ \wedge \ Q \rightarrow R >$$

The first rule states that if P then ask the abstract agent 2 whether Q is the case. The second rule is a trust rule for agent 1 which makes it trust 2 when 2 informs it that Q is the case. The third rule states that if P and Q then R. The external behaviour of the abstract agent 2 is described by the following temporal logic formula:

$$G(B_2 \ \mathsf{Ask}(1, 2, Q) \ \rightarrow \ X^{\leq 4} \ B_2 \ \mathsf{Tell}(2, 1, Q))$$

Suppose now that the initial working memory of the agents contain the following patterns: $WM_1 : \{[0 \ : \ P]\}$ and $WM_2 : \{\ \}$.

Table 6.1 gives a simple example of a run of the system starting from the initial configuration. This example helps to explain how facts are derived and communicated, and what happens when the abstract agent receives an Ask query by communication.

Time	Agent 1	Agent 2
0	$\{[0:P]\}$	$\{\ \}$
	Rule1	Idle
1	$\{[0:P]\ [1:\mathsf{Ask}(1,2,Q)]\}$	$\{\ \}$
	Idle	Copy (Ask(1,2,Q) from Agent 1)
2	$\{[0:P]\ [1:\mathsf{Ask}(1,2,Q)]\}$	$\{[2:\mathsf{Ask}(1,2,Q)]\}$
	Idle	Idle
3	$\{[0:P]\ [1:\mathsf{Ask}(1,2,Q)]\}$	$\{[2:\mathsf{Ask}(1,2,Q)]\}$
	Idle	Idle
4	$\{[0:P]\ [1:\mathsf{Ask}(1,2,Q)]\}$	$\{[2:\mathsf{Ask}(1,2,Q)]\}$
	Idle	Idle
5	$\{[0:P]\ [1:\mathsf{Ask}(1,2,Q)]\}$	$\{[2:\mathsf{Ask}(1,2,Q)]\}$
	Idle	Tell
6	$\{[0:P]\ [1:\mathsf{Ask}(1,2,Q)]\}$	$\{[2:\mathsf{Ask}(1,2,Q)]\ [6:\mathsf{Tell}(2,1,Q)]\}$
	Copy (Tell(2,1,Q) from Agent 2)	Idle
7	$\{[0:P]\ [1:\mathsf{Ask}(1,2,Q)]$ $[7:\mathsf{Tell}(2,1,Q)]\}$	$\{[2:\mathsf{Ask}(1,2,Q)]\ [6:\mathsf{Tell}(2,1,Q)]\}$
	Rule2	Idle
8	$\{[0:P]\ [1:\mathsf{Ask}(1,2,Q)]$ $[7:\mathsf{Tell}(2,1,Q)]\ [8:Q]\}$	$\{[2:\mathsf{Ask}(1,2,Q)]\ [6:\mathsf{Tell}(2,1,Q)]\}$
	Rule3	Idle
9	$\{[0:P]\ [1:\mathsf{Ask}(1,2,Q)]$ $[7:\mathsf{Tell}(2,1,Q)]\ [8:Q]\ [9:R]\}$	$\{[2:\mathsf{Ask}(1,2,Q)]\ [6:\mathsf{Tell}(2,1,Q)]\}$

Table 6.1: Example: derivation scenario 1

Note that in the above derivation it is assumed that, at step 2 when abstract agent 2 came to believe formula $\mathsf{Ask}(1,2,Q)$ (agent 1 asked for Q) the formula $\mathsf{Tell}(2,1,Q)$ appeared in the working memory of agent 2 at $2+4$ i.e., at the 6th step. However, the formula $\mathsf{Tell}(2,1,Q)$ could also appear at any of the 3rd, 4th, or 5th steps but definitely appear at step 6 if it is not already present in the working memory of agent 2. In the above run, at the 3rd, 4th, and 5th steps both the agents perform an Idle action.

Scenario 2:

Agent 1 which is a concrete agent has the following set of rules:

$$Rule2 \quad < \ 2 \ : \mathsf{Tell}(2,1,Q) \rightarrow Q >$$

$$Rule3 \quad < \ 3 \ : \ P \ \wedge \ Q \rightarrow R >$$

The external behaviour of the abstract agent 2 is described by the following tem-

poral logic formula:

$$X^{\leq 5}\ B_2\ \mathsf{Tell}(2,1,Q)$$

that is abstract agent 2 spontaneously generates a Tell. Suppose now that the initial working memory of the agents contain the following patterns: $WM_1 : \{[0 : P]\}$ and $WM_2 : \{\ \}$. Table 6.2 gives a simple example of a run of the system starting from the initial configuration. This example helps to explain how facts are derived and communicated, and what happens when an abstract agent spontaneously generates a Tell (similarly we can show for an Ask).

Time	Agent 1	Agent 2
0	$\{[0 : P]\}$	$\{\ \}$
operation:	Idle	Tell
1	$\{[0 : P]\}$	$\{[1 : \mathsf{Tell}(2,1,Q)]\}$
operation:	Copy (Tell(2,1,Q) from Agent 2)	Idle
2	$\{[0 : P]\ [2 : \mathsf{Tell}(2,1,Q)]\}$	$\{[1 : \mathsf{Tell}(2,1,Q)]\}$
operation:	Rule2	Idle
3	$\{[0 : P]\ [2 : \mathsf{Tell}(2,1,Q)]\ [3 : Q]\}$	$\{[1 : \mathsf{Tell}(2,1,Q)]\}$
operation:	Rule3	Idle
4	$\{[0 : P]\ [2 : \mathsf{Tell}(2,1,Q)]\ [3 : Q]\ [4 : R]\}$	$\{[1 : \mathsf{Tell}(2,1,Q)]\}$

Table 6.2: Example: derivation scenario 2

In this derivation it is assumed that, the formula $\mathsf{Tell}(2,1,Q)$ appeared in the working memory of agent 2 at the 1st step. However, the formula $\mathsf{Tell}(2,1,Q)$ could also appear at any of the 2rd, 3rd, 4th, or 5th steps but definitely appear at step 5 if it is not already present in the working memory of agent 2.

In both scenarios, interesting properties of the system that can be verified include, e.g., $X^n\ B_1\ R$.[3]

6.6 Discussion

Abstraction is a key technique in handling the state space explosion problem in model checking. A number of abstraction approaches have been proposed for verifying (soft-

[3]Recall that we can use $X^n\ B_1\ R$ to verify whether $X^{\leq n}\ B_1\ R$ holds.

ware/hardware) designs of industrial complexity, e.g., [Gallardo et al., 2002, Clarke et al., 1994, Cousot and Cousot, 1977]. Our use of abstraction is however different from such classic approaches which use a mapping between an abstract transition system and a concrete program. Depending on this mapping, verification results may be correct but not complete. By correct or conservative abstraction usually mean that if a formula is true in the abstract system, then it is true in the concrete system (but if a formula is false in the abstract system, it may not be false in the concrete system). In contrast, our approach uses a very specific kind of abstraction, which replaces a concrete agent with an abstract one that implements guarantees of its response time behaviour. If those guarantees are correct, then our approach gives both correct and complete results. Complete or exact abstraction means that a formula is true in the abstract system if and only if it is true in the concrete system. Agents can be modelled as abstract if their response time guarantees have already been verified or the system designer is prepared to assume them.

In the literature, there have been many other approaches to alleviate the state space explosion problem, including verification approaches based on compositional reasoning [Berezin and Clarke, 1998]. In compositional reasoning, a property φ to be verified is decomposed into sub-properties that describe the behaviour of small components of the system. The sub-properties are verified for the corresponding components. Then the system satisfies φ if all the sub-properties are satisfied locally and their conjunction implies φ. In contrast, our approach to verification using abstraction does not decompose φ into sub-properties. The property φ is verified in the whole system. However, we construct the system using a hierarchical composition in which the LTL properties can be previously verified properties of non-abstract versions of an abstract agent or set of abstract agents.

In the subsequent chapters, we implement the approach to verification described above.

Chapter 7

Automated verification tool for MAS

In this chapter, we describe an encoding based on the Maude rewriting system which implements the approach to verification described in the preceding chapter. We then describe an automated verification tool, TVRBA, which generates an encoding of a system of communicating rule-based agents for the Maude LTL model checker, which is then used to verify the desired properties of the system. TVRBA allows the system designer to specify the information about agents' interaction, behaviour, and execution strategy at different levels of abstraction. We chose the Maude LTL model checker because it can model check systems whose states involve arbitrary algebraic data types. The only assumption is that the set of states reachable from a given initial state is finite. This simplifies modelling of the agents' (first-order) rules and reasoning strategies. For example, the variables appear in a rule can be represented directly in the Maude encoding, without having to generate all ground instances resulting from possible variable substitutions.

7.1 Maude rewriting system and formal verification

Maude is a high-level declarative programming language that models systems and the actions within those systems [Clavel et al., 2007, 2008]. The Maude system integrates an equational style of functional programming with rewriting logic computa-

tion. Equations are useful for creating and mapping out structures. Rewriting rules can be used to represent transitions that occur within and between structures. Maude's formal tools, such as its inductive theorem prover, LTL model checker, and breadth-first search capability have been used successfully in several applications, for example [van Riemsdijk et al., 2007, Astefanoaei et al., 2008, Alpuente et al., 2009], among others. In this section, we present the basic foundation of Maude following [Clavel et al., 2007, 2008] and give an overview of Maude LTL model checking.

7.1.1 Basic foundation of Maude

A rewriting theory $\mathcal{R}$ = (Σ, E, R), consists of a signature Σ, a set E of equations, and a set R of rules. The static part of a system is specified in an equational sublogic of rewriting logic (membership equational logic) by means of equations E. The system dynamics (concurrent transitions or inferences) is specified by means of rules R that rewrite terms, representing parts of the system, into other terms. The rules in R are applied *modulo* the equations in E. Thus, data types are defined algebraically by equations and the dynamic behaviour of a system is defined by rewrite rules which describe how a part of the state can change in one step. A rewrite theory is often non-deterministic and could exhibit many different behaviours.

7.1.1.1 Maude modules

The fundamental concept of Maude is the *module*, which represents the basic units of specification and programming. A module is essentially a collection of sorts and a set of operations on these sorts. In Maude there are two kinds of modules: functional modules and system modules. Each module is declared with the key terms:

```
fmod <ModuleName> is
  <DeclarationsAndStatements>
endfm

mod <ModuleName> is
  <DeclarationsAndStatements>
endm
```

where a functional module begins with `fmod` keyword and ends with `endfm` keyword, and a system module begins with `mod` keyword and ends with the keyword `endm`. The `<ModuleName>` represents the name of the module, and the body of a module `<DeclarationsAndStatements>` represents all the declarations and statements in between the beginning of the module and the end of it. The body of a functional module `<DeclarationsAndStatements>` defines data types and operations on them by means of equational theory E only. In contrast, the body of a system module `<DeclarationsAndStatements>` specifies a rewrite theory, which contains an equational theory E plus rewriting rules R.

7.1.1.2 Sorts and subsorts

A `sort` is a category for values. It is declared within the module, with the key word `sort` and a period at the end. Multiple sorts may be declared using the `sorts` keyword.

```
sort Animal .
sorts Mammal Bird Color .
```

The subsort relation on sorts are just like the subset relation on the sets of elements in the intended model of these sorts. A subsort relation is declared using the keyword `subsort`. The following declaration states that the sort `Mammal` is a subsort of the sort `Animal`.

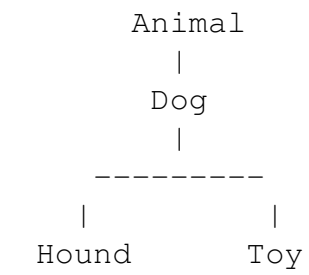
```
subsort Mammal < Animal .
```

7.1.1.3 Kinds

In Maude, sorts are grouped into equivalence classes called `kinds`. Two sorts are grouped together in the same equivalence class if and only if they belong to the same connected component. Consider the following declarations.

```
sort Animal .
subsort Dog < Animal .
subsorts Hound Toy < Dog .
```

Then we create a graph of connected sorts. The following entire graph is a connected component. Where `[Animal]` represents the `kind` of this connected component. If we were to declare another sort `Vegetable` and not declare any subsort relations with any sort on this graph, `Vegetable` would not be part of the connected component. `Vegetable` would be its own, separate, singleton connected component.



7.1.1.4 Maude operators

Maude operators are used as constructors and functions on data. An operator is declared with the keyword `op` followed by its name, followed by a colon, followed by the list of sorts for its arguments, followed by `->`, followed by the sort of its result, optionally followed by an attribute declaration, followed by white space and a period. The following declaration represent the general scheme

```
op <OpName> : <Sort-1> ... <Sort-k> -> <Sort>
                    [<OperatorAttributes>] .
```

where `<Sort-1> ... <Sort-k>` are called domain `sorts` and `<Sort>` is the range `sort` of the operator `<OpName>`. The arity and coarity pair is called the rank of the operator. The keyword `ops` can be used to declare multiple operators that have a same rank. For example, the declaration

```
ops <OpName-1> ... <OpName-n> : <Sort-1> ... <Sort-k>
              -> <Sort> [<OperatorAttributes>] .
```

represent the general scheme of multiple operators. An operator attribute could be commutative (comm), associative (assoc), identity (id), constructor (ctor), iterated (iter) etc., that provide additional information about the operator. The users can define

a (context free) grammar of operators, e.g.,

```
op _+_ : Nat Nat -> Nat .
op if_then_else_fi : Bool Nat Nat -> Nat .
```

where the underscores "_" indicate places where actual arguments are put, for example, $0 + 0$ and `if N > M then N else M fi`.

7.1.1.5 Maude constants

In the operator declaration if the argument list of the operator is empty, then the operator is called a *constant*. For example, the following declarations state that `cat` is a constant of sort `Animal` and `red`, `blue`, and `yellow` are constants of sort `Colour`.

```
op cat : -> Animal .
ops red blue yellow : -> Colour .
```

7.1.1.6 Maude variables

A Maude variable is constrained to range over a particular sort (or kind), i.e., it is an indefinite value for a sort. A Maude variable is declared in a module using the keyword `var` followed by the variable name, followed by a colon with white space before and after, followed by its sort (or its kind), followed by white space and a period. The following declaration

```
var x : Colour .
```

states that `x` is a variable of sort `Colour`. The keyword `vars` can be used to declare multiple variables of the same sort. For example, the following declaration

```
vars x y : Colour .
```

states that `x` and `y` are variables of sort `Colour`. Maude variables can also be declared on-the-fly with syntax consisting of the variable name, followed by a colon, followed by its sort. For example, `x : Colour` declares a variable named `x` of sort `Colour`. Note that the variables in Maude do not represent memory locations like variables in C++ and Java and other programming languages. Maude variables never

have a definite value assigned to them, and they do not carry values from one operation to other.

7.1.1.7 Terms

A term is either a constant, a variable, or the application of an operator to a list of argument terms. A ground term is a term containing no variables, but only constants and operators.

7.1.1.8 Equations

Unconditional equations. Unconditional equations are declared using the keyword `eq`, followed by an (optional) `[<LabelName>] :`, followed by a `term` (its left hand side), the equality sign =, then a `term` (its right hand side), optionally followed by a list of statement attributes.

```
eq [<LabelName>] : <Term-1> = <Term-2>
          [<OptionalStatementAttributes>] .
```

Conditional equations. The general form of conditional equations is the following:

```
ceq [<LabelName>] : <Term-1> = <Term-2> if <EqCond-1>
 /\ ... /\ <EqCond-k> [<OptionalStatementAttributes>] .
```

In Maude equations, variables appearing in the right-hand side term must also appear in its left-hand side term.

7.1.1.9 Rewrite rules

Equations are extremely useful for describing static part of a system, however, the real power of Maude is to provide concurrent transitions that occur within and between structures in the system. These transitions are achieved by means of rewriting rules.

Unconditional rules. Unconditional rules are declared using the keyword `rl`, followed by an (optional) `[<LabelName>] :`, a `term` (its left hand side), the Rightarrow sign =>, then a `term` (its right hand side).

```
rl [<LabelName>] : <Term-1> => <Term-2> .
```

Conditional rules. Conditional rules are declared using the following syntax:

```
crl [<LabelName>] : <Term-1> => <Term-2> if
        <RuleCond-1> /\ ... /\ <RuleCond-k> .
```

The conditions could be equations, membership conditions, or other rewriting rules. In the rules, variables appearing in the right-hand side term must also appear in its left-hand side term.

7.1.1.10 Module importation

Like most programming languages, Maude supports module importations, i.e., a Maude module can be imported as a submodule of another. However, in Maude, module importation cannot be cyclic: if module A imports module B, then module B cannot import module A.

7.1.2 Verifying systems using Maude

In Maude the model checking process comes in two flavours. Maude supports model checking invariants through search. This is essentially a breadth-first search strategy for verifying safety properties of the systems. The breadth-first search command has an optional argument n stating the maximum depth of the search. In this case, model checker does not explore all reachable states, but only those states reachable within a certain depth bound n, known as bounded model checking. Maude also provides an LTL model checker [Eker et al., 2003] that enables the verification of LTL properties of rewriting systems.

Like any other model checking tool, verification in Maude requires a system specification and a property specification. The system specification is provided by a rewrite theory, whereas the property specification is given by linear temporal logic. Maude supports on-the-fly LTL model checking for initial states $[t]$, say of sort $State$, of a rewrite theory $\mathcal{R}$ such that the set of reachable states $\{[t'] \in T_{\Sigma/E} \mid \mathcal{R} \vdash [t] \rightarrow [t']\}$

from $[t]$ is finite. Model checking is performed by constructing a Büchi automaton from the negation of the property formula and lazily searching the synchronous product of the Büchi automaton and the system state transition graph for a reachable accepting cycle. The double depth-first algorithm of [Holzmann et al., 1996] is used to lazily generate and search the synchronous product.

7.2 Maude encoding

In this section, we describe how we implement the approach to verification which uses strategies and abstraction as a Maude rewriting system.

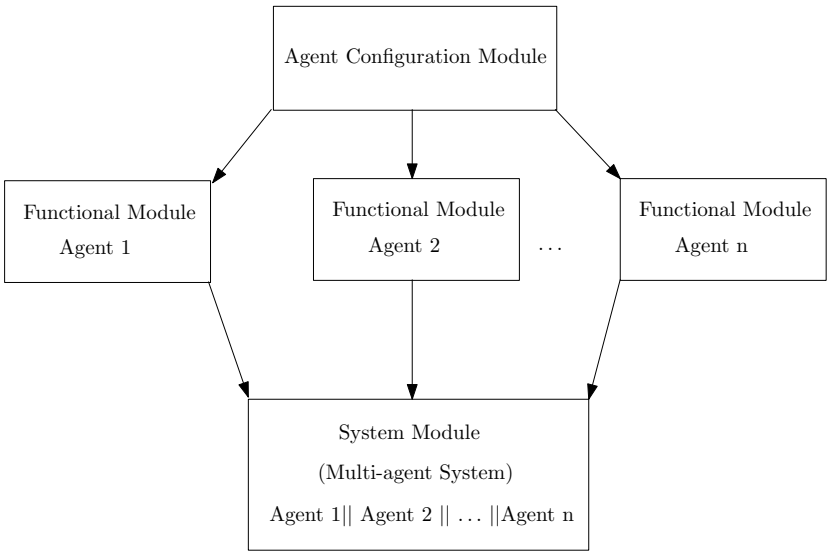


Figure 7.1: System implementation structure in Maude

We take advantage of Maude's modular structuring mechanisms to implement our system design. We use a generic functional module and a set of functional and system modules to represent the system. The overall picture of our implementation is shown in Figure 7.1.

Throughout this chapter we will use verbatim texts to represent specification of the agents into Maude. Therefore, an agent i will be denoted by `i` in Maude specification. Similarly, `Ask(i,j,P)` will have the same meaning as Ask(i, j, P) and so on.

7.2.1 Agent configuration module

Each agent in a MAS has a configuration (local state) and the composition of all these configurations (local states) make the configuration (global state) of the MAS. The types necessary to implement the local state of an agent (working memory, program, reasoning strategy, message counters, timestep etc.) are declared in a generic agent configuration functional module called ACM. The structure of the ACM is given in Listing 7.1.

```
fmod ACM is
  protecting NAT .
  protecting BOOL .
  protecting QID .
  sorts Constant Pattern Term Rule Agenda WM .
  sorts TimeP TimeWM RepT RepTime Config .
  subsort Pattern < WM .
  subsort Rule < Agenda .
  subsort Qid < Constant .
  subsort TimeP < TimeWM .
  subsorts Constant < Term .
  subsort RepT < RepTime .
  ops void rule : -> Pattern .
  ops com exec : -> Phase [ctor] .
  op nil : -> Term[ctor] .
  op _|_ : Term Term -> Constant [ctor assoc] .
  op [_] : Term -> Term [ctor] .
  op [_ : _] : Nat Pattern -> TimeP .
  op _ _ : WM WM -> WM [comm assoc] .
  op _ _ : TimeWM TimeWM -> TimeWM [comm assoc] .
  op _ _ : Agenda Agenda -> Agenda [comm assoc] .
  op <_ : _->_> : Nat TimeWM TimeP -> Rule .
  op _ _ : RepTime RepTime -> RepTime [comm assoc] .
  op Ask : Nat Nat Pattern -> Pattern .
  op Tell : Nat Nat Pattern -> Pattern .
  op Naf : Pattern -> Pattern .
    .
    .
    .
endfm
```

Listing 7.1: Sorts declaration and their relationships

A number of Maude library modules such as NAT, BOOL, and QID have been imported into the ACM functional module. The modules NAT and BOOL are used to define natural and Boolean values, respectively, whereas the module QID is used to define the set of constant symbols (constant terms of the rule-based system). The set of variable symbols (variable terms of the rule-based system) are simply Maude variables of sort QID. Both variables and constants are subsorts of sort Term. A function (function terms of the rule-based system) is simply declared as an operator

whose arguments are of sort `Term`, and returns an element of sort `Term`. Similarly, a pattern (fact/predicate of the rule-based system) is declared as an operator whose arguments are of sort `Term`, and returns an element of sort `Pattern`. Therefore, the arguments of a pattern may contain functions, constants, and variables all of which are of sort `Term`. The sort `Pattern` is declared as a subsort of the sort `WM` (working memory), and a concatenation operator is declared on sort `WM` which is the double underscore shown below.

```
op _ _ : WM WM -> WM [comm assoc] .
```

The above operation is in mixfix notation and it is commutative and associative. This means that working memory elements are a set of patterns whose order does not matter. In order to maintain time stamp for each pattern, a sort `TimeP` is declared whose elements are of the form `[ t : P ]`, where `t` represents the time stamp of pattern `P` that indicating when that pattern was added to the working memory. The sort `TimeP` is declared as a subsort of the sort `TimeWM`, and a concatenation operator is declared on sort `TimeWM` which is also the double underscore and commutative and associative.

```
op _ _ : TimeWM TimeWM -> TimeWM [comm assoc] .
```

Note that updating of `WM` and `TimeWM` take place simultaneously, for example, whenever a pattern `P` is added to `WM` the corresponding element `[ t : P ]` is also added to `TimeWM` for an appropriate time cycle `t`. Fact time stamps are maintained to implement reasoning strategies. In § 7.2.5 we give a detailed analysis of why we maintain two working memories.

The rules of each agent are defined using an operator which takes as arguments a sort `Nat` specifying the priority, a set of patterns (of sort `TimeWM`) specifying the antecedents of the rule and a single patten (of sort `TimeP`) specifying the consequent, and returns an element of sort `Rule`. The sort `Rule` is declared as a subsort of the sort `Agenda`, and a concatenation operator is declared on sort `Agenda` which is also the double underscore and commutative and associative.

```
op _ _ : Agenda Agenda -> Agenda [comm assoc] .
```

The sort `RepT` (reply/generate an Ask or a Tell within a given time bound) is used to implement the behaviour of abstract agents. The sort `RepT` is declared as a subsort of the sort `RepTime`, and a concatenation operator is declared on sort `RepTime` which is also the double underscore and commutative and associative.

```
op _ _ : RepTime RepTime -> RepTime [comm assoc] .
```

Each element of `RepTime` consists of a time bound (based on the value of n of an LTL formula discussed in § 6.4) and a pattern.

```
------Checking if a pattern is in the working memory--
   var p : Pattern .
   var M : WM .
   op inWM : Pattern WM -> Bool .
   eq inWM(p, p) = true .
   eq inWM(p, p M) = true .
   eq inWM(p, M) = false [owise] .

-----Checking if a rule-instance is in the agenda-----
   var r : Rule .
   var RL : Agenda .
   op inAgenda : Rule Agenda -> Bool .
   eq inAgenda(r,r) = true .
   eq inAgenda(r, r RL) = true .
   eq inAgenda(r, RL) = false [owise] .

-----Checking if an element is in RepTime-------------
   var rt : RepT .
   var RT : RepTime .
   op inRT : RepT RepTime -> Bool .
   eq inRT(rt, rt ) = true .
   eq inRT(rt, rt RT) = true .
   eq inRT(rt, RT) = false [owise] .
```

Listing 7.2: Checking the existence of an element

The data types presented in Listing 7.1 are manipulated using a set of equations. The equations are used for various purposes: for example, to check, whether or not a given pattern (used to represent fact/predicate) is already in the agent's working memory, whether or not a rule instance is already in the agenda etc. Some illustrative examples of the Maude implementation are given in Listing 7.2.

Additional equations are used to implement reasoning strategies, e.g., to determine the highest priority rule instance in the agenda, or the pattern with highest time stamp in working memory etc. Listing 7.3 illustrates the implementation of the rule priority

reasoning strategy.

```
var n : Nat .
var Ant : TimeWM .
var Cons : TimeP .
var A RL : Agenda .
op void-rule : -> Rule .
eq void-rule = < 0 : [ 0 : void ] -> [ 0 : rule ] > .
op priority : Rule -> Nat .
eq priority( < n : Ant -> Cons >) = n .
op max : Agenda -> Nat .
eq max(r) = priority(r) .
eq max(r A) = if priority(r) > max(A)
              then priority(r) else max(A) fi .
op strategyRule : Agenda Agenda -> Agenda .
ceq strategyRule(r A, RL) = if priority(r) >= max(RL)
                      then r strategyRule(A, RL)
                       else strategyRule(A, RL)
                       fi if r =/= void-rule .
eq strategyRule(voidrule, RL) = void-rule .
```

Listing 7.3: Strategy implementation: an example

7.2.2 Implementation of agent modules

We model each (concrete and abstract) agent using a functional module which imports the `ACM` module defined above. The local configuration of an agent i is represented as a tuple `Si[A|RL|TM|M|RT|RT'|t|msg|syn]iS`, where `Si` and `iS` indicate start and end of a state of agent i. The variables `A` and `RL` are of sort `Agenda`, `TM` is of sort `TimeWM`, `M` is of sort `WM`, `RT` and `RT'` are of sort `RepTime`. Moreover, `t`, `msg`, and `syn` are of sort `Nat`. The variables `t`, `msg`, and `syn` have been used to represent respectively the time step, message counter, and a flag for synchronisation. Note that the structure of local configurations for both concrete and abstract agents are the same. This is to maintain consistency of the shape of each agent's configuration. However, for example, the sort `RepTime` is of no use for concrete agents and its value is always empty for them.

7.2.2.1 Concrete agent module

For each concrete agent i there is a corresponding `ConcreteAgent-i` module, whose structure is given in Listing 7.4. In this module, we declare the configuration `Si[A|RL|TM|M|RT|RT'|t|msg|syn]iS` which represents the local state

of the agent. We also declare a set of operators that are used in various equations, e.g., to represent rules of the agents, to compute set of rule instances based on the current working memory patterns, and to select a set of rule instances from the agenda based on a given reasoning strategy.

```
fmod ConcreteAgent-i is
 protecting ACM .
 op Si[_|_|_|_|_|_|_|_|_]iS : Agenda Agenda TimeWM WM
                RepTime RepTime Nat Nat Nat -> Config .

 op ruleIns-i : Agenda TimeWM WM -> Agenda .
 op matchRule-i : Config -> Config .
 op selectRule-i : Config -> Config .
 .
 .
 .
endfm
```

Listing 7.4: Structure of concrete agent module

The representation of rules, generating rule instances, and selecting a set of rule instances based on a given reasoning strategy is specified as a set of equations. The following equations represent concrete agent i's set of rules.

```
ceq  [Rule1] :   ruleIns-i(A, Ant1 TM, M) = < n1 :  Ant1 -> Cons1 >

        ruleIns-i(< n1 :  Ant1 -> Cons1 > A, Ant1 TM, M) if

(not inAgenda(< n1 :  Ant1 -> Cons1 >, A)) ∧ (not inWM(pattern(Cons1), M)) .


ceq  [Rule2] :   ruleIns-i(A, Ant2 TM, M) = < n2 :  Ant2 -> Cons2 >

        ruleIns-i(< n2 :  Ant2 -> Cons2 > A, Ant2 TM, M) if

(not inAgenda(< n2 :  Ant2 -> Cons2 >, A)) ∧ (not inWM(pattern(Cons2), M)) .

                                ⋮

ceq [Rulek] : ruleIns-i(A, Antk TM, M) = < nk :  Antk -> Consk >

        ruleIns-i(< nk :  Antk -> Consk > A, Antk TM, M) if

(not inAgenda(< nk :  Antk -> Consk >, A)) ∧ (not inWM(pattern(Consk), M)) .


    eq [Default] : ruleIns-i(A, TM, M) = void-rule [owise] .
```

The left-hand side of each equation uses the operator `ruleIns-i` (that takes arguments an element of sort `Agenda`, an element of sort `TimeWM`, and an element

of sort WM, and returns an element of sort Agenda) which matches the antecedents Antk for each rule < nk : Antk -> Consk >, and the right-hand side represents the corresponding rule. The ruleIns-i operator calls itself recursively so that each equation may generate possibly multiple rule instances. The operator pattern is declared and defined in the ACM module. It takes as argument an element of sort TimeP and returns the corresponding pattern of sort Pattern, i.e., it is applied to [n : P] and returns the corresponding pattern P. In the following we give the semantics of the match equation (labelled as Match-i) which is used to generates a set of rule instances based on the current working memory patterns.

```
eq [Match-i] :  matchRule-i(Si[A|R|TM|M|RT|RT'|t|msg|1]iS)
                                =
    Si[ruleIns-i(A,TM,M) A|RL|TM|M|RT|RT'|t|msg|2]iS .
```

In the Match-i equation, the operator matchRule-i is applied on the agent's configuration to compute the set of rule instances based on the current working memory patterns. The operator ruleIns-i is called in the equation Match-i. When the equation Match-i executes, the operator ruleIns-i is called, consequently ruleIns-i matches from the topmost equation Rule1 and goes to the bottom Rulek to generate all possible rule instances recursively, and finally exits using the Default equation that uses Maude's [owise] attribute. Maude's [owise] attribute allows its equation to be applied only if all other equations for the same top term having more-specific left-hand sides fail to match.

The select equation (labelled as Select-i) is used to select a set of rule instances from the agenda based on a given reasoning strategy.

```
eq [Select-i] :  selectRule-i(Si[A|RL|TM|M|RT|RT'|t|msg|2]iS)
                                =
Si[delete(strategy(A,A), A)|strategy(A,A) RL|TM|M|RT|RT'|t|msg|3]iS .
```

The left-hand side of the above equation applies the operator selectRule-i on the agent's configuration to select a set of rule instances from the current agenda based on a given reasoning strategy. Two operators strategy and delete are called

from this equation. The `strategy` operator is applied on the current agenda that returns a set of rule instances, whereas the `delete` operator is used to delete those rule instances selected from the agenda. These operators are implemented in the `ACM` module so that they can be used by all the concrete agents in the system.

The operators `matchRule-i` (used in `Match-i`) and `selectRule-i` (used in `Select-i`) are called from Maude rules that implement concrete agent i's inference engine in the `MAS` system module (cf. 7.2.3). The communication for a concrete agent i works with the other agent j as follows: when the beliefs appear (as an `Ask(i,j,P)` or a `Tell(i,j,P)`) in the concrete agent i's working memory, they are also copied to agent j's working memory at the next cycle. Communication between agents is also implemented using Maude rules in the `MAS` system module.

7.2.2.2 Abstract agent module

For each abstract agent j there is a corresponding `AbstractAgent-j` module, whose structure is given in Listing 7.5. In this module, we declare the configuration `Sj[A|RL|TM|M|RT|RT'|t|msg|syn]jS` which represents the local state of the agent. We also declare a set of operators that are used in various equations, e.g., to compute the timing information of a pattern based on an LTL formula, to select elements of `RepTime` that represent the information regarding when j must reply to requests made by the other agents in the system, or when j must produce a message or input to the system.

```
fmod AbstractAgent-j is
  protecting ACM .
  op Sj[_|_|_|_|_|_|_|_]jS : Agenda Agenda TimeWM WM
                RepTime RepTime Nat Nat Nat -> Config .

  op setTime-j : Config -> Config .
  op proPattern-j : Config -> Config .
  op tbound-j : TimeWM RepTime -> RepTime .
  .
  .
  .
endfm
```

Listing 7.5: Structure of abstract agent module

The external behaviour of an abstract agent is represented by means of temporal formulae that include belief operators. These formulae are translated into Maude agent specifications. We now explain how we represent temporal formulae in Maude. In the configuration, the variables `RT` (fifth entry from the left) and `RT'` (sixth entry from the left) of sort `RepTime` are used to implement the external behaviour of the abstract agent. The following operators are declared in the `ACM` module:

```
op pro :  Pattern Nat -> RepTime [ctor] .
op empty-RT : -> RepTime [ctor] .
```

Therefore, the elements of `RepTime` can be of the following form:

```
pro(Ask(j,i,P),m)
pro(Tell(j,i,P),m)
empty-RT
```

where `pro(Ask(j,i,P),m)` states that an `Ask(j,i,P),m)` must be produced in `m` time steps. Similarly, the element `pro(Tell(j,i,P),m)` states that: a `Tell(j,i, P),m)` must be produced in `m` time steps. In the configuration, the fifth entry is used to store the elements defined above, and the sixth entry maintains a subset of these elements to be generated at the current cycle time.

First, we discuss how we construct the elements defined above based on LTL formulae. Recall that LTL formulae are of the form $X^n\ \varphi_1$ and $G(\varphi_2 \rightarrow X^n\ \varphi_3)$, where φ_1, φ_2, and φ_3 are ground atomic formulae.

LTL formulae of the form $X^n\ \varphi_1$: The element `pro(Ask(j,i,P), n)` can be constructed from the corresponding LTL formula $X^n\ B_j\ \mathsf{Ask}(j, i, P)$. Similarly, the element `pro(Tell(j,i,P),n)` can be constructed from the corresponding LTL formula $X^n\ B_j\ \mathsf{Tell}(j, i, P)$. In the Maude specification, these elements are added initially in the fifth entry of the configuration. For example, let us assume that the external behaviour of an abstract agent $j = 2$ is described using a set of LTL formulae that include $X^4\ B_2\ \mathsf{Tell}(2, 1, PremCust(Miller))$: in 4 time steps agent 2 will believe that it tells agent 1 that Miller is a Premium Customer. Then the corresponding element `pro(Tell(2,1,PremCust(Miller)),4)` will be added initially to the fifth entry of abstract agent `2`'s configuration.

LTL formulae of the form $G(\varphi_2 \rightarrow X^n\ \varphi_3)$: The equation `TimeBound-j` computes the timing information regarding when a particular pattern must be produced based on an LTL formula $G(\ B_j\ \mathsf{Ask}(i,j,P) \rightarrow X^n\ B_i\ \mathsf{Tell}(j,i,P))$.

```
eq [TimeBound-j] :  setTime-j(Sj[A|RL|TM|M|RT|RT'|t|msg|1]jS)
                                        =
      Sj[A|RL|TM|M|tbound-j(TM,RT) RT|RT'|t|msg|2]jS .
```

In the above equation, the operator `tbound-j` is used to compute a set of elements of the form `pro(Tell(j,i,P),m)` based on the elements of the form `[ t : Ask(i,j,P) ]` of `TimeWM`.

$$\begin{aligned} \textit{tbound-j}(TM, RT) = \{\ & pro(\mathsf{Tell}(j,i,P), m)\ \mid \\ & pro(Tell(j,i,P), m) \notin RT \\ & \wedge\ \ [t : Ask(i,j,P)] \in TM \\ & \wedge\ \ (m = t + n - 1)\ \} \end{aligned}$$

The value of n is reduced by 1 to adjust the time step. This is due to the model of communication that requires a single time step when sending (receiving) messages to other agents (cf. § 7.2.3). For example, let us assume that the external behaviour of an abstract agent $j = 2$ is described using a set of LTL formulae that include the following formula:

$$G(\ B_2\ \mathsf{Ask}(1, 2, PremCust(Miller)) \rightarrow$$
$$X^7\ B_2\ \mathsf{Tell}(2, 1, PremCust(Miller))$$

The representation of LTL formulae in the Maude are given below:

```
                                  ⋮

ceq [Ltl1] :  tbound-2( [ t :  Ask(1,2,PremCust(Miller)) ] TM,RT)
                                        =
    pro(Tell(2,1,PremCust(Miller)), t+7) tbound-2(TM,RT) if
     (not inRT(pro(Tell(2,1,PremCust(Miller)), t+7),RT) ) .

                                  ⋮

  eq [Default] :  tbound-2(TM,RT) = empty-RT [owise] .
```

In the above equations, the operator `tbound-2` is applied recursively to compute all the elements at the current cycle time. That is, it constructs elements of the form `pro(Tell(2,1,PremCust(Miller)), t+7)`, upon the arrival of new requests of the form `[ t : Ask(1,2,PremCust(Miller)) ]` from other agents.

In the following, we give the semantics of the produce-pattern equation (labelled as `ProPattern-j`) which is used to select those patterns to be generated at the current cycle time.

```
eq [ProPattern-j] :  proPattern-j(Sj[A|RL|TM|M|RT|RT'|t|msg|2]jS)
                                  =
          Sj[A|RL|TM|M|RT|pull(RT,RT,t) RT'|t|msg|3]jS .
```

In the `ProPattern-j` equation, the operator `pull` is declared and defined in the `ACM` module. It takes arguments of sorts `RepTime` and `Nat`, and returns an element of sort `RepTime`. `RT''` represents those elements of the form:

```
pro(Ask(j,i,P),m)
pro(Tell(j,i,P),m)
```

such that `t < m`, where `t` is the current cycle time. Therefore, the elements of the sixth entry of the configuration will be used to produce (add to the working memory) `Ask(j,i,P)` or `Tell(j,i,P)` patterns at the current cycle time. This is achieved using two operators `genTime : RepTime Nat -> TimeWM` and `genM : RepTime -> WM`. These operators are declared and defined in the `ACM` module, and they are used to update the working memory of the abstract agent based on the values of `RT'` and `t`. The first operator `genTime` takes as an argument an element of sort `RepTime` and the current cycle time of sort `Nat`, and it returns an element of sort `TimeWM`. That is, `genTime` is applied to `(pro(Ask(j,i,P),m), t)` and produces the corresponding element `[ t : Ask(j,i,P) ]` of `TimeWM`. Similarly, it is applied to `(pro(Tell(j,i,P),m), t)` and produces the corresponding element `[ t : Tell(j,i,P) ]` of `TimeWM`. The second operator `genM` takes as an argument an element of sort `RepTime`, and it returns an element of sort `WM`. That is, `genM` is applied to `(pro(Ask(j,i,P),m))` and produces the corresponding el-

ement `Ask(j,i,P)` of `WM`. Similarly, it is applied to `(pro(Tell(j,i,P),m))` and produces the corresponding element `Tell(j,i,P)` of `WM`. When the operators `genTime` and `genM` are applied to all the elements of `RT'` (sixth entry of the configuration), `RT'` becomes empty and the whole process repeats in the next cycles.

The operators `setTime-j`, `proPattern-j`, `genTime`, and `genM` are called from Maude rules that implement the partial behaviour of an abstract agent in the `MAS` system module (cf. 7.2.3).

7.2.3 Implementation of the MAS module

Computation steps of multi-agent systems are represented by transitions, which take systems from one configuration to subsequent ones. Each agent in the system has its own local state and the composition of all these local states comprises the configuration (global state) of the multi-agent system. In every configuration (global state), all agents proceed simultaneously. Each agent changes its next local configuration, possibly depending on the current local configurations of the other agents in the system. However, there can be an alternative interleaved execution model, where at most one agent is allowed to act at any one time. It depends on the modelled system which execution model (interleaved or synchronous) is more realistic. If we count timesteps required by a system of agents to derive something, interleaved model gives rather pessimistic results because only one agent can 'think' at any single step and the rest are waiting. This makes sense if the agents run on the same processor. However if, as in most our examples, agents are running on different processors and can 'think' in parallel, a synchronous model is more realistic.

The `MAS` module imports all the agent modules and contains both functions and rewrite rules which are used to implement the dynamic behaviour of the system. The structure of the MAS module is given in Listing 7.6. The parallel composition of agent configurations in the system is achieved using the `_||_` operator. In the `MAS` module we declare a sort `masConfig` to represent the global configuration of the system.

We then define an operator <_,_> whose first argument is the composition of all the local configurations of the system and the second argument is a phase, and it returns an element of sort masConfig.

The masConfig moves through communication and execution phases. The communication phase simply says that if there is something to be communicated then do it, and then return to the execution phase. The inference engine of concrete agents and the partial behaviour of abstract agents are implemented using a set of rules: Generate, Choice, Apply, Idle, and Communication.

```
mod MAS is
  protecting ConcreteAgent-i .
  protecting AbstractAgent-j .
  .
  .
  .
  sort masConfig .
  sort Phase .
  ops com exec : -> Phase .
  var phase : Phase .
  op ruleIns-i : Agenda TimeWM WM -> Agenda .
  op matchRule-i : Config -> Config .
  op selectRule-i : Config -> Config .
  op _||_ : Config Config -> Config [comm assoc] .
  op <_,_> : Config Phase -> masConfig [ctor] .
  op copy : masConfig -> masConfig [ctor] .
  .
  .
  .
endm
```

Listing 7.6: Structure of MAS module

The Generate rule causes each concrete agent to generate its conflict set using the equation labelled as Match-i in the concrete agent module, and each abstract agent to update its RepTime using the equation labelled as TimeBound-j in the abstract agent module. The Generate rule is given below. Here each variable is associated with its corresponding agent index to make it clear which variables are used to store state information and for what agent.

```
rl [Generate] :  <...|| Si[Ai|RLi|TMi|Mi|RTi|RTi'|ti|msgi|1]iS ||...

      || Sj[Aj|RLj|TMj|Mj|RTj|RTj'|tj|msgj|1]jS ||...,phase >
                                  =>
  <...|| matchRule-i(Si[Ai|RLi|TMi|Mi|RTi|RTi'|ti|msgi|1]iS) ||...
|| setTime-j(Sj[Aj|RLj|TMj|Mj|RTj|RTj'|tj|msgj|1]jS) ||...,phase > .
```

In the above `Generate` rule and throughout the rest of the discussion in this section, it is assumed that `Si[Ai|RLi|TMi|Mi|RTi|RTi'|ti|msgi|syni]iS` represents the local configuration of a concrete agent i, and `Sj[Aj|RLj|TMj|Mj|RTj|RTj'|tj|msgj|synj]jS` represents that of an abstract agent j.

The `Choice` rule causes each concrete agent to apply its reasoning strategy using the equation labelled as `Select-i` in the concrete agent module, and each abstract agent to select a subset of `RepTime` whose time bounds are less than or equal to the current system cycle time using the equation labelled as `ProPattern-j` in the abstract agent module. The `Choice` rule is given below.

```
rl [Choice] :  <...|| Si[Ai|RLi|TMi|Mi|RTi|RTi'|ti|msgi|2]iS ||...

     || Sj[Aj|RLj|TMj|Mj|RTj|RTj'|tj|msgj|2]jS ||...,phase >
                                  =>
   <...|| selectRule-i(Si[Ai|RLi|TMi|Mi|RTi|RTi'|ti|msgi|2]iS) ||...

|| proPattern-j(Sj[Aj|RLj|TMj|Mj|RTj|RTj'|tj|msgj|2]jS) ||...,phase > .
```

The `Apply` rule causes each concrete agent to execute the rule instances selected for execution and each abstract agent to produce those patterns selected from `RepTime`. Each agent (concrete or abstract) of the system has an additional `Idle` rule, which advances the time of each agent, leaving everything else unchanged. The `Idle` rule has a depth parameter which specifies the maximum depth at which the rule can be applied.

The `Apply` rule of a concrete agent i (labelled as `Apply-i`) is given below. In this rule, the consequent of a rule instance is added to `WM` and `TimeWM` using the operators `pattern` and `time`, respectively. These operators are declared and defined in the `ACM` module. In the concrete agent module section, we have already mentioned `pattern`. The operator `time` takes as arguments an element of sort `TimeP` and an element of sort `Nat`, and returns an element of sort `TimeP`. That is, it is applied to `([ 0 : P ],t+1)` and returns the corresponding time pattern `[ t+1 : P ]`, thus associating the appropriate time cycle with `P`.

```
crl [Apply-i] :  <...|| Si[Ai|< n :  Ant -> Cons > RLi|Ant TMi|Mi|RTi|RTi'|ti|msgi|3]iS
```

```
||...|| Sj[Aj|RLj|TMj|Mj|RTj|RTj'|tj|msgj|3]jS ||...,exec >
                                    =>
<...|| Si[Ai|RLi|Ant time(Cons, ti+1) TMi|pattern(Cons) Mi|RTi|RTi'|ti+1|msgi|1]iS

||...|| Sj[Aj|RLj|TMj|Mj|RTj|RTj'|tj|msgj|3]jS ||...,exec > if
                    (not inWM(pattern(Cons), Mi)) .
```

The `Idle` rule of a concrete agent i (labelled as `Idle-i`) is given below. The `Idle` rule executes only when there are no rule instances to be executed, that is, when `RLi` contains only the default void rule. The application of the `Idle-i` rule advances the cycle time of the concrete agent i, leaving everything else unchanged.

```
crl [Idle-i] :  <...|| Si[Ai|RLi|TMi|Mi|RTi|RTi'|ti|msgi|3]iS

 ||...|| Sj[Aj|RLj|TMj|Mj|RTj|RTj'|tj|msgj|3]jS ||...,exec >
                                    =>
          <...|| Si[Ai|RLi|TMi|Mi|RTi|RTi'|ti+1|msgi|1]iS

||...|| Sj[Aj|RLj|TMj|Mj|RTj|RTj'|tj|msgj|3]jS ||...,exec > if (size(RLi)=1) .
```

The `Apply` rule of an abstract agent j (labelled as `Apply-j`) is given below. The rule executes when `RTj'` is non-empty. In this rule, the operators `genM` and `genTime` have been used to add patterns to `WM` and `TimeWM`, respectively. These operators have already been mentioned in the abstract agent module section.

```
crl [Apply-j] :  <...|| Si[Ai|RLi|Ant TMi|Mi|RTi|RTi'|ti|msgi|1]iS ||...||

        Sj[Aj|RLj|TMj|Mj|RTj|RTj'|tj|msgj|3]jS ||...,exec >
                                    =>
      <...|| Si[Ai|RLi|Ant TMi|Mi|RTi|RTi'|ti|msgi|1]iS ||...||

Sj[Aj|RLj|genTime(RTj',tj) TMj| genM(RTj') Mj|RTj|empty-RT|tj+1|msgj|1]jS ||...,exec >

                    if (sizeRT(RTj') > 1) .
```

The `Idle` rule of an abstract agent j (labelled as `Idle-j`) is given below. For abstract agents, the `Idle` rule executes non-deterministically along with the `Apply`

rule. However, the `Idle` rule for abstract agents cannot be applied in some situations: for example, if at the current cycle time `tj` the abstract agent j's working memory contains `[ t0 : Ask(i,j,P) ]`, where `t0` is the time stamp when abstract agent j came to believe that agent i asked for P and abstract agent j's behaviour is described by the formula $G(B_j\ \mathsf{Ask}(i,j,P) \rightarrow\ X^n\ B_i\ \mathsf{Tell}(j,i,P))$, then the `Idle` rule of agent j cannot be applied when `tj = t0+n-1`, forcing the agent j to reply at `t0+n-1` if it has not already done so. This is achieved using the `bound` operator that checks if there exists such a situation. The operator `bound` is declared and defined in the `ACM` module, which takes as arguments an element of sort `RepTime` and an element of sort `Nat`, and returns an element of sort `Bool`. The application of `Idle-j` rule advances the cycle time of the abstract agent j, leaving everything else unchanged.

```
crl [Idle-j] :  <...|| Si[Ai|RLi|TMi|Mi|RTi|RTi'|ti|msgi|1]iS

 ||...|| Sj[Aj|RLj|TMj|Mj|RTj|RTj'|tj|msgj|3]jS ||...,exec >
                                  =>
        <...|| Si[Ai|RLi|TMi|Mi|RTi|RTi'|ti|msgi|1]iS

 ||...|| Sj[Aj|RLj|TMj|Mj|RTj|RTj'|tj+1|1|1]jS ||...,exec >

                  if (not bound(RTj', tj)) .
```

These Maude rules are controlled using a flag (the last component of each local configuration) which ensures that the configurations of all the agents move forward in a synchronous way. When the last flag of each configuration becomes 1, the system will execute the following Maude rule to change the status of the `masConfig` from execute to communication.

```
rl [Exec-Comm] :  <...|| Si[Ai|RLi|TMi|Mi|RTi|RTi'|ti|msgi|1]iS

 ||...|| Sj[Aj|RLj|TMj|Mj|RTj|RTj'|tj|msgj|1]jS ||...,exec >
                                  =>
        <...|| Si[Ai|RLi|TMi|Mi|RTi|RTi'|ti|msgi|1]iS

 ||...|| Sj[Aj|RLj|TMj|Mj|RTj|RTj'|tj|msgj|1]jS ||...,com > .
```

Communication among agents is achieved using a `Communication` rule. When

agents communicate with each other, one agent copies the communicated fact from another agent's memory. Copying is only allowed if the fact to be copied is not already in the working memory of the agent intending to copy. The `Communication` rule is given below.

```
rl [Communication] :  <...|| Si[Ai|RLi|TMi|Mi|RTi|RTi'|ti|msgi|3]iS

||...|| Sj[Aj|RLj|TMj|Mj|RTj|RTj'|tj|msgj|3]jS ||...,com >
                                =>
    copy(<...|| Si[Ai|RLi|TMi|Mi|RTi|RTi'|ti|msgi|3]iS

||...|| Sj[Aj|RLj|TMj|Mj|RTj|RTj'|tj|msgj|3]jS ||...,com >) .
```

Whenever the above `Communication` rule executes, the right hand side `copy` operator (which is declared in the MAS module) reduces the `masConfig` using a set of equations defined below. If the equational condition is satisfied then it will pull the communicated fact from one agent's memory to the other. In each equation the `copy` operator calls itself recursively so that all the agents in the system can copy among each other all the communicated facts possible at this cycle.

```
ceq [Copy1] :  copy(<...|| Si[Ai|RLi|TMi|Ask(i,j,P) Mi|RTi|RTi'|ti|msgi|3]iS

||...|| Sj[Aj|RLj|TMj|Mj|RTj|RTj'|tj|msgj|3]jS ||...,com >)
                                =
copy(<...|| Si[Ai|RLi|TMi|Ask(i,j,P) Mi|RTi|RTi'|ti|msgi+1|3]iS ||...||

Sj[Aj|RLj|[tj : Ask(i,j,P)] TMj|Ask(i,j,P) Mj|RTj|RTj'|tj|msgj+1|3]jS ||...,com >)

                if (not inWM(Ask(i,j,P), Mj)) .


ceq [Copy2] :  copy(<...|| Si[Ai|RLi|TMi|Mi|RTi|RTi'|ti|msgi|3]iS

||...|| Sj[Aj|RLj|TMj|Tell(j,i,P) Mj|RTj|RTj'|tj|msgj|3]jS ||...,com >)
                                =
copy(<...|| Si[Ai|RLi|[ti : Tell(j,i,P)] TMi|Tell(j,i,P) Mi|RTi|RTi'|ti|msgi+1|syni]iS

||...|| Sj[Aj|RLj|TMj|Tell(j,i,P) Mj|RTj|RTj'|tj|msgj+1|3]jS ||...,com >)

                if (not inWM(Tell(j,i,P), Mi)) .
```

```
                                    ⋮

  eq [Default] :  copy(<...|| Si[Ai|RLi|TMi|Mi|RTi|RTi'|ti|msgi|3]iS

      ||...|| Sj[Aj|RLj|TMj|Mj|RTj|RTj'|tj|msgj|3]jS ||...,com >)
                                    =
            <...|| Si[Ai|RLi|TMi|Mi|RTi|RTi'|ti|msgi|3]iS
    ||...|| Sj[Aj|RLj|TMj|Mj|RTj|RTj'|tj|msgj|3]jS ||...,exec > [owise] .
```

Note that, during `copy` operations, agents only update their respective message counters but not their cycle times. When all the agents in the system have copied all the communicated facts at this cycle, the system changes its status from `Communication` to `Apply` mode by changing the flag `com` to `exec`. This is achieved using Maude's otherwise (`[owise]`) attribute. Therefore, the execution of the `Communication` rule is followed by the `Apply` rule and the system moves forward.

Note that when `Ask(i,j,P)` is added to agent j's working memory, j may perform some computation if it does not know whether P is the case. In this model, communication requires a single time step, i.e., when agent i asks agent j whether P is the case at time step t, agent j will receive the request at time cycle $t + 1$. The time agent i has to wait for a response to its query depends on the reasoning j must (or chooses) to do (if j is concrete), or j's specification (if j is abstract). A similar approach is used when j tells i that P. Recall that when `Generate`, `Choice`, and `Communication` rules execute agents do not change their time cycles, however time cycles are increased by one when the `Apply`, and `Idle` rules execute. All three phases match (`Generate`), select (`Choice`) and execute (consisting of `Communication`, `Apply`, and `Idle`) therefore happen in a single time step.

7.2.4 Verifying system properties

Model checking in Maude involves a Maude specification of a system together with a property of interest. A property is a LTL formula interpreted as a property of computations of the system (linear sequences of states generated by application of rewrite rules). A simple path from a given initial state s, to a state satisfying a property φ is

a list of rules together with a state s' satisfying φ such that applying the rules starting with s leads to s'. One way to find a simple path is to model check the assertion that from s no state can be reached satisfying φ: modelCheck(s, $\sim$ F φ). If there is a reachable state satisfying φ, a counterexample will be returned. The counterexample contains the list of rules applied. Given a system module, say `MAS`, and an initial state, say s of sort `masConfig`, we can model check different LTL properties beginning at this initial state by doing the following:

- defining a new module, `ModelCheck-MAS`, that includes the module `MAS` and Maude's built-in module `MODEL-CHECKER` module as submodules;
- giving a subsort declaration, `masConfig < State`, where `State` is a sort in the module `MODEL-CHECKER`;
- defining the syntax of the (target) state predicates we wish to use by means of constants and operators of sort `Prop`, a subsort of the sort `Formula` (i.e., LTL formulae) in the module `MODEL-CHECKER`;
- defining the semantics of the state predicates by means of equations.

The following `ModelCheck-MAS` system module shows how we can define state predicates whose semantics are defined by appropriate equations.

```
mod ModelCheck-MAS is
 including MAS .
 including MODEL-CHECKER .
 subsort masConfig < State .
 op success : -> Prop .
 var C : Config .
 var phase : Phase .
 eq < Si[Ai:Agenda|RLi:Agenda|TMi:TimeWM|P Mi:WM|RTi:Rep
 TWM|RTi':RepTWM|t:Nat|msgi:Nat|syni:Nat]iS || C:Config,
  phase:Phase > |= success = true .
 eq C |= success = false [owise] .
 op init : -> masConfig .
 eq init = < S1[...|...|...|...|...|...|0|0|1]1S ||...||
             Si[...|...|...|...|...|...|0|0|1]iS ||...||
             Sn[...|...|...|...|...|...|0|0|1]nS,com > .

endm
```

Listing 7.7: Structure of ModelCheck-MAS module

In the state predicate semantics defined in Listing 7.7, the `masConfig` says that agent i's working memory contains pattern `P`. The remaining information of the configuration is specified using Maude's on-the-fly variable declaration. Note however that, the initial state must contain information using ground terms only. In the `ModelCheck-MAS` module the initial system state is represented using `init`, where all the `...` entries of the configuration represent ground terms. Once the semantics of each of the state predicates has been defined, given an initial state `init`, we can model check any LTL formula, say φ, involving such predicates. We do so by executing in Maude, the command `reduce modelCheck(init, φ)`, where φ could be, for example, `[] success`, `<> success`, `<> ~success` etc. Two things can then happen: if the property holds, then we get the result true; if it does not, we get a counterexample.

7.2.5 Analysis of the implementation

When implementing reasoning strategies which involve time stamps of patterns, it is convenient to be able to associate a time stamp to each pattern. To achieve this, we have declared the sort `TimeWM` in the above encoding. However, in the encoding we maintained both the sorts `TimeWM` and `WM` simultaneously. In this section, we explain why. Let us suppose that each agent uses `TimeWM` as its only working memory. When agents generate their conflict sets, they check whether consequents of rule instances are already present in their working memory. If so, then these rule instances will not be added to their agendas. Similarly, when agent fires a rule instance or receives a message from another agent, it will make sure these patterns are not present in its working memory. For example, suppose a pattern `P` with time stamp `t1` is already added to the working memory of an agent i. That is `[ t1 : P ]` is already present in `TimeWM`. Sometimes later, say at time `t2 (> t1)`, agent i needs to check whether `[ t2 : P ]` is already present in its working memory. It is apparent that the elements `[ t1 : P ]` and `[ t2 : P ]` of `TimeWM` are distinct because `t1` $\neq$ `t2`. However, the pattern `P` is common to both of them. Therefore, to ensure that

working memory does not contain duplicate patterns it is necessary to ensure that the second part `P` of `[ t2 : P ]` is not already present in the working memory. This can be accomplished in one of two ways. One way is to compare the second part of `[ t2 : P ]` with the second part of each element `[ tk : P ]` of `TimeWM`. In order to implement this approach some Maude conditional equations are required. However, the execution of additional conditional equations slows down the computation. Another way is to maintain a duplicate working memory `WM` which contains all the patterns of the form `P`. Whenever an element `[ t : P ]` is added to `TimeWM`, the corresponding pattern `P` will be added to `WM`. In other words `TimeWM` and `WM` is updated simultaneously. Thus it is only necessary to check whether the second part `P` of `[ t2 : P ]` is already present in `WM` or not. Therefore, although maintaining only one working memory is enough, we use duplicate working memory for efficiency purposes.

7.3 The TVRBA verification tool

In this section, we describe the tool TVRBA, which generates an encoding of a system of communicating rule-based agents for the Maude LTL model checker. The architecture of TVRBA has three layers, as shown in Figure 7.2. The purpose of dividing this into three separate layers is for ease of implementation and maintenance.

1. **User interface:** the first layer is the user interface layer by which the system designer can interact with the tool and vice-versa. This layer is responsible for validating input data from designer and presenting the output of model checker to the designer. The tool takes as input:

   (i) a set of concrete agent descriptions, each comprising a set of rules, a set of initial working memory facts, and a reasoning strategy;

   (ii) a set of abstract agent descriptions specified by a set of temporal doxastic logic formulae;

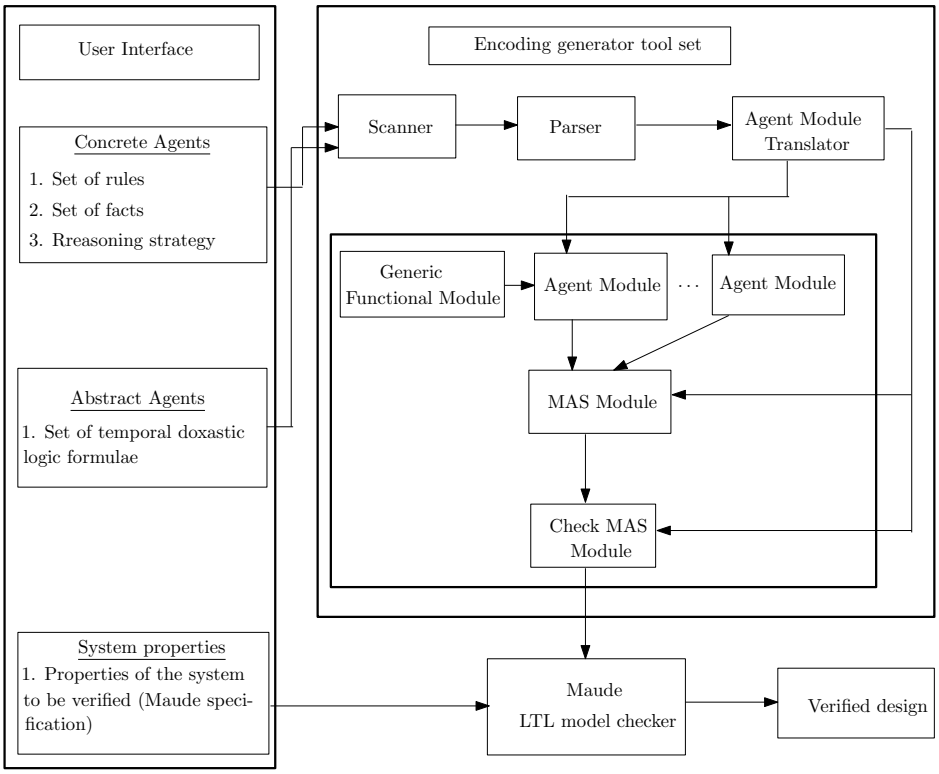


Figure 7.2: Architecture of TVRBA

(iii) the properties of the system to be verified specified in temporal doxastic logic.

Rules and facts can be expressed in XML or in a simplified ASCII syntax e.g., $< \ n : P_1 \ \wedge \ldots \wedge \ P_n \rightarrow P \ >, \ P_k$. Note that one of our aims is to use some standard rule representation language, such as HornLog RuleML. However, rule priorities, which are required by some of the supported inference (conflict resolution) strategies are not supported by standard HornLog. Therefore, the TVRBA tool will automatically translate the input syntax $< \ n : P_1 \wedge \ldots \wedge \ P_n \rightarrow P \ >$ into the corresponding time pattern rule language syntax:

$$< \ n : \ [\ t_1 \ : \ P_1 \ ] \ \wedge \ldots \wedge \ [\ t_n \ : \ P_n \ ] \rightarrow [\ t \ : \ P \ ] \ >$$

The general XML syntax of rules accepted by TVRBA corresponds to HornLog RuleML with negation as failure. An XML document consists of possibly nested elements, where each element is a sequence of the form `<tag>`

E1...Em </tag>. In RuleML, a variable is denoted by a var element of the form <var>variableName</var>, a constant is denoted by a ind element of the form <ind>constantName </ind>, a relation (predicate) is denoted by a rel element of the form <rel>relationName</rel>. The application of rel to a sequence of terms is denoted by an atom element of the form given below.

```
<atom>
 <rel>relationName</rel>
 <var>variableName</var>
 <ind>constantName</ind>
</atom>
```

A HornLog rule is asserted as an imp element that has two parts: a head consists of a single atom and a body consists of one or more atom elements. Negation as failure is represented using a naf element of the form <naf> <atom>...</atom></naf>. A BNF of HornLog rules is given in Figure 7.3.

$$
\begin{array}{lcl}
Rule & ::= & imp(head, body) \\
head & ::= & atom \\
body & ::= & (atom \mid naf)^{*} \\
atom & ::= & rel((var \mid ind)^{+}) \\
naf & ::= & naf(atom)
\end{array}
$$

Figure 7.3: HornLog rule syntax

Rules expressed in XML are translated internally into the simplified ASCII syntax. Once translated, they can be annotated by the user with rule priorities, and these annotated rules are then used to produce Maude specification. TVRBA supports the full range of conflict resolution strategies given in § 6.3.2. Different agents in the system may use different strategies. The LTL specification of the behaviour of abstract agents and properties to be verified are given in a simplified ASCII notation.

2. **Encoding generator:** the second layer is the encoding generator layer which consists of three main components: a scanner, a parser, and a translator. The

scanner or lexical analyser is used to validate the syntax of the input data provided by the system designer. If the input presented by the designer is lexically valid then the scanner will pass it to the parser for further processing, otherwise, it will report an error in case of an invalid input information. The parser retrieves tokens from the scanner and processes these tokens to construct the model. The translator is the main component of the tool. It takes inputs from parser as abstract syntax and translate them into the corresponding Maude specifications.

3. **System verifier:** the third layer is the system verifier layer which feeds the output of the Encoding generator to Maude LTL model checker along with the properties of the system to be verified. The output of the tool is the verified design result, i.e., whether the MAS satisfies the desired property.

7.3.1 TVRBA implementation

We chose to build the tool using the Java platform. Java is a widely used object-oriented programming language and has the advantage of being platform-independent. The user interface(UI) of the tool is implemented in Java Swing.

This section describes how the encoding generator component of the tool translates the designer's input into a Maude system specification.

When the system designer loads a set of rules and facts in valid Hornlog RuleML syntax, the tool uses XSLT transformation to transform the XML syntax into the ASCII syntax. These translated ASCII rules are then annotated by the user with rule priorities and are used as input for Maude specification.

The scanner and parser are implemented as Java classes. The scanner, represented by Scanner object class, is used to validate input data from users. If a symbol is valid, the scanner will send the corresponding token to the parser. Otherwise, it will throw a ScannerException to report details of the failure. The scanner in our implementation is based on the grammar shown in Figure 7.4. Terminal symbols in the grammar are as

usual, and some special symbols are used , e.g., $ stands for white space, EOL stands for end of line($\backslash n$) and, the symbol # denotes as end of text (EOT).

$$
\begin{array}{lll}
Program & ::= & (Token \mid Separator)^{*}\# \\
Token & ::= & Identifier \mid Connector \mid (\mid) \mid <\mid>\mid, \mid' \\
Identifier & ::= & Letter(Letter \mid Digit)^{*} \\
Letter & ::= & A \mid B \mid \ldots \mid Z \mid a \mid b \mid \ldots \mid z \\
Number & ::= & (Digit)^{+} \\
Digit & ::= & 0 \mid 1 \mid \ldots \mid 9 \\
Connector & ::= & \& \mid \rightarrow \\
Separator & ::= & \$ \mid EOL
\end{array}
$$

Figure 7.4: Lexical syntax

A Parser object wraps a Scanner object, and feeds the input into it. It also receives the output of the scanner in the form of Tokens. The implementation of the parser is based on the grammar, depicted in Figure 7.5. The parser is a $LL(1)$ predictive parser, because it only needs to look ahead one token to decide the next parsing process.

$$
\begin{array}{lll}
Belief & ::= & Fact \\
Rule & ::= & Literals \rightarrow Literal \\
Literals & ::= & Literal(\&\ Literal)^{*} \\
Literal & ::= & Predicate(Terms) \\
Terms & ::= & Term(,\ Term)^{*} \\
Term & ::= & Constant \mid Variable \mid Function \\
Function & ::= & Identifier(Terms) \\
Constant & ::= & QuotedIdentifier \\
Variable & ::= & Identifier \\
Predicate & ::= & Identifier \\
QuotedIdentifier & ::= & 'Identifier \\
Fact & ::= & Literal \text{ where all terms are } Constant
\end{array}
$$

Figure 7.5: Abstract syntax for rules

The rules and facts in ASCII syntax are represented using abstract syntax trees generated by the parser. These abstract syntax trees can then easily be translated (decorated) to give the desired format.

Abstract agents are specified by a set of temporal doxastic logic formulae using the syntax given in §6.4. For example, when the temporal logic formula with belief operators $G(B\ 2\ \mathsf{Ask}(1, 2, P) \rightarrow X\ 5\ B\ 2\ \mathsf{Tell}(2, 1, P))$ is specified as input, its syntax

must be validated. Input validation is carried out by the scanner and parser. The implementation of the parser is based on the grammar, depicted in Figure 7.6. The temporal doxastic formulae in ASCII syntax are represented using abstract syntax trees generated by the parser. These abstract syntax trees can then be used to translate the temporal specification into the desired Maude system specification. Some Java object classes such as ResponseFormula and SimpleAskTellFormula are used to merge the abstract agents specification with that for the concrete agents.

$$
\begin{array}{lll}
LTLFormula & ::= & Next\ num\ Phi \mid Globally(PhiOne \rightarrow\ Next\ num\ PhiTwo) \\
Next & ::= & X \\
Globally & ::= & G \\
Phi & ::= & Belief\ Agi\ Ask(Agi, Agj, Predicate) \\
 & & \mid Belief\ Agi\ Ask(Agj, Agi, Predicate) \\
 & & \mid Belief\ Agi\ Tell(Agi, Agj, Predicate) \\
 & & \mid Belief\ Agi\ Tell(Agj, Agi, Predicate) \\
 & & \mid Belief\ Agi\ Predicate \\
PhiOne & ::= & Belief\ Agj\ Ask(Agi, Agj, Predicate) \\
PhiTwo & ::= & Belief\ Agi\ Tell(Agj, Agi, Predicate) \\
num & ::= & Number \\
Agi & ::= & Number \\
Agj & ::= & Number \\
Belief & ::= & B
\end{array}
$$

Figure 7.6: Abstract syntax for LTL fornulae

Once a system encoding has been generated the designer can specify properties of interest, by providing an initial and a target state of the system. An LtlPropertySpecification object class generates the ModelCheck-MAS module in order to model check properties of the system. The same procedure as discussed above is used to parse and validate user input. The Maude LTL model checker has been integrated with the tool using MaudeWrapper object class which is used to check the specified properties of the system.

Now we explain how Maude implements the properties that are to be checked specified in temporal doxastic logic. Note that an agent i believes a formula φ if φ is in the agent i's working memory. In Listing 7.7, we have shown, for example, how to define a proposition `success` in which the `masConfig` says that agent i's working

memory contains pattern `P`. Therefore, some interesting properties that can be verified include, for example, $X^n B_i P$.[1] In Maude $X^n B_i P$ can be specified as:

```
reduce modelCheck(init, O O...O success).
```

where `O O...O` stands for n application of `O`. In Maude specification `O` stands for X. A screenshot of TVRBA's user interface is presented in Figure 7.7.

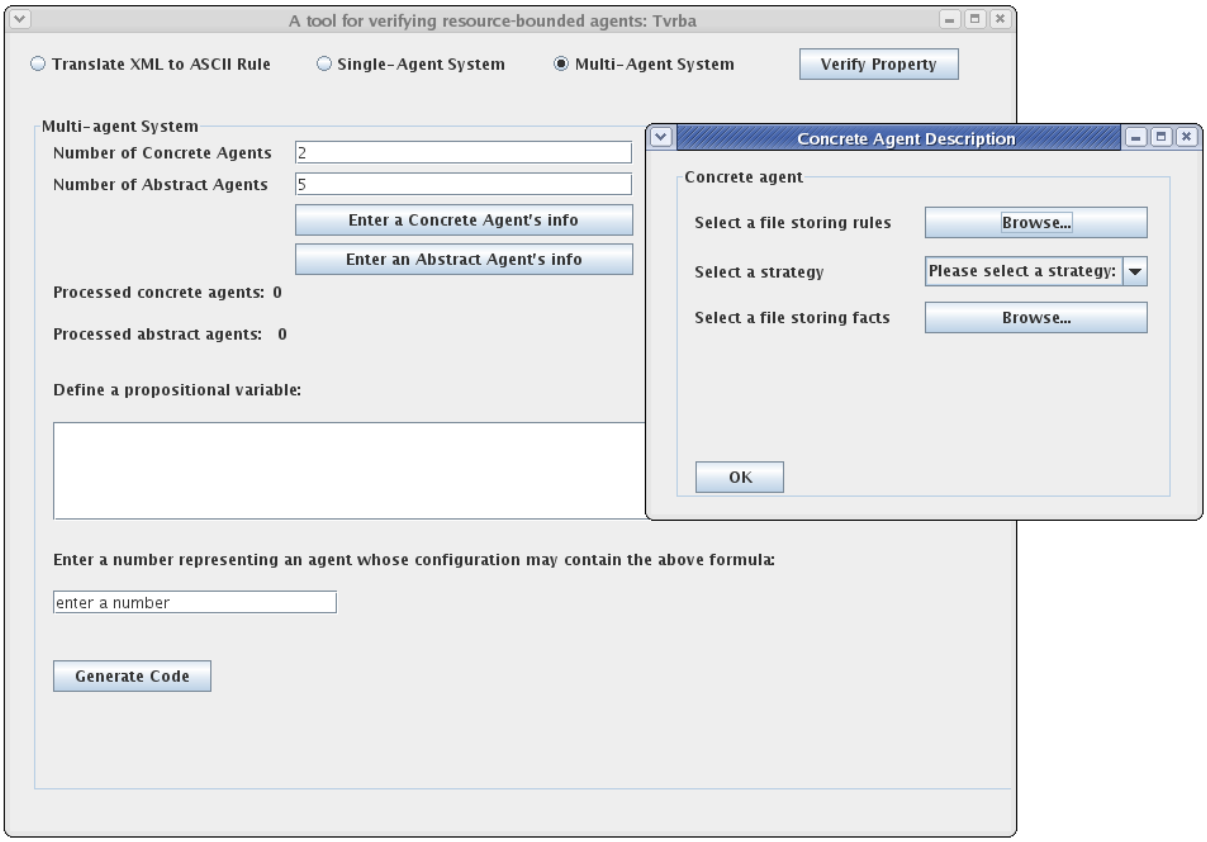


Figure 7.7: Screenshot of TVRBA's graphical user interface

[1] Recall that we can use $X^n B_i P$ to verify whether $X^{\leq n} B_i P$ holds.

Chapter 8

Scalable MAS verification: case studies

In this chapter, we report experiments designed to illustrate the scalability and expressiveness of the approach described in Chapter 6 and the TVRBA tool described in Chapter 7. The first two experimental results described in this chapter were originally presented in [Alechina et al., 2010]. All the experiments reported here were performed on an Intel Pentium 4 CPU 3.20GHz machine with 2GB of RAM under CentOS release 4.8.

8.1 Binary tree example

To illustrate the scalability of our approach we re-implemented an example scenario introduced in Chapter 5. In this scenario, a system of communicating reasoners attempt to solve a distributed reasoning problem where the set of rules and facts that describes the agents' knowledge base are constructed from a complete binary tree. For example, a complete binary tree with 8 leaf facts has the following set of rules

$$\textbf{RuleB1}\ A_1(x) \wedge A_2(x) \rightarrow B_1(x) \quad \textbf{RuleB2}\ A_3(x) \wedge A_4(x) \rightarrow B_2(x)$$

$$\textbf{RuleB3}\ A_5(x) \wedge A_6(x) \rightarrow B_3(x) \quad \textbf{RuleB4}\ A_7(x) \wedge A_8(x) \rightarrow B_4(x)$$

$$\textbf{RuleC1}\ B_1(x) \wedge B_2(x) \rightarrow C_1(x) \quad \textbf{RuleC2}\ B_3(x) \wedge B_4(x) \rightarrow C_2(x)$$

$$\textbf{RuleD1}\ C_1(x) \wedge C_2(x) \rightarrow D_1(x)$$

# leaves	# steps	CPU Time (in seconds)
128	127	1
512	511	97
1024	1023	903
2048	2047	13252

Table 8.1: Resource requirements for a single agent

For compatibility with the propositional example considered in [Alechina et al., 2008b], (see § 5.7.3) we assume that the variable x is substituted by a single constant value 'a', and the goal is to derive $D_1(a)$. One can easily see that a larger system can be generated using 16 'leaf' facts $A_1(x), \ldots, A_{16}(x)$, adding extra rules to derive $B_5(x)$ from $A_9(x)$ and $A_{10}(x)$, etc., and a new goal $E_1(x)$ derivable from $D_1(x)$ and $D_2(x)$ to give a '16 leaf example'. Similarly, we can consider systems with 32, 64, 128, ..., 2048 etc. leaf facts.

In [Alechina et al., 2008b], the results of experiments on such problems using the Mocha model-checker are reported. In the simplest case of a single agent, the largest problem that could be verified using Mocha had 128 leaf facts (cf. § 5.7.3). However, using our tool we are able to verify a system with 2048 leaf facts. This was modelled as a single concrete agent, with varying numbers of facts and rules. The experimental results are summarised in Table 8.1.

In the case of multi-agent systems, the exchange of information between agents was modelled as an abstract **Copy** operation (cf. § 5.7.3). Each **Copy** operation takes one tick of system time and does not require any special communication rules. We were able to verify a multi-agent system consisting of two agents with 16 leaf facts. An invariant property of the form $AG\neg(B_1\ \varphi \vee B_2\ \varphi)$ (where φ represents the the root node) was verified when the odd position node facts were assigned to one agent and the even position node facts were assigned to the other agent in the system. In our re-implementation, communication between agents is achieved using Ask and Tell actions. The results presented in § 5.7.3 and those for our tool are therefore not directly comparable in the multi-agent case. Nevertheless, we can show that much larger multi-

agent systems can be modelled using our approach.

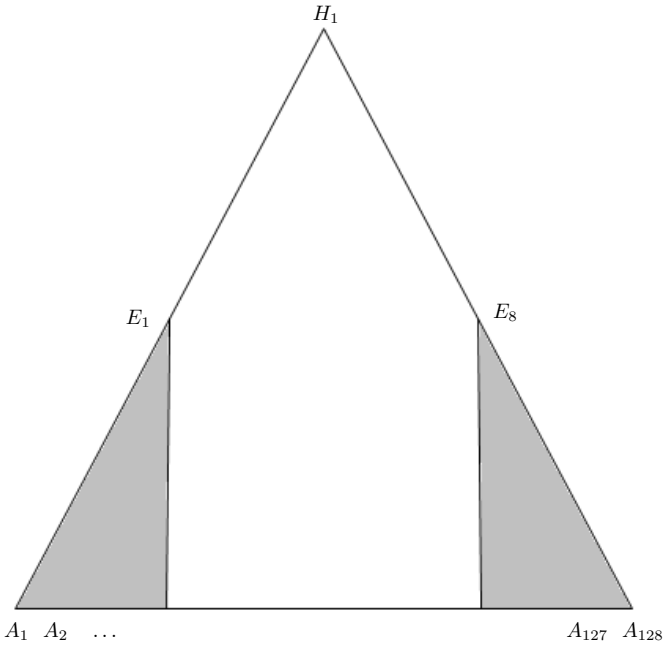


Figure 8.1: Binary tree example with triangular regions

Consider a multi-agent system consisting of two concrete agents each with a knowledge base of facts and rules for the 128 leaf example (i.e., both agents have all the rules and leaf facts). Both agents in the system use rule ordering reasoning strategy. Agent 1 assigns lower priority to rules in the right-hand shaded triangular region depicted in Figure 8.1. In contrast, agent 2 assigns lower priority to rules in the left-hand shaded triangular region of Figure 8.1. In Appendix G, using a smaller example (cf. Table G1), we show how we can direct the agents to focus on a particular region of the tree by assigning rule priority. Suppose agent 1 asks agent 2 if $E_8(a)$ is the case. If agent 1 receives the fact $E_8(a)$ from agent 2 before deriving $E_8(a)$ itself, it can avoid firing 15 rules, and the agents are able to derive the goal $H_1(a)$ in 115 steps while exchanging two messages.

Similarly, consider the scenario in which there are three concrete agents, each with a knowledge base of facts and rules for the 128 leaf example. All the agents in the system use the rule ordering reasoning strategy. Assume agent 1 asks agent 2 if $E_1(a)$ is the case and also that agent 1 asks agent 3 if $E_8(a)$ is the case. In this case the set

# agents	# leaves	# steps	#msgs	CPU Time (in seconds)
2	128	115	2	7
3	128	103	4	18

Table 8.2: Resource requirements for multiple agents

of rules in the unshaded region have higher priority for agent 1, the rules in left hand shaded region have higher priority for agent 2, and the rules in the right hand shaded region have higher priority for agent 3. Then the agents can derive the goal $H_1(a)$ in 103 steps while exchanging four messages. The experimental results are summarised in Table 8.2. Although these examples are very simple, they point to the possibility of complex trade-offs between time and communication bounds in systems of reasoning agents. In Appendix G, using a smaller example (cf. Table G2), we show how we can direct the agents to focus on a particular region of the tree by assigning rule priority in the three agent case.

8.2 A route planning example

To illustrate the application of the framework on a more complex example we consider the following scenario. The system consists of several agents representing users who have queries about possible subway routes on the London Underground, denoted by u_i, and two agents that provide travel advice: a 'route planning' agent, p, that computes routes between stations and an 'engineering work' agent, e, which has information about line closures and other service disruptions. The user agents ask the route planning agent for route information, that is, they generate queries of the form:

$$\mathsf{Ask}(u_i, p, Route(start\_station, destination\_station)).$$

The route planning agent has a set of facts corresponding to connections between stations, and a set of rules for finding a path between stations which returns a route (a list of intermediate stations). Upon receiving a request from the user agent, the route planning agent tries to find a route from the $start\_station$ to the $destination\_station$ by firing a sequence of rules based on the facts in its working memory. To ensure

a route is valid, the planner must check that it is not affected by service disruptions caused by engineering work, which it does by querying the engineering work agent. If the route is open, the planner returns the route from $source\_station$ to the $destination\_station$ to the user agent.

The user agents are modelled as abstract agents that generate a query at a non-deterministically chosen time step within a specified interval, e.g.:

$$X^{\leq 5} B_{u_i} \mathsf{Ask}(u_i, p, Route(MarbleArch, Victoria))$$

The engineering work agent is also modelled as an abstract agent which is assumed to respond to a query within some bounded number of time steps, e.g., n time steps:

$$\begin{gathered} G(B_e \mathsf{Ask}(p, e, RouteList(start\_station, destination\_station, \\ [\ station_1 \mid station_2 \mid \ldots \mid station_n\ ])) \rightarrow \\ X^{\leq n}\ B_e \mathsf{Tell}(e, p, RouteList(start\_station, destination\_station, \\ [\ station_1 \mid station_2 \mid \ldots \mid station_n\ ])) \end{gathered}$$

where $[\ station_1 \mid station_2 \mid \ldots \mid station_n\ ]$ is a list of intermediate stations from the $start\_station$ to the $destination\_station$, and the response from the engineering agent indicates that the route from the $start\_station$ to the $destination\_station$ via $station_1,\ station_2\ , \ldots, station_n$ is open.

The system designer may wish to verify that the proposed design of the route planning agent, together with the assumed or known properties of the engineering work agent, is able to respond to a given number of user queries arriving within a specified interval, within a specified period of time. For a typical routing query, e.g., for an abstract user agent u_i asking for a route between station1 and station2, we can verify that response is received within n time steps:

$$\begin{gathered} G(B_{u_i} \mathsf{Ask}(u_i, p, Route(s1, s2)) \rightarrow \\ X^{\leq n}\ B_{u_i} \mathsf{Tell}(p, u_i, RouteList(s1, s2, [t_1 | t_2 | \ldots | t_n]))) \end{gathered}$$

# user agents	# time steps	CPU Time (in seconds)
2	21	39
4	29	236
5	33	530

Table 8.3: Resource requirements for the route planning example

Table 8.3 reports experimental results for a multi-agent system consisting of a planner agent, an engineering agent and varying number of user agents. In this experiment, we have used 6 stations connected by 3 different lines (a total of 7 facts) and the planner can derive 8 different routes. Different user agents in the system make queries about different routes at different times in the interval $[1, 10]$. For example, the user agent u_i may request a route between $Marble\ Arch$ and $Victoria$:

$$\mathsf{Ask}(u_i, p, Route(MarbleArch, Victoria))$$

and receive the reply

$$\mathsf{Tell}(p, u_i, RouteList(MarbleArch, Victoria, [BondStreet|GreenPark]))$$

The time steps value in Table 8.3 gives the maximum number of time steps necessary to return a route to a user agent under the specified system load.

8.3 A home health-care monitoring alarm system

Finally, we consider a home health care monitoring alarm system adapted from [Paganelli and Giuli, 2007]. The system consists of several concrete and abstract agents. The concrete agents in the system include a number of home healthPCs, pc_i, and a central Health Planner, p. Each pc_i agent in the system is connected with two body sensor agents: a Blood pressure monitoring agent, b_i, and a Heart rate monitoring agent, h_i. The agents b_is and h_is are modelled as abstract agents. All the home healthPC agents pc_i can communicate with the agent p, which is located at the health centre. The agent p can also communicate with various other agents in the system including doctors, nurses, relatives of patients, and an emergency operator. The over-all picture of the

system is depicted in Figure 8.2.

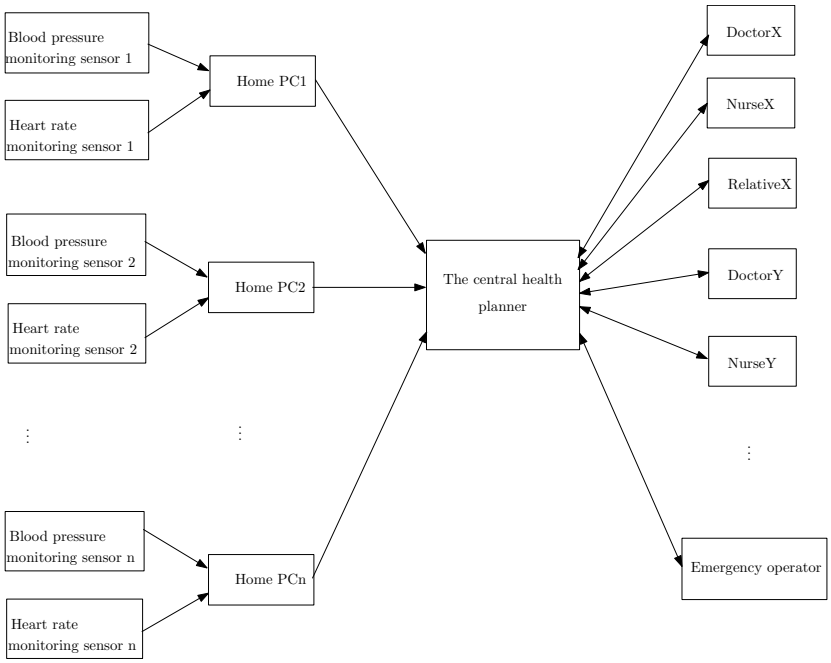


Figure 8.2: Health-care monitoring system

The abstract agents b_i and h_i measure the Blood pressure and Heart rate information of a patient and inform to the corresponding home healthPC, pc_i, as messages of the form:

$$\mathsf{Tell}(b_i, pc_i, BloodPressure(patientID, bp))$$

$$\mathsf{Tell}(h_k, pc_i, HeartRate(patientID, hr))$$

Upon receiving the Blood pressure and Heart rate information from the body sensor agents, the agent pc_i derives an alarm level by firing a sequence of rules from its knowledge base. The alarm level may be Low, $Very\ Low$, $Medium$ and $High$ depending on the blood-pressure and heart-rate measurement values. The agent pc_i then sends the alarm level information to the agent p for the patient's health planning. In this system, the doctors, d_i, nurses, n_i, relatives of patients, r_i, and an emergency operator, e are modelled as abstract agents. These abstract agents can notify the agent p about their availability by sending messages, e.g., "available", "busy", and "notAvailable". The messages generated by these abstract agents are of the form:

$$\mathsf{Tell}(d_i, p, DoctorX(status))$$

$$\mathsf{Tell}(n_i, p, NurseX(status))$$

$$\mathsf{Tell}(r_i, p, RelativeX(status))$$

The agent p implements alarm notification policies specifying whom should be alerted, how and when the notification is to be sent and if acknowledge is required. The alarm notification policies are given below.

Alarm Level	Notification Policies
Very Low	message to relative, no ack
Low	message to doctor, no ack and message to relative, no ack
Medium	message to doctor or nurse, ack and message to relative, no ack
High	message to emergency operator, ack and message to relative, ack

The agent p alerts a contact person (doctor, nurse, or relative of a patient) based on their availability status and for certain cases the agent p may require an acknowledgement. The availability status of a doctor, nurse, or relative of a patient may change from "available" to "busy" or "notAvailable" when they are contacted by the agent p. In this case, the agent p waits for a fixed time interval and then based on the acknowledgement received it might contact other agents for a service. For instance, when a $Medium$ level alarm occurs, the agent p first alerts a doctor, d_i. If the received acknowledgement from the agent d_i within a fixed time interval is "busy" or "notAvailable", then the agent p_i alerts a nurse, n_i, if she also sends an "busy" or "notAvailable" message within a fixed time interval, then the agent p alerts an emergency operator. At the same time, the agent p alerts relative of patient, but acknowledgement is not required in this case.

The Blood pressure and Heart rate sensor agents in the system generate information about the measurement values at different times in the interval $[1, 5]$. For example, the agent b_i generates blood pressure information for a patient with patientID $P001$ and blood pressure $B4$ (where $B4$ symbolically representing the fact that systolic blood

pressure is higher than 160mm/Hg) using the following formula:

$$X^{\leq 5}\ B_{b_i}\ \mathsf{Tell}(b_i, hpc_j, BloodPressure(P001, B4))$$

In this experiment, the priorities (from higher to the lower) of the rules of the central Health Planner are assigned corresponding to the alarm levels $High$, $Medium$, Low, and $Very\ Low$, respectively. The experimental results reported in Table 8.4, for the 1 patient scenario, the system generates $Medium$ alarms, for the 2 patients scenario, the system generates $Medium$ alarms for one patient and $High$ alarm for the other patient. For ease of illustration, we modelled one doctor, one nurse, and one relative corresponding to each patient in the system. We verified the following property of the system: whenever patient's alarm level is $Medium$ and the agent p has received acknowledgements from the doctor and nurse as busy the agent p contacts the emergency operator within n timesteps.

$$G(\ AlarmLevel(P001, Medium) \wedge \mathsf{Tell}(d_i, p, DoctorXAck(P001, busy)) \wedge$$
$$\mathsf{Tell}(n_i, p, NurseXAck(P001, busy)) \rightarrow X^{\leq n}\ \mathsf{Tell}(p, e, AlermLevel(P001, Medium))\ )$$

The above property is verified as true when the value of n is 3 and the model checker spends 72 seconds for the 1 patient scenario and 165 seconds for the 2 patient scenario. However when we assign a value less than 3 to n the property is verified as false and the model checker returns a counterexample. This also ensures the correctness of the encoding. In Table 8.4, we have shown the required time steps (from the system startup) and the number of messages that are exchanged between the agents when the property is verified as false.

#Patients	#Concrete agents	# Abstract agents	# timesteps	#msgs	CPU Time (in seconds)
1	2	6	19	22	0.04
2	3	11	26	32	0.09

Table 8.4: Resource requirements for the health planner

8.4 Discussion

In this chapter, we illustrated the scalability of our new approach by comparing it to results presented in [Alechina et al., 2008b] and in Chapter 5 for a synthetic distributed reasoning problem. In our previous work, the largest problem that could be verified using Mocha had 128 leaf facts, however, using our new approach we are able to verify a system with 2048 leaf facts. This was modelled as a single concrete agent which uses a rule ordering reasoning strategy. In our new approach, communication between agents is achieved using Ask and Tell actions which is different from the **Copy** operation used in our previous work, therefore the performance of verification is not directly comparable in the multi-agent case. However, we showed that much larger multi-agent systems can be modelled using our new approach. We also showed how to further improve scalability by using abstract agents specified in terms of temporal doxastic formulae. The modelling and experimental results using the notion of abstract agents show the usability of TVRBA for various other problems.

Chapter 9

Conclusions and future work

This thesis has centred on the development of techniques and tools for verifying resource requirements for systems of reasoning agents, such as, for example, agents which reason using resolution or rules. In this chapter, we briefly summarise the main contributions of the thesis, and suggest some possible future lines of research.

9.1 Summary of contributions

In this thesis we have argued that there is a need for frameworks for modelling and verifying properties of resource-bounded multi-agent systems. When solving problems, each intelligent agent in a multi-agent system requires some basic resources such as *time* (number of computational steps), *space* (amount of memory) and perhaps *communication bandwidth* (number of messages that need to be exchanged). We briefly reviewed the existing logical formalisms for reasoning about resource-bounded agents and discussed their expressiveness. We then argued that there is a need to define temporal doxastic logics which allow us to express properties of systems to investigate trade-offs between multiple resource bounds (memory, time and communication bandwidth). The logical formalisms described in § 4.4 were developed by Nga and Alechina [Alechina et al., 2009a], to meet this need, and form the starting point for some of the work reported in this thesis a brief description of these logics has been

provided whenever necessary to understand the frameworks presented in this thesis. We reviewed the state-of-the-art in verifying multi-agent systems. The two most popular approaches to formal verification we surveyed are theorem proving and model checking, and we briefly summarised how these techniques have been used to verify properties of multi-agent systems. We then analysed the limitations of the current approaches.

We then proposed some frameworks for modelling and verifying resource-bounded reasoning agents. First, we presented a framework for verifying systems composed of resolution-based reasoning agents, where the resources each agent is prepared to commit to a goal (time, memory and communication bandwidth) are bounded. We then presented a second framework for verifying systems of distributed rule-based reasoning agents. The work presented in Chapter 4 extends the work presented in [Albore et al., 2006] which proposed a method of verifying memory and time bounds in a single reasoner that reasons in classical logic using natural deduction rather than resolution. The expressiveness and the scalability of our approaches (presented in Chapters 4 & 5) are illustrated through the verification of two typical multi-agent reasoning problems which can be easily parameterised to increase or decrease the problem size. Although our approaches scale better than those presented in [Albore et al., 2006], the results were still not satisfactory. In order to improve scalability of model checking for larger problems, we analysed the problem and its encoding complexity to better understand the scalability issues. Both the complexity analysis and experimental results suggested that reasonably sized problem instances are unlikely to be tractable for a standard model checker without steps to reduce the branching factor of the state space.

To address the scalability issues identified in Chapters 4 & 5, we proposed a new approach to model checking MAS which uses strategies and abstraction. Our modelling approach abstracts from some aspects of system behaviour to obtain a system model that is tractable for a standard model checker. When verifying response time guarantees of the 'focal' agent(s), the concrete representation of 'peripheral' agents can

be replaced by an abstract specification of their external (communication) behaviour, so long as the abstract specification results in behaviour that is indistinguishable from the original concrete representation for the purposes of verification, i.e., it produces queries and responds to queries within specified bounds. All other details of an abstract agent's internal behaviour are omitted. Abstract specifications are given as LTL formulae which describe the external behaviour of the agents, allowing their temporal behaviour to be compactly modelled. We assume that each concrete agent has a *reasoning strategy* (or conflict resolution strategy) which determines the order in which rules are applied when more than one rule matches the contents of the agent's working memory. The system designer can specify a range of conflict resolution strategies (based on those provided in [Culbert, 2007, Friedman-Hill, 2008, Tzafestas et al., 1989]). Both approaches have been combined in an automated verification tool TVRBA for rule-based multi-agent systems which allows the designer to specify information about agents' interaction, behaviour, and execution strategy at different levels of abstraction. The TVRBA tool generates an encoding of the multi-agent system for the Maude LTL model checker, allowing properties of the system to be verified.

We illustrated the scalability of this approach by comparing it to results presented in Chapter 5 for a synthetic distributed reasoning problem. We also showed how to further improve scalability by using abstract agents specified in terms of temporal doxastic formulae. Even with the initial prototype implementation of the TVRBA tool, the results from the case studies suggest that new approach scales significantly better than the approach presented in Chapter 5 that uses traditional model checking. We believe that the new approach can form a useful framework for the scalable verification of rule-based multi-agent systems. Although the approach assumes rule-based agents, the basic idea can be implemented for other reasoning systems. How could it be extended to other reasoning systems, such as agents which reason using resolution, is discussed below in § 9.2.2.

9.2 Future work

The scope of the work described in this thesis has raised a number of issues which could be addressed in future work. In this section, we highlight three different directions, namely: practical applications of the current framework, extensions to the framework presented in this thesis, and a re-engineering the Maude LTL model checker to allow reasoning strategies to be propagated down to the model checking level.

9.2.1 Potential application areas

In this thesis, we have mainly focused on synthetic rule-based examples. A fruitful area for future work would be to the application of the framework and the TVRBA tool to the verification of resource tradeoffs in practical MAS applications where resources are particularly important. Sensor networks have emerged as a promising new monitoring and control solution for a variety of applications. However, sensor nodes are resource-bounded having a of relatively small amount of physical memory, processing power, power supply and communication throughput. Note that communication between sensor nodes consumes most of the available power. In order to increase the life time of sensor nodes, the amount of information broadcast to all sensor nodes should be minimised. Each sensor node should make local decisions in order to determine what information should be communicated, and to whom. For example, instead of broadcasting all the temperature readings, a sensor node may only send the average of temperature readings taken over a specified amount of time. Interesting properties of such systems that can be verified include for example "data always reaches the base station from the source node in n_T time steps while exchanging fewer than n_C messages, in all topologies where there is a path between them ". In the literature, rule-based approaches have been used for the design and implementation of wireless sensor networks including those studied by [Chu et al., 2007, Terfloth and Schiller, 2008, Baliosian et al., 2009]. Future work will look at exploring case studies in this area to demonstrate the utility of the proposed framework.

Another potential area of future application could be to verify systems focusing on business rules represented in Hornlog RuleML [Hirtle et al., 2006]. For example, there has been work on rule-based approaches to representation of business contacts that enables software agents to create, evaluate, negotiate, and execute contacts [Grosof and Poon, 2003, Governatori, 2005, Boley et al., 2010]. As an example, consider an agent which acts as a provider agent for a company which is in charge of quoting for products, and also decides when a discount should be given to a potential customer (e.g., premium customers are entitled a 5% discount on new orders on regular products [Boley et al., 2010]). The designer of such an agent may wish to ensure that the agent's behaviour is correct. This includes verifying that, for example, the agent does not offer a customer an inappropriate discount. The designer also needs to be able to verify other interesting properties, for example, the designer may wish the system to offer certain quality of service guarantees, e.g., to bound the time a potential customer has to wait between the acknowledgement of a request for a price and receiving a quote.

9.2.2 Extensions to the current framework

In this thesis, the verification approach using abstraction and strategies has been focused on rule-based agents. It would be interesting to extend the approach to other types of reasoning agents, such as agents which reason using resolution. In Maude resolution could be modelled in a similar way to the non-positional encoding discussed in § 4.8.2 in which resolution is limited to sets of valid clauses (tautology free) and valid transition (two resolvable clauses never produce a tautology). Memory bounds, message counters, clause addition and overwriting operations can easily be implemented using simple Maude algebra. Actions by each agent such as reading a premise, resolving two clauses, and communication with other agents can be implemented using equations and rules in Maude. Agents could communicate clauses using Ask and Tell communication primitives. As in the rule-based case, abstract specifications consisting of a set of LTL formulae could be used to specify some of the

agents in the system. The set of LTL formulae extended with doxastic operators would be similar to those defined in § 6.4. For example, for some clause C,[1] the formula $G(B_i\ \mathsf{Ask}(i, j, C) \rightarrow\ X^n\ B_j\ \mathsf{Tell}(j, i, C))$ describes agents which are always guaranteed to reply to a request for information within n timesteps. Here C is a propositional Horn clause that we considered in Chapter 4, however we would like to extend the resolution-based systems from propositional to first-order. In order to reduce branching factor of the models, we may consider e.g., the following reasoning strategies for agents which reason with Horn clauses:

- **resolution:** always resolve the shortest clauses first;
- **overwrite:** always overwrite the oldest clause in memory when agents run out of space;
- **subsumption:** eliminate subsumed clauses when agents read clauses from their knowledge base or perform a resolution or a Copy operation. For example, if the clause A_1 is already in an agent's memory then clauses subsumed by A_1 should not be read again from the KB. Furthermore, an agent will not be allowed to perform a resolution at the current state for which A_1 subsumes the resolvent.

The development of an automated tool would be similar to the TVRBA. The automated tool will allow the designer of a multi-agent resolution-based system to specify the information about agents' interaction, behaviour, and execution strategy at different levels of abstraction. The existing parser can easily be extended to support the syntax of Horn clauses, and the part of the TVRBA which implements the abstract agents can be reused. The other parts of the TVRBA which implement the reasoning strategies and concrete agents would need to be changed to support agents which reason using resolution. This includes the implementation of the inference rule of resolution and reasoning strategies. Ultimately, the approach could be extended to multi-agent systems in which different agents reason in different ways.

[1] For example, C could be any clause from the set F (cf. § 4.6).

9.2.3 Re-engineering the Maude LTL model checker

A basic strategy language for Maude has been developed by [Eker et al., 2007]. The Maude strategy language can be used to control how rules are applied to rewrite a term. However, the Maude LTL model checker does not support the strategy language. Model checking in the presence of strategies is an interesting research problem. Re-engineering the Maude LTL model checker to integrate the strategy language would involve redefining the satisfaction relation of LTL formulae in the Maude's LTL-SIMPLIFIER module, and employing heuristic guided search algorithms. While this would involve a non-trivial amount of work, the integration of strategy language into the model checker would give the system designer greater flexibility in allowing them to specify new agent conflict resolution strategies.

Bibliography

NuSMV examples: the collection. http://nusmv.fbk.eu/examples/examples.html.

A. Albore, N. Alechina, P. Bertoli, C. Ghidini, B. Logan, and L. Serafini. Model-checking memory requirements of resource-bounded reasoners. In *Proceedings of the Twenty-First Conference on Artificial Intelligence*, pages 213–218, Boston, Massachusetts, 2006.

N. Alechina, B. Logan, and M. Whitsey. A complete and decidable logic for resource-bounded agents. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multi-Agent Systems*, pages 606–613, New York, USA, 2004.

N. Alechina, M. Jago, and B. Logan. Modal logics for communicating rule-based agents. In *Proceedings of the 17th European Conference on Artificial Intelligence (ECAI'06)*, pages 322–326. IOS Press, 2006.

N. Alechina, P. Bertoli, C. Ghidini, M. Jago, B. Logan, and L. Serafini. Model-checking space and time requirements for resource-bounded agents. In *Proceedings of the Fourth Workshop on Model Checking and Artificial Intelligence*, volume 4428 of *Lecture Notes in Artificial Intelligence*, pages 19–35. Springer-Verlag, 2007.

N. Alechina, B. Logan, H. N. Nguyen, and A. Rakib. Verifying time, memory and communication bounds in systems of reasoning agents. In *Proceedings of the Seventh International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2008)*. IFAAMAS, 2008a.

N. Alechina, B. Logan, H. N. Nguyen, and A. Rakib. Verifying time and communication costs of rule-based reasoners. In *Proceedings of the 5th International Workshop on Model Checking and Artificial Intelligence*, volume 5348 of *Lecture Notes in Computer Science*, pages 1–14. Springer-Verlag, 2008b.

N. Alechina, B. Logan, N. H. Nga, and A. Rakib. Verifying time, memory and communication bounds in systems of reasoning agents. *Synthese*, 169(2):385–403, April 2009a.

N. Alechina, B. Logan, N. H. Nga, and A. Rakib. Reasoning about other agents' beliefs under bounded resources. In J.-J. Meyer and J. Broersen, editors, *Post-proceedings of KR2008-workshop on Knowledge Representation for Agents and Multi-Agent Systems (KRAMAS), Sydney, September 2008*, volume 5605 of *Lecture Notes in Artificial Intelligence*, pages 1–15. Springer-Verlag, 2009b.

N. Alechina, B. Logan, N. H. Nga, and A. Rakib. Automated verification of resource requirements in multi-agent systems using abstraction. In Ron van der Meyden and Jan-Georg Smaus, editors, *Proceedings of the Sixth Workshop on Model Checking and Artificial Intelligence (MoChArt-2010)*, Atlanta, GA, July 2010.

M. Alekhnovich, E. Ben-Sasson, A. A. Razborov, and A. Wigderson. Space complexity in propositional calculus. *SIAM Journal of Computing*, 31(4):1184–1211, 2002.

M. Alpuente, M. A. Feliu, C. Joubert, and A. Villanueva. Implementing Datalog in Maude. In *Proceedings of the IX Jornadas sobre Programación y Lenguajes (PROLE'09)*, San Sebastián, Spain, 2009.

R. Alur and D. Dill. Automata for modelling real-time systems. In *Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP'90)*, volume 443 of *Lecture Notes in Computer Science*, pages 322–335. Springer-Verlag, 1990.

R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

R. Alur, C. Courcoubetis, and D. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2–34, 1993.

R. Alur, T. A. Henzinger, F. Y. C. Mang, S. Qadeer, S. K. Rajamani, and S. Tasiran. MOCHA: Modularity in model checking. In *Proceedings of the 10th International Conference on Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 521–525. Springer-Verlag, 1998a.

R. Alur, T.A. Henzinger, and O. Kupferman. Alternating-time temporal logic. In *Revised Lectures from the International Symposium on Compositionality: The Significant Difference*, pages 23–60, London, UK, 1998b. Springer-Verlag.

L. Astefanoaei, M. Dastani, John-Jules Ch. Meyer, and F. S. de Boer. A verification framework for normative multi-agent systems. In *Proceedings of the 11th Pacific Rim International Conference on Multi-Agents (PRIMA 2008)*, volume 5357 of *Lecture Notes in Computer Science*, pages 54–65. Springer-Verlag, 2008.

C. Baier and J. P. Katoen. *Principles of Model Checking*. The MIT Press, 2008.

J. Baliosian, J. Visca, E. Grampín, L. Vidal, and M. Giachino. A rule-based distributed system for self-optimization of constrained devices. In *Proceedings of the 11th IFIP/IEEE international conference on Symposium on Integrated Network Management*, pages 41–48, Piscataway, NJ, USA, 2009. IEEE Press.

A. Baltag and L.S. Moss. Logics for epistemic programs. *Synthese, 139:165-224*, Knowledge, Rationality & Action 1-60, 2004.

M. Benerecetti, F. Giunchiglia, L. Serafini, M. Benerecetti, and L. Serafini. Model checking multiagent systems. *Journal of Logic and Computation*, 8(3):401–423, 1998.

S. Berezin and S. V. A. Campos E. M. Clarke. Compositional reasoning in model checking. In *Revised Lectures from the International Symposium on Compositionality: The Significant Difference*, pages 81–102, London, UK, 1998. Springer-Verlag.

P. Bertoli, A. Cimatti, M. Pistore, M. Roveri, and P. Traverso. MBP: a model based planner. In *Proceedings of the IJCAI'01 Workshop on Planning under Uncertainty and Incomplete Information*, pages 93–97, 2001.

R. Bharadwaj. *Tools to support a formal verification method for systems with concurrency and nondeterminism*. PhD thesis, McMaster University, 1996.

P. Blackburn, M. de Rijke, and Y. Venema. *Modal Logic*. Cambridge University Press, 2001.

H. Boley, A. Paschke, and O. Shafiq. RuleML 1.0: The overarching specification of web rules. In *Proceedings of the International Symposium on Rule Representation, Interchange and Reasoning on the Web (RuleML 2010)*, volume 6403 of *Lecture Notes in Computer Science*, pages 162–178. Springer-Verlag, 2010.

G. Boone. Concept features in re:agent, an intelligent email agent. In *Proceedings of the Second International Conference on Autonomous Agents*, pages 141–148. ACM Press, 1998.

R. H. Bordini, M. Fisher, C. Pardavila, V. Willem, and M. Wooldridge. Model checking multi-agent programs with CASP. In *Proceedings of the 15th International Conference on Computer Aided Verification*, volume 2725 of *Lecture Notes in Computer Science*, pages 110–113. Springer-Verlag, 2003.

R. H. Bordini, M. Fisher, W. Visser, and M. Wooldridge. State-space reduction techniques in agent verification. *Proceedings of the Third International Joint Conference on Autonomous Agents and Multi-Agent Systems*, 2:896–903, 2004.

R. H. Bordini, M. Fisher, W. Visser, and M. Wooldridge. Verifying multi-agent programs by model checking. *Autonomous Agents and Multi-Agent Systems*, 12(2): 239–256, 2006.

R. H. Bordini, M. Dastani, and M. Winikoff. Current issues in multi-agent systems development (invited paper). In *Proceedings of the 7th International Workshop Engineering Societies in the Agents World(ESAW'06)*, volume 4457 of *Lecture Notes in Computer Science*, pages 38–61. Springer-Verlag, 2007a.

R. H. Bordini, M. Wooldridge, and J. M. Hübner. *Programming Multi-Agent Systems in AgentSpeak using Jason*. John Wiley & Sons, 2007b.

A. Pokahrand L. Braubach and W. Lamersdorf. A flexible BDI architecture supporting extensibility. In *Proceedings of the IEEE/WIC/ACM International Conference on Intelligent Agent Technology*, pages 379–385, Washington, DC, USA, 2005. IEEE Computer Society.

R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transaction on Computers*, 35(8):677–691, 1986.

R. E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. Technical report, Carnegie Mellon University, Pittsburgh, PA, USA, 1992.

R. E. Bryant and C. Meinel. Ordered Binary Decision Diagrams In Electronic Design Automation: Foundations, Applications and Innovations. Technical report, Universität Trier, Mathematik/Informatik, Forschungsbericht, 2002.

S. Budkowski and P. Dembinski. An introduction to Estelle: a specification language for distributed systems. *Computer Networks and ISDN Systems*, 14(1):3–23, 1987.

J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13:401–424, 1993.

T. J. Callantine. CATS -based air traffic controller agents. NASA Contractor Report 2002-211856. Moffett Field,CA: NASA Ames Research Center, 2002.

T. J. Callantine. Air traffic controller agents. In *Proceedings of the Second International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2003)*. IFAAMAS, 2003.

D. Chaum. The dining cryptographers problem: unconditional sender and recipient untraceability. *Journal of Cryptology*, 1(1):65–75, 1988.

D. Chu, L. Popa, A. Tavakoli, J. M. Hellerstein, P. Levis, S. Shenker, and I. Stoica. The design and implementation of a declarative sensor network system. In *Proceedings of the 5th international conference on Embedded networked sensor systems*, pages 175–188. ACM, 2007.

A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV : A new Symbolic Model Checker. *International Journal on Software Tools for Technology Transfer*, 2 (4):410–425, 2000.

A. Cimatti, M. Pistore, M. Roveri, and P. Traverso. Weak, strong, and strong cyclic planning via symbolic model checking. *Artificial Intelligence*, 147(1-2):35–84, 2003.

E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16:1512–1542, September 1994.

E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge,Massachusetts, 2000.

M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, and J. Meseguer. *All About Maude - A High Performance Logical Framework*, volume 4350 of *Lecture Notes in Computer Science*. Springer-Verlag, 2007.

M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *Maude Manual (Version 2.4)*. SRI Internationa, Menlo Park, CA 94025, USA, 2008.

C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1(2-3):275–288, 1992.

P. Cousot. Automatic verification by abstract interpretation. In *Verification, Model Checking, and Abstract Interpretation, 4th International Conference(VMCAI'03)*, volume 2575 of *Lecture Notes in Computer Science*, pages 20–24, New York, NY, USA, 2003.

P. Cousot. The verification grand challenge and abstract interpretation. In *Verified Software: Theories, Tools, Experiments, First IFIP TC 2/WG 2.3 Conference(VSTTE'05)*, volume 4171 of *Lecture Notes in Computer Science*, pages 189–201, 2008.

P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977.

P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE analyser. In *The European Symposium on Programming*, volume 3444 of *Lecture Notes in Computer Science*, pages 21–30. Springer-Verlag, 2005.

C. Culbert. *CLIPS reference manual*. NASA, 2007.

B. D'Angelo, S. Sankaranarayanan, C. Sánchez, W. Robinson, B. Finkbeiner, H. B. Sipma, S. Mehrotra, and Z. Manna. LOLA: Runtime monitoring of synchronous systems. In *Proceedings of the 12th International Symposium on Temporal Representation and Reasoning*, pages 166–174, 2005.

G. de Giacomo, Y. Lespérance, and H. J. Levesque. GOLOG: A logic programming language for dynamic domains.s. *Journal of Logic Programming*, 31:59–84, 1997.

G. de Giacomo, Y. Lespérance, and H. J. Levesque. ConGolog: a concurrent programming language based on the situation calculus. *Artificial Intelligence*, 121:109–169, 2000.

L. A. Dennis, B. Farwer, R. H. Bordini, and M. Fisher. A flexible framework for verifying agent programs. In *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multi-Agent Systems*, pages 1303–1306, Richland, SC, 2008a. IFAAMS.

L. A. Dennis, B. Farwer, R. H. Bordini, M. Fisher, and M. Wooldridge. A common semantic basis for BDI languages. In *Proceedings of the 5th workshop on Programming Multi-Agent Systems*, volume 4908 of *Lecture Notes in Computer Science*, pages 124–139. Springer-Verlag, 2008b.

H. Ditmarsch, W. van der Hoek, R. van der Meyden, and J. Ruan. Model checking russian cards. In K. Jensen and A. Podelski, editors, *Proceedings of the Third International Workshop on Model Checking and Artificial Intelligence*, volume 149(2). Electronic Notes in Theoretical Computer Science, 2005.

C. Dixon, M. Gago, M. Fisher, and W. van der Hoek. Temporal logics of knowledge and their applications in security. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 186:27–42, July 2007.

A. Doroś, A. Janowska, and P. Janowski. From specification languages to timed automata. In *Proceedings of the Int. Workshop on Concurrency, Specification and Programming*, volume 161, pages 117–128, Humboldt University, 2002.

H. N. Duc. Logical omniscience vs. logical ignorance on a dilemma of epistemic logic. In *Progress in Artificial Intelligence, 7th Portuguese Conference on Artificial Intelligence*, pages 237–248, 1995.

H. N. Duc. Reasoning about rational, but not logically omniscient, agents. *Journal of Logic and Computation*, 7(5):633–648, 1997.

C. Eisner and D. Peled. Comparing symbolic and explicit model checking of a software system. In *Proceedings of the 9th International SPIN Workshop*, volume 2318 of *Lecture Notes in Computer Science*, pages 79–82. Springer-Verlag, 2002.

S. Eker, J. Meseguer, and A. Sridharanarayanan. The Maude LTL model checker and its implementation. In *Proceedings of the 10th International SPIN Workshop*, volume 2648 of *Lecture Notes in Computer Science*, pages 623–624. Springer-Verlag, 2003.

S. Eker, N. Martí-Oliet, J. Meseguer, and A. Verdejo. Deduction, Strategies, and Rewriting. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 174:3–25, July 2007.

J. J. Elgot-Drapkin and D. Perlis. Reasoning situated in time I: basic concepts. *Journal of Experimental and Theoretical Artificial Intelligence*, 2(1):75–98, 1990.

J. J. Elgot-Drapkin, D. Perlis, and M. Miller. Memory, Reason, and Time: the Step-logic Approach. *Philosophy and AI: Essays at the Interface*, pages 79–103, 1991.

E. A. Emerson and C. L. Lei. Modalities for model checking: Branching time strikes back. In *Science of Computer Programming*, volume 8, pages 275–306. Elsevier North-Holland, Inc, 1987.

K. Engelhardt, R. van der Meyden, and Y. Moses. Knowledge and the logic of local propositions. In *Proceedings of the 7th conference on Theoretical aspects of rationality and knowledge*, pages 29–41, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

J. L. Esteban and J. Torán. Space bounds for resolution. In *Proceedings of the 16th Annual Symposium on Theoretical Aspects of Computer Science*, volume 1563 of *Lecture Notes in Computer Science*, pages 551–560. Springer-Verlag, 1999.

R. Fagin and J. Y. Halpern. Belief, awareness, and limited reasoning: Preliminary report. In *Proceedings of the International Joint Conference on Artifical intelligence*, pages 491–501, 1985.

R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. *Reasoning about Knowledge*. MIT Press, Cambridge, Mass, 1995.

T. Finin, R. Fritzson, D. McKay, and R. McEntire. KQML as an agent communication language. In *Proceedings of the Third International Conference on Information and Knowledge Management*, pages 456–463, New York, NY, USA, 1994. ACM.

M. Fisher. Concurrent METATEM - A language for modelling reactive systems. In *Proceedings of the 5th International Conference on Parallel Architectures and Languages Europe*, pages 185–196, London, UK, 1993. Springer-Verlag.

M. Fisher and C. Ghidini. Programming resource-bounded deliberative agents. In *Proceedings of the 16th International Joint Conference on Artifical Intelligence*, pages 200–205, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

M. Fisher and M. Wooldridge. On the formal specification and verification of multi-agent systems. *International Journal of Cooperative Information Systems*, 6(1): 37–66, 1997.

M. Fisher, R. H. Bordini, B. Hirsch, and P. Torroni. Computational logics and agents: A road map of current technologies and future trends. *Computational Intelligence*, 23(1):61–91, 2007.

E. J. Friedman-Hill. *Jess, The Rule Engine for the Java Platform*. Sandia National Laboratories, 2008.

M. Gallardo, J. Martínez, P. Merino, and E. Pimentel. A Tool for Abstraction in Model Checking. *Electronic Notes in Theoretical Computer Science*, 66(2):17, 2002.

J. H. Gallier. *Logic for Computer Science: Foundations of Automatic Theorem Proving*. Harper & Row Publishers, 1986.

P. Gammie and R. van der Meyden. MCK: Model checking the logic of knowledge. In *Proceedings of International Conference on Computer Aided Verification-2004*, volume 3114 of *Lecture Notes in Computer Science*, pages 479–483. Springer-Verlag, 2004.

G. Gardey, D. Lime, M. Magnin, and O. r H. Roux. Romeo: A Tool for Analyzing Time Petri Nets. In *Proceedings of the 17th International Conference on Computer Aided Verification*, volume 3576 of *Lecture Notes in Computer Science*. Springer-Verlag, 2005.

G. De Giacomo, Y. Lespérance, and H. J. Levesque. ConGolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, 121(1-2): 109–169, 2000.

P. Godefroid. Model checking for programming languages using VeriSoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 174–186, New York, NY, USA, 1997. ACM.

G. Governatori. Representing business contracts in RuleML. *International Journal of Cooperative Information Systems*, 14(2-3):181–216, 2005.

B. N. Grosof and T. C. Poon. Representing agent contracts with exceptions using XML rules, ontologies, and process descriptions. In *Proceedings of the 12th International Conference on World Wide Web*, pages 340–349, New York, NY, USA, 2003. ACM.

A. Haken. *The intractability of resolution (complexity)*. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1984.

J. Y. Halpern and M. Y. Vardi. Model checking vs. theorem proving: A manifesto. In *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning (KR'91)*, pages 325–334. Morgan Kaufmann Publishers, 1991.

J. Y. Halpern, Y. Moses, and M. Y. Vardi. Algorithmic knowledge. In *Proceedings of the 5th Conference on Theoretical Aspects of Reasoning about Knowledge*, pages 255–266, 1994.

K. Havelund and G. Rosu. Java PathExplorer - A Runtime Verification Tool. In *Proceedings of the 6th International Symposium on Artificial Intelligence, Robotics and Automation in Space: A New Space Odyssey*, pages 200–217, 2001.

T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software Verification with BLAST. In *Proceedings of the 10th SPIN Workshop on Model Checking Software (SPIN)*, volume 2648 of *Lecture Notes in Computer Science*, pages 235–239. Springer-Verlag, 2003.

J. Hintikka. *Knowledge and Belief*. New York, 1962.

D. Hirtle, H. Boley, B. Grosof, M. Kifer, M. Sintek, S. Tabet, and G. Wagner. Schema Specification of RuleML 0.91. http://ruleml.org/0.91/, 2006.

G. Holzmann, D. Peled, and M. Yannakakis. On nested depth first search (extended abstract). In *Poceeding of the second Workshop on the SPIN Verification system*, pages 23–32. American Mathematical Society, 1996.

G. J. Holzmann. On-the-fly model checking. *ACM Computing Surveys*, 28(4), December 1996.

G. J. Holzmann. The model checker SPIN. *IEEE Transaction on Software Engineering*, 23(5):279–295, 1997.

A. J. Hu, G. York, and D. L. Dill. New techniques for efficient verification with implicitly conjoined BDDs. In *Proceedings of the 31st Annual Design Automation Conference*, pages 276–282, New York, NY, USA, 1994. ACM.

G. Huet, G. Kahn, and C. P. Mohring. The coq proof assistant: A tutorial. INRIA, France, 2009.

U. Hustadt and R. A. Schmidt. MSPASS: Modal reasoning by translation and first-order resolution. In *Proceedings of the International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*, volume 1847 of *Lecture Notes in Computer Science*, pages 67–71, London, UK, 2000. Springer-Verlag.

U. Hustadt, B. Konev, A. Riazanov, and A. Voronkov. TeMP: A temporal monodic prover. In *Proceedings of the Second International Joint Conference IJCAR 2004*, volume 3097 of *Lecture Notes in Computer Science*, pages 326–330. Springer-Verlag, 2004.

M. Jago. *Logics for resource-bounded agents*. PhD thesis, University of Nottingham, 2006.

N. R. Jennings and M. Wooldridge. Applications of intelligent agents. In *Agent technology*, pages 3–28. Springer-Verlag, 1998.

K. Konolige. *A Deduction Model of Belief*. Morgan Kaufmann Publishers Inc. San Francisco, CA, USA, 1986.

S. Kripke. Semantical Analysis of Modal Logic I. Normal Propositional Calculi. *Zeitschrift fur mathematische Logik und Grundlagen der Mathematik*, 9:67–96, 1963.

L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, 1977.

Y. Lespérance, H. J. Levesque, F. Lin, D. Marcu, R. Reiter, and R. B. Scherl. Foundations of a logical approach to agent programming. In *ATAL*, pages 331–346, 1995.

Y. Lespérance, T. G. Kelley, J. Mylopoulos, and E. S. K. Yu. Modeling dynamic domains with ConGolog. In *Proceedings of the 11th International Conference on Advanced Information Systems Engineering*, pages 365–380, London, UK, 1999. Springer-Verlag.

H. J. Levesque. A logic of implicit and explicit belief. In *AAAI*, pages 198–202, 1984.

O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 97–107. ACM New York, NY, USA, 1985.

A. Lomuscio and W. Penczek. Symbolic model checking for temporal epistemic logic. *SIGACT News Logic Column*, 38(3):76–100, 2007.

A. Lomuscio, W. Penczek, and B. Woźna. Bounded model checking for knowledge and real time. *Artificial Intelligence*, 171(16-17):1011–1038, 2007.

A. Lomuscio, H. Qu, and F. Raimondi. MCMAS: A model checker for the verification of multi-agent systems. In *Proceedings o 21st International Conference on Computer Aided Verification*, pages 682–688, 2009.

D. Magazzeni. *Explicit Model Checking Techniques Applied to Control and Planning Problems*. PhD thesis, Dipartimento di Informatica, Universitȁ di L'Aquila, 2009.

F. Martin. PAG : an efficient program analyzer generator. *International Journal on Software Tools for Technology Transfer*, 2(1):46–67, 1998.

J. McCarthy and P. J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. pages 26–45, 1987.

K. McMillan. The SMV system. Technical Report CMU-CS-92-131, Carnegie-Mellon University, 1992.

J. Meseguer. Rewriting as a unified model of concurrency. In *CONCUR '90: theories of concurrency–unification and extension, Amsterdam, the Netherlands*, volume 458 of *Lecture Notes in Computer Science*, pages 384–400. Springer-Verlag, 1990.

J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.

J. J. C. Meyer and W. van der Hoek. *Epistemic Logic for AI and Computer Science*. Cambridge University Press, New York, NY, USA, 1995.

W. Nabialek, A. Niewiadomski, W. Penczek, A. Pólrola, and M. Szreter. 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, 2004.

R. M. Needham and M. D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21:993–999, December 1978.

M. Negnevitsky. *Artificial Intelligence. A Guide to Intelligent Systems*. Addison-Wesley, 2005.

H. N. Nga. *Logics for resource-bounded multi-agent systems*. PhD thesis, University of Nottingham, 2010.

F. Nielson, H. R. Nielson, and C. Hankin. *Principles of program analysis*. Springer-Verlag Berlin Heidelberg, Printed in Germany, 2005.

S. Owre, J. M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In *Proceedings of the 11th International Conference on Automated Deduction*, pages 748–752, London, UK, 1992. Springer-Verlag.

F. Paganelli and D. Giuli. An ontology-based context model for home health monitoring and alerting in chronic patient care networks. In *Proceedings of the 21st International Conference on Advanced Information Networking and Applications Workshops - Volume 02*, pages 838–845, Washington, DC, USA, 2007. IEEE Computer Society.

J. Pan. Software testing (18-849b dependable embedded systems). Technical report, 1999.

W. Penczek and A. Lomuscio. Verifying epistemic properties of multi-agent systems via bounded model checking. *Fundamenta Informaticae*, 55(2):167–185, 2002.

G. Platt, J. Wall, P. Valencia, and J. K. Ward. The tiny agent - wireless sensor networks controlling energy resources. *Journal of Networks*, 3(4):42–50, 2008.

A. Pnueli. The temporal semantics of concurrent programs. In *Semantics of Concurrent Computation: Proceedings of the International Symposium*, volume 70 of *Lecture Notes in Computer Science*, pages 1–20. Springer-Verlag, 1979.

B. R. Preiss. *Data structures and algorithms with object-oriented design patterns in C++*. John Wiley & Sons, Inc., 1999.

F. Raimondi and A. Lomuscio. Automatic verification of multi-agent systems by model checking via ordered binary decision diagrams. *Journal of Applied Logic*, 5(2):235–251, 2007.

R. K. Ranjan, A. Aziz, R. K. Brayton, B. Plessier, and C. Pixley. Efficient BDD algorithms for FSM synthesis and verification. In *IEEE/ACM Proceedings International Workshop on Logic Synthesis, Lake Tahoe (NV)*, 1995.

A. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In *Proceedings of the 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World : Agents Breaking Away*, volume 1038 of *Lecture Notes in Computer Science*, pages 42–55, Secaucus, NJ, USA, 1996.

A. S. Rao and M. P. Georgeff. Modeling rational agents within a BDI-architecture. In James Allen, Richard Fikes, and Erik Sandewall, editors, *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning*, pages 473–484. Morgan Kaufmann publishers Inc.: San Mateo, CA, USA, 1991.

A. S. Rao and M. P. Georgeff. A model-theoretic approach to the verification of situated reasoning systems. In *Proceedings of the 13th International Joint Conference on Artifical intelligence*, pages 318–324, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

W. Reisig. *Petri nets: an introduction*. Springer-Verlag New York, Inc, New York, NY, USA, 1985.

J. Rushby. Theorem Proving for Verification. In *Modelling and Verification of Parallel Processes: MOVEP 2000*, volume 2067 of *Lecture Notes in Computer Science*, pages 39–57. Springer Verlag, 2001.

R. B. Scherl and H. J. Levesque. The frame problem and knowledge-producing actions. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 689–695. AAAI Press/The MIT Press, 1993.

Ph. Schnoebelen. The complexity of temporal logic model checking. In *Advances in Modal Logic*, pages 393–436, 2002.

S. Shapiro, Y. Lespérance, and H. J. Levesque. Specifying communicative multi-agent systems with ConGolog. In *Working Notes of the AAAI Fall 1997 Symposium on Communicative Action in Humans and Machines*, pages 72–82, Cambridge, MA, Novemeber 1997. AAAI Press.

S. Shapiro, Y. Lespérance, and H. J. Levesque. The cognitive agents specification language and verification environment for multiagent systems. In *Proceedings of the*

first International Joint Conference on Autonomous Agents and Multi-Agent Systems, pages 19–26, New York, NY, USA, 2002. ACM.

S. Sharma, G. Gopalakrishnan, E. Mercer, and J. Holt. MCC: A runtime verification tool for MCAPI user applications. In *Proceedings of the 9th International Conference on Formal Methods in Computer-Aided Design*, pages 41–44, 2009.

H. Shin, Y. Endoh, and Y. Kataoka. ARVE: Aspect-oriented runtime verification environment. In *Runtime Verification, 7th International Workshop, RV 2007*, volume 4839 of *Lecture Notes in Computer Science*, pages 87–96. Springer-Verlag, 2007.

Y. Shoham. Agent oriented programming. *Journal of Artificial Intelligence*, 60(1): 51–92, 1993.

A. P Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. *Journal of the ACM*, 32(3):733–749, 1985.

A. Srinivasan, T. Kam, S. Malik, and R. K. Brayton. Algorithms for discrete function manipulation. In *Proceedings of the IEEE International Conference on Computer-Aided Design (ICCAD'90)*, pages 92–95, 1990.

Z. Sun and G. R. Finnie. *Intelligent techniques in E-Commerce*. Springer-Verlag Berlin Heidelberg, Printed in Germany, 2004.

PolySpace Technologies. PolySpace Client/Server for C/C++/Ada. http://www.mathworks.com/products/polyspace/, 2008.

K. Terfloth and J. Schiller. Ruling Networks with RDL: A domain-specific language to task wireless sensor networks. In *Proceedings of the International Symposium on Rule Representation, Interchange and Reasoning on the Web*, volume 5321 of *Lecture Notes in Computer Science*, pages 127–134. Springer-Verlag, 2008.

R. Tynan, D. Marsh, D. O'Kane, and G. M. P. O'Hare. Intelligent agents for wireless sensor networks. In *Proceedings of the Fourth International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2008)*. IFAAMAS, 2005.

S. Tzafestas, S. Ata-Doss, and G. Papakonstantinou. *Knowledge-Base System Diagnosis, Supervision and Control*. New York, London, Plenum Press, 1989.

W. van der Hoek and M. Wooldridge. Model checking knowledge and time. In *Proceedings of the 9th International SPIN Workshop*, pages 95–111, 2002.

R. van der Meyden and N. V. Shilov. Model checking knowledge and time in systems with perfect recall (extended abstract). In *Proceedings of the 19th Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 432–445, London, UK, 1999. Springer-Verlag.

R. van der Meyden and K. Su. Symbolic model checking the knowledge of the dining cryptographers. In *Proceedings of the 17th IEEE workshop on Computer Security Foundations*, pages 280–291, Washington, DC, USA, 2004. IEEE Computer Society.

M. Birna van Riemsdijk, F. S. de Boer, M. Dastani, and John-Jules Ch. Meyer. Prototyping 3APL in the Maude term rewriting language. In *Proceedings of the Seventh International Workshop on Computational Logic in Multi-Agent Systems (CLIMA VII, Hakodate, Japan, May 2006)*, volume 4371 of *Lecture Notes in Computer Science*, pages 95–114. Springer-Verlag, 2007.

M. Y. Vardi. Branching vs. linear time: Final showdown. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *Lecture Notes in Computer Science*, pages 1–22. Springer-Verlag, 2001.

W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering*, 10:203–232, 2003.

W. Wang, Z. Hidvegi, A. B. Bailey, and A. D. Whinston. Model checking - a rigorous and efficient tool for e-commerce internal control and assurance. Gozuita School Business, Emory University, Atlanta, Georgia, USA, 2001.

M. Wooldridge. *Reasoning about Rational Agents*. The MIT Press:Cambridge, MA, 2000.

M. Wooldridge. *An Introduction to Multi-Agent Systems*. John Wiley & Sons Inc, 2009.

M. Wooldridge, M. Fisher, M. P. Huget, and S. Parsons. Model checking multi-agent systems with MABLE. In *Proceedings of the First International Joint Conference on Autonomous Agents and Multi-Agent Systems*, pages 952–959. ACM, 2002.

A. C.-C. Yao. Some complexity questions related to distributive computing (preliminary report). In *Conference Record of the Eleventh Annual ACM Symposium on Theory of Computing*, pages 209–213. ACM, 1979.

Appendix

A Proof of theorem 5.5.1

$Proof$. Let us recall that a tree with n leaf nodes, at level h the number of nodes is $2^h = n$, at level $h-1$ the number of nodes is $2^{h-1} = 2^h/2 = n/2$ and so on. We also recall the expression $\mathcal{N} = \sum_{l=0}^{h} N_l$ (cf. Eq. 5.2), which gives the number of reachable states of the system. Now we can expand N_j, for $0 \leq j \leq h$ as follows:

$$
\begin{aligned}
N_0 &= {}^1C_1 \cdot {}^2C_2 \cdot \ldots \cdot {}^{\frac{n}{2}}C_{\frac{n}{2}} \cdot {}^nC_n \\
&= 1
\end{aligned}
$$

$$
\begin{aligned}
N_1 &= {}^2C_1 \cdot (\sum_{i=0}^{2} {}^2C_i) \cdot (\sum_{i=0}^{4} {}^4C_i) \cdot \ldots \cdot (\sum_{i=0}^{\frac{n}{4}} {}^{\frac{n}{4}}C_i) \cdot {}^nC_n \\
&\quad + {}^2C_2 \cdot {}^4C_4 \cdot \ldots \cdot {}^{\frac{n}{2}}C_{\frac{n}{2}} \cdot {}^nC_n \\
&= {}^2C_1 \cdot 2^2 \cdot 2^4 \cdot \ldots \cdot 2^{\frac{n}{4}} \\
&\quad + 1
\end{aligned}
$$

$$
\begin{aligned}
N_2 &= {}^4C_1 \cdot (\sum_{i=0}^{6} {}^6C_i) \cdot (\sum_{i=0}^{12} {}^{12}C_i) \cdot \ldots \cdot (\sum_{i=0}^{\frac{3n}{8}} {}^{\frac{3n}{8}}C_i) \cdot {}^nC_n \\
&\quad + {}^4C_2 \cdot (\sum_{i=0}^{4} {}^4C_i) \cdot (\sum_{i=0}^{8} {}^8C_i) \cdot \ldots \cdot (\sum_{i=0}^{\frac{2n}{8}} {}^{\frac{2n}{8}}C_i) \cdot {}^nC_n \\
&\quad + {}^4C_3 \cdot (\sum_{i=0}^{2} {}^2C_i) \cdot (\sum_{i=0}^{4} {}^4C_i) \cdot \ldots \cdot (\sum_{i=0}^{\frac{n}{8}} {}^{\frac{n}{8}}C_i) \cdot {}^nC_n \\
&\quad + {}^4C_4 \cdot {}^8C_8 \cdot \ldots \cdot {}^{\frac{n}{2}}C_{\frac{n}{2}} \cdot {}^nC_n \\
&= {}^4C_1 \cdot 2^6 \cdot 2^{12} \cdot \ldots \cdot 2^{\frac{3n}{8}} \\
&\quad + {}^4C_2 \cdot 2^4 \cdot 2^8 \cdot \ldots \cdot 2^{\frac{2n}{8}} \\
&\quad + {}^4C_3 \cdot 2^2 \cdot 2^4 \cdot \ldots \cdot 2^{\frac{n}{8}} \\
&\quad + 1
\end{aligned}
$$

In a similar fashion, by expanding N_j for $3 \leq j \leq h$ we obtain the following expressions:

$$
\begin{aligned}
N_3 &= {}^8C_1 \cdot 2^{14} \cdot 2^{28} \cdot \ldots \cdot 2^{\frac{7n}{16}} \\
&\quad + {}^8C_2 \cdot 2^{12} \cdot 2^{24} \cdot \ldots \cdot 2^{\frac{6n}{16}} \\
&\quad + {}^8C_3 \cdot 2^{10} \cdot 2^{20} \cdot \ldots \cdot 2^{\frac{5n}{16}} \\
&\quad + {}^8C_4 \cdot 2^{8} \cdot 2^{16} \cdot \ldots \cdot 2^{\frac{4n}{16}} \\
&\quad + {}^8C_5 \cdot 2^{6} \cdot 2^{12} \cdot \ldots \cdot 2^{\frac{3n}{16}} \\
&\quad + {}^8C_6 \cdot 2^{4} \cdot 2^{8} \cdot \ldots \cdot 2^{\frac{2n}{16}} \\
&\quad + {}^8C_7 \cdot 2^{2} \cdot 2^{4} \cdot \ldots \cdot 2^{\frac{n}{16}} \\
&\quad + 1 \\
N_4 &= {}^{16}C_1 \cdot 2^{30} \cdot 2^{60} \cdot \ldots \cdot 2^{\frac{15n}{32}} \\
&\quad + {}^{16}C_2 \cdot 2^{28} \cdot 2^{56} \cdot \ldots \cdot 2^{\frac{14n}{32}} \\
&\quad + \\
&\quad \vdots \\
&\quad + {}^{16}C_{15} \cdot 2^{2} \cdot 2^{4} \cdot \ldots \cdot 2^{\frac{n}{32}} \\
&\quad + 1 \\
&\vdots \\
N_{h-3} &= {}^{\frac{n}{8}}C_1 \cdot 2^{\frac{n}{4}-2} \cdot 2^{\frac{n}{2}-4} + {}^{\frac{n}{8}}C_2 \cdot 2^{\frac{n}{4}-4} \cdot 2^{\frac{n}{2}-8} + \ldots + 1 \\
N_{h-2} &= {}^{\frac{n}{4}}C_1 \cdot 2^{\frac{n}{2}-2} + {}^{\frac{n}{4}}C_2 \cdot 2^{\frac{n}{2}-4} + \ldots + 1 \\
N_{h-1} &= {}^{\frac{n}{2}}C_1 + {}^{\frac{n}{2}}C_2 + \ldots + 1 \\
N_h &= 1
\end{aligned}
$$

Therefore, we can define a function $f : \mathbb{N}_+ \to \mathbb{N}$ on $\mathbb{N}_+ = \{2^m : m \geq 1\}$ by (sum of $N_0, N_1, \ldots, N_h$):

$$
\begin{aligned}
f(n) &= [1] \\
&\quad + [({}^2C_1 \cdot 2^2 \cdot 2^4 \cdot \ldots \cdot 2^{\frac{n}{4}}) + 1] \\
&\quad + [({}^4C_1 \cdot 2^6 \cdot 2^{12} \cdot \ldots \cdot 2^{\frac{3n}{8}}) + ({}^4C_2 \cdot 2^4 \cdot 2^8 \cdot \ldots \cdot 2^{\frac{2n}{8}}) + ({}^4C_3 \cdot 2^2 \cdot 2^4 \cdot \ldots \cdot 2^{\frac{n}{8}}) + 1] \\
&\quad + \\
&\quad \vdots \\
&\quad + [{}^{\frac{n}{2}}C_1 + {}^{\frac{n}{2}}C_2 + \ldots + 1] \\
&\quad + [1] \qquad \text{(A1)}
\end{aligned}
$$

Now, let us consider each N_j $(h - 1 \geq j \geq 1)$ for a further simplification:

$$
\begin{aligned}
N_{h-1} &= {}^{\frac{n}{2}}C_1 + {}^{\frac{n}{2}}C_2 + \ldots + 1 \\
&= \sum_{i=0}^{\frac{n}{2}} {}^{\frac{n}{2}}C_i - {}^{\frac{n}{2}}C_0 \\
&= 2^{\frac{n}{2}} - \frac{n}{2} \\
&\leq 2^{\frac{n}{2}}, \forall n \in \mathbb{N}_+ \\
N_{h-2} &= {}^{\frac{n}{4}}C_1 \cdot 2^{\frac{n}{2}-2} + {}^{\frac{n}{4}}C_2 \cdot 2^{\frac{n}{2}-4} + \ldots + {}^{\frac{n}{4}}C_{\frac{n}{8}} \cdot 2^{\frac{n}{2}-\frac{n}{4}} + \ldots + 1 \\
&= \frac{n}{4} \cdot 2^{\frac{n}{2}-2} + \frac{\frac{n}{4}(\frac{n}{4}-1)}{2!} \cdot 2^{\frac{n}{2}-4} + \ldots + \frac{\frac{n}{4}(\frac{n}{4}-1)\ldots(\frac{n}{4}-(\frac{n}{8}-1))}{(\frac{n}{8})!} \cdot 2^{\frac{n}{2}-\frac{n}{4}} + \ldots + 1 \\
&\leq n \cdot 2^{\frac{n}{2}} + \frac{n^2}{2!} \cdot 2^{\frac{n}{2}} + \ldots + \frac{n^{\frac{n}{8}}}{(\frac{n}{8})!} \cdot 2^{\frac{n}{2}} + \ldots + 1 \\
&\leq (\frac{n}{4}-1) \cdot [2^2 \cdot 2^4 \cdot 2^6 \cdot \ldots \cdot 2^{\frac{n}{2}}] + 1, \text{follows from the following proof}
\end{aligned}
$$

Let us consider the most significant term $\frac{n^{\frac{n}{8}}}{(\frac{n}{8})!} \cdot 2^{\frac{n}{2}}$ of the series

$n \cdot 2^{\frac{n}{2}} + \frac{n^2}{2!} \cdot 2^{\frac{n}{2}} + \ldots + \frac{n^{\frac{n}{8}}}{(\frac{n}{8})!} \cdot 2^{\frac{n}{2}} + \ldots + 1$ and show that

$\frac{n^{\frac{n}{8}}}{(\frac{n}{8})!} \cdot 2^{\frac{n}{2}} \leq 2^2 \cdot 2^4 \cdot 2^6 \cdot \ldots \cdot 2^{\frac{n}{2}-2} \cdot 2^{\frac{n}{2}}$

In order to show that, it is sufficient to show that $\frac{n^{\frac{n}{8}}}{(\frac{n}{8})!} \leq 2^2 \cdot 2^4 \cdot 2^6 \cdot \ldots \cdot 2^{\frac{n}{2}-2}$

Let us consider the series:

$$
\begin{aligned}
&2^2 \cdot 2^4 \cdot 2^6 \cdot \ldots \cdot 2^{\frac{n}{2}-2} \\
&= 2^{2+4+6+\ldots+(\frac{n}{2}-2)} \\
&= 2^{\frac{(\frac{n}{2}-2)}{2}[2\cdot 2+(\frac{n}{2}-2-1)\cdot 2]}, \text{substituting sum of the A.P. series } 2+4+6+\ldots+(\frac{n}{2}-2) \\
&= 2^{\frac{(n-4)(n-2)}{4}}
\end{aligned}
$$

Now, it is easy to see that $\frac{n^{\frac{n}{8}}}{(\frac{n}{8})!} \leq 2^{\frac{(n-4)(n-2)}{4}}, \forall n(\geq 8) \in \mathbb{N}_+$. Since there are $(\frac{n}{4}-1)$ terms, and it can easily be seen that each of them are $\leq 2^2 \cdot 2^4 \cdot 2^6 \cdot \ldots \cdot 2^{\frac{n}{2}}$, hence it is multiplied by $(\frac{n}{4}-1)$.

Using a similar calculation, it can be shown that:

$$
\begin{aligned}
N_{h-3} &\leq (\frac{n}{8}-1) \cdot [2^2 \cdot 2^4 \cdot 2^6 \cdot \ldots \cdot 2^{\frac{n}{2}}] + 1 \\
&\vdots \\
N_2 &\leq (4-1) \cdot [2^2 \cdot 2^4 \cdot 2^6 \cdot \ldots \cdot 2^{\frac{n}{2}}] + 1 \\
N_1 &\leq (2-1) \cdot [2^2 \cdot 2^4 \cdot 2^6 \cdot \ldots \cdot 2^{\frac{n}{2}}] + 1
\end{aligned}
$$

Thus from A1 we have

$$f(n) \leq (1+3+7+15+\ldots+(\frac{n}{2}-1)) \cdot [2^2 \cdot 2^4 \cdot 2^6 \cdot \ldots \cdot 2^{\frac{n}{2}}] + \log_2 n + 1 \qquad \text{(A2)}$$

In A2, the additional $\log_2 n + 1$ comes from $\underbrace{1+1+\ldots+1}_{h+1 \text{ times}} = h+1 = \log_2 n + 1$. Now, the sum of the series $1+3+7+15+\ldots+(\frac{n}{2}-1)$ can be obtained from the following calculation:

$$\begin{aligned}
&1+3+7+15+\ldots+(\frac{n}{2}-1) \\
&= (2^0-1)+(2^1-1)+(2^2-1)+(2^3-1)+\ldots+(2^{h-1}-1), n = 2^{h-1} \\
&= (2^0+2^1+2^2+2^3+\ldots+2^{h-1}) - h \\
&= \frac{2^{h-1+1}-1}{2-1} - h, \text{substituting the sum of the geometric series } 2^0+2^1+2^2+2^3+\ldots+2^{h-1} \\
&= (2^h-1)-h \\
&= (n-1)-\log_2 n, \text{substituting } 2^h = n \text{ and } h = \log_2 n \\
&= n - \log_2 n - 1
\end{aligned}$$

Similarly, the product series $2^2 \cdot 2^4 \cdot 2^6 \cdot \ldots \cdot 2^{\frac{n}{2}}$ can be simplified as follows:

$$\begin{aligned}
&2^2 \cdot 2^4 \cdot 2^6 \cdot \ldots \cdot 2^{\frac{n}{2}} \\
&= 2^{2+4+6+\ldots+2^{\frac{n}{2}}} \\
&= 2^{\frac{\frac{n}{2}}{2}[2\cdot 2+(\frac{n}{2}-1)\cdot 2]} \\
&= 2^{\frac{n(n+2)}{4}}
\end{aligned}$$

Therefore, from A2 we get

$$\begin{aligned}
&f(n) \leq (1+3+7+15+\ldots+(\frac{n}{2}-1)) \cdot [2^2 \cdot 2^4 \cdot 2^6 \cdot \ldots \cdot 2^{\frac{n}{2}}] + \log_2 n + 1 \\
&= (n - \log_2 n - 1) \cdot 2^{\frac{n(n+2)}{4}} + \log_2 n + 1 \\
&\leq n \cdot 2^{\frac{n(n+2)}{4}}, \forall\, n \in \mathbb{N}_+
\end{aligned}$$

Therefore, there exists a function $g : \mathbb{N}_+ \to \mathbb{N}$ on $\mathbb{N}_+$ defined by $g(n) = n \cdot 2^{\frac{n(n+2)}{4}}$, and an $n_0 \in \mathbb{N}_+$ such that $0 \leq f(n) \leq g(n)$ for all $n \geq n_0$. Hence, the function $f(n)$ has order $O(n \cdot 2^{\frac{n(n+2)}{4}})$. □

B Proof of theorem 5.5.3

Proof. Let $\{X_1^h, X_2^h, \ldots, X_{n_h}^h\}$ be the set of leaf nodes of the tree for $n_h \geq 2$. Then the branching factor of the initial state of the state space is $\frac{n}{2}$. This is because the agent

can fire $\frac{n}{2}$ rules non-deterministically. At the next level, the branching factor of a state of the state space is $(\frac{n}{2}-1)$. At the next level it is less than or equal to $(\frac{n}{2}-1)$ and so on. The search space expands until it reaches the root node (goal state) and the branching factor reaches its minimal value to one. Hence the worst case branching factor of the state space is of order $O(\frac{n}{2})$. □

C Proof of theorem 5.5.4

$Proof$. Without loss of generality, we assume that the tree is a perfect binary tree. Since the tree has n leaf nodes, it has total $2n-1$ nodes. We consider the following cases:

distribution of leaves $(\frac{n}{2}, \frac{n}{2})$: Let us assume that the leaf facts are distributed between the agents as $X^h_1, X^h_2, \ldots, X^h_{\frac{n}{2}}$ to agent 1 and $X^h_{\frac{n}{2}+1}, X^h_{\frac{n}{2}+2}, \ldots, X^h_{n_h}$ to agent 2. Then at the initial state, each agent can perform $\frac{n}{4}$ rule firing actions, $\frac{n}{2}$ copy actions (in order to copy $\frac{n}{2}$ facts from other agent's memory), and they can be idle. Thus, the total number of non-deterministic actions that can be performed in this state by each agent is $(\frac{n}{4}+\frac{n}{2}+1)$. Therefore, the branching factor at this state is $(\frac{n}{4}+\frac{n}{2}+1)\times(\frac{n}{4}+\frac{n}{2}+1)$ which is of order $O(n^2)$. $O(n^2)$ is the maximal branching factor of a given state of the state space. Because the maximal number of actions that an agent can perform in a given state is always less than n.

distribution of leaves $(even, odd)$: Let us assume that the leaf facts are distributed between the agents as $X^h_2, X^h_4, \ldots, X^h_{n_h}$ to agent 1 and $X^h_1, X^h_3, \ldots, X^h_{n_h-1}$ to agent 2. Then at the initial state, each agent can perform $\frac{n}{2}$ copy actions (in order to copy $\frac{n}{2}$ facts from other agent's memory), and they can be idle. Therefore, the branching factor at this state is $(\frac{n}{2}+1)\times(\frac{n}{2}+1)$ which is of order $O(n^2)$. $O(n^2)$ is the maximal branching factor of a given state of the state space. Because the maximal number of actions that an agent can perform in a given state is always less than n.

distribution of leaves $(n-k, k)$: Let us assume that the leaf facts are distributed between the agents as $(n-k)$ leaf facts to agent 1 and k leaf fact(s) to agent 2 for some

$k \geq 1$. If the leaf facts distribution $(n-k, k)$ between the agents is equivalent to $(\frac{n}{2}, \frac{n}{2})$ or $(even, odd)$ then as in the above cases, the worst case branching factor of the state space is of order $O(n^2)$. Now, if $k = 1$, then at the initial state agent 1 can perform $(\frac{n}{2} - 1)$ rule firing actions, one copy action (in order to copy one fact from other agent's memory), and an idle action. Similarly, agent 2 can perform $(n-1)$ copy actions (in order to copy $(n-1)$ facts from other agent's memory), and an idle action. Therefore, the branching factor at this state is $(\frac{n}{2} - 1 + 1 + 1) \times (n - 1 + 1)$ i.e., $(\frac{n}{2} + 1) \times (n)$ which is of order $O(n^2)$. $O(n^2)$ is the maximal branching factor of a given state of the state space. Because the maximal number of actions that an agent can perform in a given state is always less than n.

Therefore, in a multi-agent rule-based system consisting of two agents which share the same set of rules of an 'n leaf example', the worst case branching factor of the search space is of order $O(n^2)$. □

D Mocha positional encoding

```
– Positional Mocha encoding for a single agent two variable tree resolution.

    – The size (maximum number of literals) in a clause

    #define NUM_LITERALS 2

    – The maximum number of clause cells that can be used in a proof

    #define MAX_CELLS 3

    – Type representing the postive and negative literals in a clause cell

    type literals : bitvector $NUM_LITERALS

    –Agent clauses

    module Agent

    – Whether we have found a proof

    interface proof : bool

    – The positive and negative literals in each clause cell

    interface a1cell0_pos : literals
```

```
interface a1cell0_neg : literals

interface a1cell1_pos : literals

interface a1cell1_neg : literals

interface a1cell2_pos : literals

interface a1cell2_neg : literals

– Whether each clause cell has been allocated. It is more convenient to

– represent this as a bitvector rather than an "array cells of bool",

– though we need to ensure that the size of the allocated bitvector and

– the cells type agree.

private a1_allocated : array (0 .. $MAX_CELLS - 1) of bool

– private allocated : bitvector $MAX_CELLS

– The postive and negative literals of the new cluase at this cycle.

– This may either be the resolvent of two literals currently in memory

– (if any resolve) or a clause read from the KB

private clause_pos : literals

private clause_neg : literals

atom Clause

controls clause_pos, clause_neg,

reads a1cell0_pos, a1cell0_neg, a1cell1_pos, a1cell1_neg, a1cell2_pos, a1cell2_neg

init

[ ] true -> clause_pos' := 0; clause_neg' := 0

update

– For each literal in each pair of cells, check to see if they

– resolve. Note that we don't have to check whether cells are

– allocated as only allocated cells can have non-zero contents.

– Resolve on the first literal

[ ] (a1cell0_pos[0] & a1cell1_neg[0]) -> clause_pos' := ((a1cell0_pos & 2) | a1cell1_pos); clause_neg' := (a1cell0_neg
| (a1cell1_neg & 2))
```

[ ] (a1cell0_neg[0] & a1cell1_pos[0]) -> clause_pos' := (a1cell0_pos | (a1cell1_pos & 2)); clause_neg' := ((a1cell0_neg & 2) | a1cell1_neg)

[ ] (a1cell0_pos[0] & a1cell2_neg[0]) -> clause_pos' := ((a1cell0_pos & 2) | a1cell2_pos); clause_neg' := (a1cell0_neg | (a1cell2_neg & 2))

[ ] (a1cell0_neg[0] & a1cell2_pos[0]) -> clause_pos' := (a1cell0_pos | (a1cell2_pos & 2)); clause_neg' := ((a1cell0_neg & 2) | a1cell2_neg)

[ ] (a1cell1_pos[0] & a1cell2_neg[0]) -> clause_pos' := ((a1cell1_pos & 2) | a1cell2_pos); clause_neg' := (a1cell1_neg | (a1cell2_neg & 2))

[ ] (a1cell1_neg[0] & a1cell2_pos[0]) -> clause_pos' := (a1cell1_pos | (a1cell2_pos & 2)); clause_neg' := ((a1cell1_neg & 2) | a1cell2_neg)

– Resolve on the second literal

[ ] (a1cell0_pos[1] & a1cell1_neg[1]) -> clause_pos' := ((a1cell0_pos & 1) | a1cell1_pos); clause_neg' := (a1cell0_neg | (a1cell1_neg & 1))

[ ] (a1cell0_neg[1] & a1cell1_pos[1]) -> clause_pos' := (a1cell0_pos | (a1cell1_pos & 1)); clause_neg' := ((a1cell0_neg & 1) | a1cell1_neg)

[ ] (a1cell0_pos[1] & a1cell2_neg[1]) -> clause_pos' := ((a1cell0_pos & 1) | a1cell2_pos); clause_neg' := (a1cell0_neg | (a1cell2_neg & 1))

[ ] (a1cell0_neg[1] & a1cell2_pos[1]) -> clause_pos' := (a1cell0_pos | (a1cell2_pos & 1)); clause_neg' := ((a1cell0_neg & 1) | a1cell2_neg)

[ ] (a1cell1_pos[1] & a1cell2_neg[1]) -> clause_pos' := ((a1cell1_pos & 1) | a1cell2_pos); clause_neg' := (a1cell1_neg | (a1cell2_neg & 1))

[ ] (a1cell1_neg[1] & a1cell2_pos[1]) -> clause_pos' := (a1cell1_pos | (a1cell2_pos & 1)); clause_neg' := ((a1cell1_neg & 1) | a1cell2_neg)

– Alternatively, we can read the new value from the KB

– We have two variables, A1 and A2 with indices 0, and 1.

– A1 v A2

[ ] true -> clause_pos'[0] := true; clause_pos'[1] := true; clause_neg'[0] := false; clause_neg'[1] := false

– A1 v A2

[ ] true -> clause_pos'[0] := false; clause_pos'[1] := true; clause_neg'[0] := true; clause_neg'[1] := false

```
– A1 v  A2
[ ] true -> clause_pos'[0] := true; clause_pos'[1] := false; clause_neg'[0] := false; clause_neg'[1] := true
–  A1 v  A2
[ ] true -> clause_pos'[0] := false; clause_pos'[1] := false; clause_neg'[0] := true; clause_neg'[1] := true
endatom
atom Overwrite
controls a1cell0_pos, a1cell0_neg, a1cell1_pos, a1cell1_neg, a1cell2_pos, a1cell2_neg, a1_allocated
reads a1cell0_pos, a1cell0_neg, a1cell1_pos, a1cell1_neg, a1cell2_pos, a1cell2_neg, a1_allocated
awaits clause_pos, clause_neg
init
[ ] true ->
a1cell0_pos' := 0; a1cell0_neg' := 0;
a1cell1_pos' := 0; a1cell1_neg' := 0;
a1cell2_pos' := 0; a1cell2_neg' := 0;
forall j a1_allocated'[j] := false
update
[ ] true -> a1cell0_pos' := clause_pos'; a1cell0_neg' := clause_neg'; a1_allocated'[0] := true
[ ] true -> a1cell1_pos' := clause_pos'; a1cell1_neg' := clause_neg'; a1_allocated'[1] := true
[ ] true -> a1cell2_pos' := clause_pos'; a1cell2_neg' := clause_neg'; a1_allocated'[2] := true
endatom
atom Proof
controls proof
reads proof
awaits a1cell0_pos, a1cell0_neg, a1cell1_pos, a1cell1_neg, a1cell2_pos, a1cell2_neg, a1_allocated
init
[ ] true -> proof' := false
update
– We have found a proof if we have an allocated cell containing
– no positive or negative literals
```

```
[ ] a1_allocated'[0] & ((a1cell0_pos' | a1cell0_neg') = 0) -> proof' := true

[ ] a1_allocated'[1] & ((a1cell1_pos' | a1cell1_neg') = 0) -> proof' := true

[ ] a1_allocated'[2] & ((a1cell2_pos' | a1cell2_neg') = 0) -> proof' := true

endatom

endmodule
```

E Mocha non-positional encoding

```
– Non-positional Mocha encoding for a single agent two variable tree resolution.

– The maximum number of clause cells that can be used in a proof

#define MAX_CELLS 3

–Agent1 clauses

module Agent1

– Whether we have found a proof

interface phi : bool

– The possible clauses

interface AvB, nAvB, AvnB, nAvnB, A, nA, B, nB : bool

– The number of clauses in memory

interface count : (0 .. $MAX_CELLS)

– Events

private add_AvB, add_nAvB, add_AvnB, add_nAvnB, add_A, add_nA, add_B, add_nB, add_phi, new_clause :
event

private overwrite_AvB, overwrite_nAvB, overwrite_AvnB, overwrite_nAvnB, overwrite_A, overwrite_nA, overwrite_B,
overwrite_nB : event

atom Clause

controls add_AvB, add_nAvB, add_AvnB, add_nAvnB, add_A, add_nA, add_B, add_nB, add_phi, new_clause

reads AvB, nAvB, AvnB, nAvnB, A, nA, B, nB, phi, add_AvB, add_nAvB, add_AvnB, add_nAvnB, add_A, add_nA,
add_B, add_nB, add_phi, new_clause

update
```

– Clauses can only be read if they are not already in memory.

[ ] ∼AvB -> add_AvB!; new_clause!

[ ] ∼nAvB -> add_nAvB!; new_clause!

[ ] ∼AvnB -> add_AvnB!; new_clause!

[ ] ∼nAvnB -> add_nAvnB!; new_clause!

– Clauses can only resolve if their resolvent is not in memory. Note that we do not allow resolution to produce tautologies.

– Copying from other agents can be done with additional guards in the same way.

[ ] ∼B & AvB & nAvB -> add_B!; new_clause!

[ ] ∼B & AvB & nA -> add_B!; new_clause!

[ ] ∼B & nAvB & A -> add_B!; new_clause!

[ ] ∼nB & AvnB & nAvnB -> add_nB!; new_clause!

[ ] ∼nB & AvnB & nA -> add_nB!; new_clause!

[ ] ∼nB & nAvnB & A -> add_nB!; new_clause!

[ ] ∼A & AvB & AvnB -> add_A!; new_clause!

[ ] ∼A & AvB & nB -> add_A!; new_clause!

[ ] ∼A & AvnB & B -> add_A!; new_clause!

[ ] ∼nA & nAvB & nAvnB -> add_nA!; new_clause!

[ ] ∼nA & nAvB & nB -> add_nA!; new_clause!

[ ] ∼nA & nAvnB & B -> add_nA!; new_clause!

[ ] ∼phi & A & nA -> add_phi!; new_clause!

[ ] ∼phi & B & nB -> add_phi!; new_clause!

endatom

atom Overwrite

controls count, overwrite_AvB, overwrite_nAvB, overwrite_AvnB, overwrite_nAvnB, overwrite_A, overwrite_nA, overwrite_B, overwrite_nB

reads count, new_clause,AvB, nAvB, AvnB, nAvnB, A, nA, B, nB, overwrite_AvB, overwrite_nAvB, overwrite_AvnB, overwrite_nAvnB,overwrite_A, overwrite_nA, overwrite_B, overwrite_nB

awaits new_clause

init

[ ] true -> count' := 0

update

[ ] count < $MAX_CELLS & new_clause? -> count' := count + 1

[ ] count = $MAX_CELLS & new_clause? & AvB -> overwrite_AvB!

[ ] count = $MAX_CELLS & new_clause? & nAvB -> overwrite_nAvB!

[ ] count = $MAX_CELLS & new_clause? & AvnB -> overwrite_AvnB!

[ ] count = $MAX_CELLS & new_clause? & nAvnB -> overwrite_nAvnB!

[ ] count = $MAX_CELLS & new_clause? & A -> overwrite_A!

[ ] count = $MAX_CELLS & new_clause? & nA -> overwrite_nA!

[ ] count = $MAX_CELLS & new_clause? & B -> overwrite_B!

[ ] count = $MAX_CELLS & new_clause? & nB -> overwrite_nB!

endatom

– Each clause is controlled by its corresponding atom, which waits for the appropriate add and overwrite events.

– Note that a clause can only be added or overwritten at a cycle (not both), and that overwriting does not reduce count,

– since we only overwrite when memory is full.

atom Clause_AvB

controls AvB

reads AvB, add_AvB, overwrite_AvB

awaits add_AvB, overwrite_AvB

init

[ ] true -> AvB' := false

update

[ ] ∼AvB & add_AvB? -> AvB' := true

[ ] AvB & overwrite_AvB? -> AvB' := false

endatom

atom Clause_nAvB

controls nAvB

```
reads nAvB, add_nAvB, overwrite_nAvB

awaits add_nAvB, overwrite_nAvB

init

[ ] true -> nAvB' := false

update

[ ] ∼nAvB & add_nAvB? -> nAvB' := true

[ ] nAvB & overwrite_nAvB? -> nAvB' := false

endatom

atom Clause_AvnB

controls AvnB

reads AvnB, add_AvnB, overwrite_AvnB

awaits add_AvnB, overwrite_AvnB

init

[ ] true -> AvnB' := false

update

[ ] ∼AvnB & add_AvnB? -> AvnB' := true

[ ] AvnB & overwrite_AvnB? -> AvnB' := false

endatom

atom Clause_nAvnB

controls nAvnB

reads nAvnB, add_nAvnB, overwrite_nAvnB

awaits add_nAvnB, overwrite_nAvnB

init

[ ] true -> nAvnB' := false

update

[ ] ∼nAvnB & add_nAvnB? -> nAvnB' := true

[ ] nAvnB & overwrite_nAvnB? -> nAvnB' := false

endatom

atom Clause_A
```

```
controls A

reads A, add_A, overwrite_A

awaits add_A, overwrite_A

init

[ ] true -> A' := false

update

[ ] ∼A & add_A? -> A' := true

[ ] A & overwrite_A? -> A' := false

endatom

atom Clause_nA

controls nA

reads nA, add_nA, overwrite_nA

awaits add_nA, overwrite_nA

init

[ ] true -> nA' := false

update

[ ] ∼nA & add_nA? -> nA' := true

[ ] nA & overwrite_nA? -> nA' := false

endatom

atom Clause_B

controls B

reads B, add_B, overwrite_B

awaits add_B, overwrite_B

init

[ ] true -> B' := false

update

[ ] ∼B & add_B? -> B' := true

[ ] B & overwrite_B? -> B' := false

endatom
```

```
atom Clause_nB
controls nB
reads nB, add_nB, overwrite_nB
awaits add_nB, overwrite_nB
init
[ ] true -> nB' := false
update
[ ] ∼nB & add_nB? -> nB' := true
[ ] nB & overwrite_nB? -> nB' := false
endatom
atom Clause_phi
controls phi
reads phi, add_phi
awaits add_phi
init
[ ] true -> phi' := false
update
[ ] ∼phi & add_phi? -> phi' := true
endatom
endmodule
```

F NuSMV positional encoding

– Positional NuSMV encoding for a single agent two variable tree resolution.

```
MODULE read(cell,x,y,clause)
ASSIGN
next(cell) := case
cell!=clause & x!=clause & y!=clause : clause;
1 : cell;
```

```
esac;
MODULE resolve(proof, cell, x, y, z, i, j)
DEFINE
i_bit := 0b4_0001 << i;
j_bit := 0b4_0001 << j;
i_mask := !i_bit;
j_mask := !j_bit;
v1 := (x & i_mask) | (y & j_mask);
v2 := (x & j_mask) | (y &i_mask);
ASSIGN
next(cell) :=
case
– If we can resolve in either order, it doesn't matter which we do.
cell!=v1 & x!=v1 & y!=v1 & z!=v1 & !bool((v1[0:0] & v1[2:2]) | (v1[1:1] & v1[3:3])) & (((x & i_bit) = i_bit) & ((y &
j_bit) = j_bit)) : v1;
cell!=v2 & x!=v2 & y!=v2 & z!=v2 & !bool((v2[0:0] & v2[2:2]) | (v2[1:1] & v2[3:3])) & (((x & j_bit) = j_bit) & ((y &
i_bit) = i_bit)) : v2;
1 : cell;
esac;
– We have a proof if the cell was allocated (i.e., wasn't empty) and now is
next(proof) := cell != 0b4_0000 & next(cell) = 0b4_0000;
MODULE clause_cell(proof, x, y)
VAR
cell : word[4];
– Read a clause into this clause cell
read0 : process read(cell,x,y, 0b4_1100);
read1 : process read(cell,x,y, 0b4_1001);
read2 : process read(cell,x,y, 0b4_0110);
read3 : process read(cell,x,y, 0b4_0011);
```

```
– Resolve on A1 with the x clause

resolve0 : process resolve(proof, cell, cell, x, y, A1, nA1);

– Resolve on A2 with the x clause

resolve1 : process resolve(proof, cell, cell, x, y, A2, nA2);

– Resolve on A1 with the y clause

resolve2 : process resolve(proof, cell, cell, y, x, A1, nA1);

– Resolve on A2 with the y clause

resolve3 : process resolve(proof, cell, cell, y, x, A2, nA2);

– Resolve on A1 in the x and y clauses

resolve4 : process resolve(proof, cell, x, y, cell, A1, nA1);

– Resolve on A2 in the x and y clauses

resolve5 : process resolve(proof, cell, x, y, cell, A2, nA2);

DEFINE

– Bit 0 codes neg A1 and bit 1 codes neg A2, bit 2 codes A1 and bit 3 A2

A2 := 3;

A1 := 2;

nA2 := 1;

nA1 := 0;

ASSIGN

init(cell) := 0b4_0000;

MODULE main

VAR

proof : boolean;

cell0 : process clause_cell(proof, cell1.cell, cell2.cell);

cell1 : process clause_cell(proof, cell0.cell, cell2.cell);

cell2 : process clause_cell(proof, cell0.cell, cell1.cell);

ASSIGN

init(proof) := 0;

FAIRNESS
```

running

— property to be verified

SPEC AG ! proof

G Rule ordering strategy in an "16 leaf example"

For ease of illustration, we consider "16 leaf example". Table G1 and Table G2 show how we can direct the agents to focus on a particular region of the tree by assigning rule priority. In Table G1, we consider a multi-agent system consisting of two concrete agents. Both agents use the rule ordering reasoning strategy. Agent 1 assigns lower priority to rules in the right-hand shaded triangular region depicted in Figure G1. In contrast, agent 2 assigns lower priority to rules in the left-hand shaded triangular region of Figure G1.

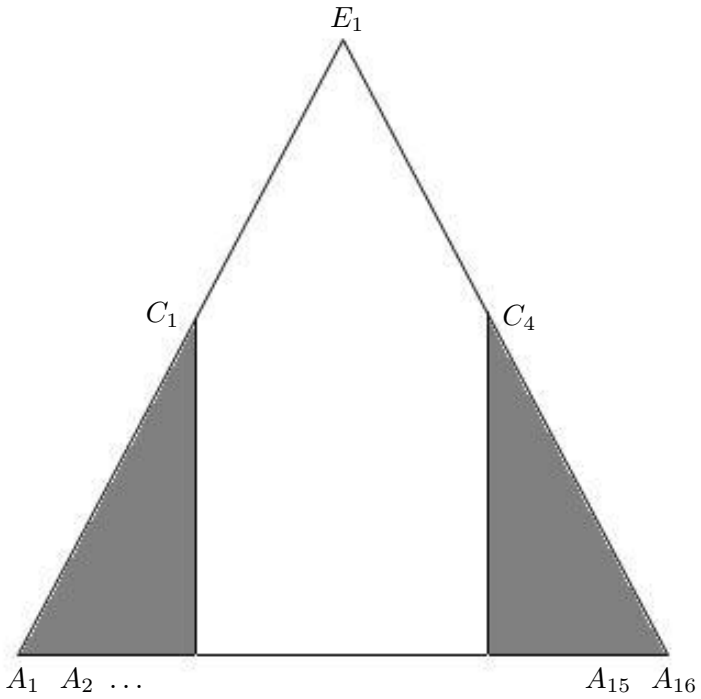


Figure G1: Focus on a particular region of the tree

In Table G2, we consider a multi-agent system consisting of three concrete agents. All the agents use the rule ordering reasoning strategy. In this case, the set of rules in the unshaded region have higher priority for agent 1, the rules in left hand shaded

region have higher priority for agent 2, and the rules in the right hand shaded region have higher priority for agent 3.

Agent 1	Agent 2
Rules	Rules
$\langle\ 15\ :\ A_1(x) \wedge A_2(x) \rightarrow B_1(x)\ \rangle$ $\langle\ 14\ :\ A_3(x) \wedge A_4(x) \rightarrow B_2(x)\ \rangle$ $\langle\ 13\ :\ A_5(x) \wedge A_6(x) \rightarrow B_3(x)\ \rangle$ $\langle\ 12\ :\ A_7(x) \wedge A_8(x) \rightarrow B_4(x)\ \rangle$ $\langle\ 11\ :\ A_9(x) \wedge A_{10}(x) \rightarrow B_5(x)\ \rangle$ $\langle\ 10\ :\ A_{11}(x) \wedge A_{12}(x) \rightarrow B_6(x)\ \rangle$ $\langle\ 3\ :\ A_{13}(x) \wedge A_{14}(x) \rightarrow B_7(x)\ \rangle$ $\langle\ 2\ :\ A_{15}(x) \wedge A_{16}(x) \rightarrow B_8(x)\ \rangle$ $\langle\ 9\ :\ B_1(x) \wedge B_2(x) \rightarrow C_1(x)\ \rangle$ $\langle\ 8\ :\ B_3(x) \wedge B_4(x) \rightarrow C_2(x)\ \rangle$ $\langle\ 7\ :\ B_5(x) \wedge B_6(x) \rightarrow C_3(x)\ \rangle$ $\langle\ 1\ :\ B_7(x) \wedge B_8(x) \rightarrow C_4(x)\ \rangle$ $\langle\ 6\ :\ C_1(x) \wedge C_2(x) \rightarrow D_1(x)\ \rangle$ $\langle\ 5\ :\ C_3(x) \wedge C_4(x) \rightarrow D_2(x)\ \rangle$ $\langle\ 4\ :\ D_1(x) \wedge D_2(x) \rightarrow E_1(x)\ \rangle$ $\langle\ 17\ :\ A_1(x) \rightarrow \mathsf{Ask}(1, 2, C_4(a))\ \rangle$ $\langle\ 16\ :\ \mathsf{Tell}(2, 1, C_4(a)) \rightarrow C_4(a)\ \rangle$	$\langle\ 12\ :\ A_1(x) \wedge A_2(x) \rightarrow B_1(x)\ \rangle$ $\langle\ 11\ :\ A_3(x) \wedge A_4(x) \rightarrow B_2(x)\ \rangle$ $\langle\ 10\ :\ A_5(x) \wedge A_6(x) \rightarrow B_3(x)\ \rangle$ $\langle\ 9\ :\ A_7(x) \wedge A_8(x) \rightarrow B_4(x)\ \rangle$ $\langle\ 8\ :\ A_9(x) \wedge A_{10}(x) \rightarrow B_5(x)\ \rangle$ $\langle\ 7\ :\ A_{11}(x) \wedge A_{12}(x) \rightarrow B_6(x)\ \rangle$ $\langle\ 15\ :\ A_{13}(x) \wedge A_{14}(x) \rightarrow B_7(x)\ \rangle$ $\langle\ 14\ :\ A_{15}(x) \wedge A_{16}(x) \rightarrow B_8(x)\ \rangle$ $\langle\ 6\ :\ B_1(x) \wedge B_2(x) \rightarrow C_1(x)\ \rangle$ $\langle\ 5\ :\ B_3(x) \wedge B_4(x) \rightarrow C_2(x)\ \rangle$ $\langle\ 4\ :\ B_5(x) \wedge B_6(x) \rightarrow C_3(x)\ \rangle$ $\langle\ 13\ :\ B_7(x) \wedge B_8(x) \rightarrow C_4(x)\ \rangle$ $\langle\ 3\ :\ C_1(x) \wedge C_2(x) \rightarrow D_1(x)\ \rangle$ $\langle\ 2\ :\ C_3(x) \wedge C_4(x) \rightarrow D_2(x)\ \rangle$ $\langle\ 1\ :\ D_1(x) \wedge D_2(x) \rightarrow E_1(x)\ \rangle$ $\langle\ 16\ :\ \mathsf{Ask}(1, 2, C_4(a)) \wedge C_4(a)$ $\rightarrow \mathsf{Tell}(2, 1, C_4(a))\ \rangle$
Initial WM facts	Initial WM facts
$\{A_1(a), A_2(a), \ldots, A_{16}(a)\}$	$\{A_1(a), A_2(a), \ldots, A_{16}(a)\}$

Table G1: Two agents "16 leaf example"

Agent 1	Agent 2	Agent 3
Rules	Rules	Rules
$\langle\ 6\ :\ A_1(x) \wedge A_2(x) \rightarrow B_1(x)\ \rangle$ $\langle\ 5\ :\ A_3(x) \wedge A_4(x) \rightarrow B_2(x)\ \rangle$ $\langle\ 15\ :\ A_5(x) \wedge A_6(x) \rightarrow B_3(x)\ \rangle$ $\langle\ 14\ :\ A_7(x) \wedge A_8(x) \rightarrow B_4(x)\ \rangle$ $\langle\ 13\ :\ A_9(x) \wedge A_{10}(x) \rightarrow B_5(x)\ \rangle$ $\langle\ 12\ :\ A_{11}(x) \wedge A_{12}(x) \rightarrow B_6(x)\ \rangle$ $\langle\ 4\ :\ A_{13}(x) \wedge A_{14}(x) \rightarrow B_7(x)\ \rangle$ $\langle\ 3\ :\ A_{15}(x) \wedge A_{16}(x) \rightarrow B_8(x)\ \rangle$ $\langle\ 2\ :\ B_1(x) \wedge B_2(x) \rightarrow C_1(x)\ \rangle$ $\langle\ 11\ :\ B_3(x) \wedge B_4(x) \rightarrow C_2(x)\ \rangle$ $\langle\ 10\ :\ B_5(x) \wedge B_6(x) \rightarrow C_3(x)\ \rangle$ $\langle\ 1\ :\ B_7(x) \wedge B_8(x) \rightarrow C_4(x)\ \rangle$ $\langle\ 8\ :\ C_1(x) \wedge C_2(x) \rightarrow D_1(x)\ \rangle$ $\langle\ 9\ :\ C_3(x) \wedge C_4(x) \rightarrow D_2(x)\ \rangle$ $\langle\ 7\ :\ D_1(x) \wedge D_2(x) \rightarrow E_1(x)\ \rangle$ $\langle\ 19\ :\ A_1(x) \rightarrow \mathsf{Ask}(1, 2, C_1(a))\ \rangle$ $\langle\ 18\ :\ A_2(x) \rightarrow \mathsf{Ask}(1, 3, C_4(a))\ \rangle$ $\langle\ 17\ :\ \mathsf{Tell}(2, 1, C_1(a)) \rightarrow C_1(a)\ \rangle$ $\langle\ 16\ :\ \mathsf{Tell}(3, 1, C_4(a)) \rightarrow C_4(a)\ \rangle$	$\langle\ 15\ :\ A_1(x) \wedge A_2(x) \rightarrow B_1(x)\ \rangle$ $\langle\ 14\ :\ A_3(x) \wedge A_4(x) \rightarrow B_2(x)\ \rangle$ $\langle\ 12\ :\ A_5(x) \wedge A_6(x) \rightarrow B_3(x)\ \rangle$ $\langle\ 11\ :\ A_7(x) \wedge A_8(x) \rightarrow B_4(x)\ \rangle$ $\langle\ 10\ :\ A_9(x) \wedge A_{10}(x) \rightarrow B_5(x)\ \rangle$ $\langle\ 9\ :\ A_{11}(x) \wedge A_{12}(x) \rightarrow B_6(x)\ \rangle$ $\langle\ 8\ :\ A_{13}(x) \wedge A_{14}(x) \rightarrow B_7(x)\ \rangle$ $\langle\ 7\ :\ A_{15}(x) \wedge A_{16}(x) \rightarrow B_8(x)\ \rangle$ $\langle\ 13\ :\ B_1(x) \wedge B_2(x) \rightarrow C_1(x)\ \rangle$ $\langle\ 6\ :\ B_3(x) \wedge B_4(x) \rightarrow C_2(x)\ \rangle$ $\langle\ 5\ :\ B_5(x) \wedge B_6(x) \rightarrow C_3(x)\ \rangle$ $\langle\ 4\ :\ B_7(x) \wedge B_8(x) \rightarrow C_4(x)\ \rangle$ $\langle\ 3\ :\ C_1(x) \wedge C_2(x) \rightarrow D_1(x)\ \rangle$ $\langle\ 2\ :\ C_3(x) \wedge C_4(x) \rightarrow D_2(x)\ \rangle$ $\langle\ 1\ :\ D_1(x) \wedge D_2(x) \rightarrow E_1(x)\ \rangle$ $\langle\ 16\ :\ \mathsf{Ask}(1, 2, C_1(a)) \wedge C_1(a)$ $\rightarrow \mathsf{Tell}(2, 1, C_1(a))\ \rangle$	$\langle\ 12\ :\ A_1(x) \wedge A_2(x) \rightarrow B_1(x)\ \rangle$ $\langle\ 11\ :\ A_3(x) \wedge A_4(x) \rightarrow B_2(x)\ \rangle$ $\langle\ 10\ :\ A_5(x) \wedge A_6(x) \rightarrow B_3(x)\ \rangle$ $\langle\ 9\ :\ A_7(x) \wedge A_8(x) \rightarrow B_4(x)\ \rangle$ $\langle\ 8\ :\ A_9(x) \wedge A_{10}(x) \rightarrow B_5(x)\ \rangle$ $\langle\ 7\ :\ A_{11}(x) \wedge A_{12}(x) \rightarrow B_6(x)\ \rangle$ $\langle\ 15\ :\ A_{13}(x) \wedge A_{14}(x) \rightarrow B_7(x)\ \rangle$ $\langle\ 14\ :\ A_{15}(x) \wedge A_{16}(x) \rightarrow B_8(x)\ \rangle$ $\langle\ 6\ :\ B_1(x) \wedge B_2(x) \rightarrow C_1(x)\ \rangle$ $\langle\ 5\ :\ B_3(x) \wedge B_4(x) \rightarrow C_2(x)\ \rangle$ $\langle\ 4\ :\ B_5(x) \wedge B_6(x) \rightarrow C_3(x)\ \rangle$ $\langle\ 13\ :\ B_7(x) \wedge B_8(x) \rightarrow C_4(x)\ \rangle$ $\langle\ 3\ :\ C_1(x) \wedge C_2(x) \rightarrow D_1(x)\ \rangle$ $\langle\ 2\ :\ C_3(x) \wedge C_4(x) \rightarrow D_2(x)\ \rangle$ $\langle\ 1\ :\ D_1(x) \wedge D_2(x) \rightarrow E_1(x)\ \rangle$ $\langle\ 16\ :\ \mathsf{Ask}(1, 3, C_4(a)) \wedge C_4(a)$ $\rightarrow \mathsf{Tell}(3, 1, C_4(a))\ \rangle$
Initial WM facts	Initial WM facts	Initial WM facts
$\{A_1(a), A_2(a), \ldots, A_{16}(a)\}$	$\{A_1(a), A_2(a), \ldots, A_{16}(a)\}$	$\{A_1(a), A_2(a), \ldots, A_{16}(a)\}$

Table G2: Three agents "16 leaf example"