University of Groningen

Master Thesis

A Model Checker for Unbounded Dynamic Epistemic Logic Models

Author:
Jorge Medina

Supervisors:
Prof. dr. Barteld Kooi
Prof. dr. Rineke Verbrugge

A thesis submitted in fulfilment of the requirements for the degree of Master of Artificial Intelligence

in the

Institute of Artificial Intelligence

August 2013

“Nobody told me it was impossible, so I did it!”

Jean Cocteau

UNIVERSITY OF GRONINGEN

Abstract

Faculty of Mathematics and Natural Sciences
Institute of Artificial Intelligence

Master of Artificial Intelligence

A Model Checker for Unbounded Dynamic Epistemic Logic Models

by Jorge MEDINA

Model checking is a relatively new technology. It has been here for around 20 years and it has moved from pure research to practical application in the industry. The application of model checking can enhance existing validation techniques such as simulation and testing.

The model checker expresses the system specifications as logical formulas. Symbolic algorithms are used to analyse the model defined by the system. These algorithms check if the given specification holds in the system or not, in other words, given a model, the model checker tries to exhaustively and automatically check whether this model meets a given specification.

The aim of this project is to create an intuitive model checker capable of checking epistemic formulas in unbounded models. Models may form an infinite collection of finite models which share similar characteristics and patterns since they are created using a regular expression. There may not be a fixed size of the individual models in the collection. Although many model checkers have been created for epistemic logic, no one was developed so far that could handle this kind of unbounded classes models.

The model checker that we created cn be used to analyse a collection of unbounded models represented by a regular expression and show a possible model in which a formula can hold. As unbounded collections of models are created using a regular expression and regular expressions denote regular languages which represent strings, we start by only investigating unbounded models that are limited to a string-like form. Because of this, we chose the simplest form of models, namely linear models. The project is focused only on the domain of the epistemic logic $S5$, since it is considered one of the most well-known epistemic logics.

Epistemic logic is known as the logic of knowledge. It provides information about the properties of individual agents and also tools to model complicated scenarios which involve two or more agents and helps to improve the understanding of the dynamics of knowledge.

Most epistemic logics are propositional modal logics. Logicians have found ways to formally handle a wide variety of knowledge claims in propositional terms.

In order to analyse unbounded models, the model checker was built in four phases. Each one handles a different topic in epistemic logic. These phases were built in such

a way that the user feels more comfortable when inserting the formulas in the model checker as if he was writing them on paper.

The first module handles the epistemic logic $S5$. The theory of computation has focused its attention on the system $S5$ in order to be able to represent knowledge and process information by robotic systems and other agents.

The second module handles public announcements that can be pronounced by an agent inside the model or by an agent that is not involved in it. It is possible that this announcement can help the rest of the agents to increase their own knowledge.

The model checker implements the logic of public announcement in two ways: you can either submit a formula with the public announcement operator or you can announce a formula in a model so the model will be updated.

The third module analyses unbounded models. These are models which will follow a specific pattern capable of being represented with a regular expression. This means that the models are somehow dynamic and may grow indefinitely depending on the regular expression used to define them.

The fourth and final module deals with common knowledge, a special kind of knowledge for a group of agents. Common knowledge is useful for agents since it allows them to communicate or coordinate their behaviour. In order for the agents to have a successful interaction, they require mutual understanding or background knowledge.

These four phases or modules depend on each other since each module uses functions defined in external modules. With this approach, models that at first sight may have required a big input, can be defined using a single line.

The unbounded models module allows us to obtain a representative model of a regular expression. This obtained model, is the result of running an epistemic formula which has a special meaning for our system.

The model checking technique has been applied in a number of real-life applications, especially in hardware verification and to prove correctness of a variety of software protocols like the two generals problem described in this document.

This project in particular, is using model checking as a simulation for logic puzzles and research purposes. Model checking is a powerful extension of the traditional verification process, and we must consider it as a complement to simulation and emulation.

Contents

Abstract **ii**

1 Introduction **1**
1.1 Epistemic Logic . . . 1
1.2 Model Checking . . . 2
1.3 Model Checkers and Logic Puzzles . . . 4

2 The Model Checker **5**
2.1 Module 1: Epistemic Logic S5. . . . 5
2.2 Module 2: Public Announcements. . . . 6
2.3 Module 3: Unbounded Models. . . . 8
2.4 Module 4: Common Knowledge. . . . 8

3 The Model **10**
3.1 Valid Characters in Formulas . . . 10
3.2 Model Checker Functions . . . 11
3.2.1 Use of Functions . . . 11
3.2.1.1 formula . . . 11
3.2.1.2 model . . . 12
3.2.1.3 announce . . . 12
3.2.1.4 check . . . 13
3.2.1.5 print . . . 13
3.2.1.6 delete . . . 13
3.2.1.7 longmodel . . . 14
3.2.1.8 longformula . . . 15
3.2.1.9 regex . . . 15

4 Model Checker for S5 **16**

5 Public Announcements **19**
5.1 The wise men . . . 19
5.2 Muddy children . . . 21
5.3 Sum and product . . . 24

6 Unbounded Models **26**
6.1 Alphabet . . . 26
6.2 Strings . . . 27

6.2.1 Powers of an Alphabet: 27
6.2.2 Model checker strings 27
6.2.3 Languages 28
6.3 Regular Expressions 28
6.3.1 Priority of operators. 30
6.4 Building Regular Expressions 30
6.5 Two Generals Problem 32
6.6 The *regex* function. 34

7 Common Knowledge 38
7.1 The Language of Common Knowledge 38
7.2 Semantics of Common Knowledge 39
7.3 Public Announcements and Common Knowledge 39
7.4 Common Knowledge in the Model Checker 42

8 Examples 47
8.1 Example 1 47
8.2 Example 2 49
8.3 Example 3 50

9 Conclusions and Future Work 52
9.1 Conclusions 52
9.2 Future Work 53

A The Interpreter 55
A.1 Lexicon analyser 55
A.2 Syntactic analyser 56
A.3 Semantic analyser 57

B Installation 59
B.1 Installation in Windows 59
B.1.1 Steps for installing Java 59
B.1.2 Installing Tomcat 59
B.1.3 Setting the environment variables 60
B.1.4 Deploying the Model Checker 60
B.2 Installation on Linux 60
B.2.1 Steps for installing Java 60
B.2.2 Installing Tomcat 61
B.2.3 Setting the environment variables 61
B.2.4 Deploying the Model Checker 61

Bibliography 62

Chapter 1

Introduction

This project tries to apply epistemic logic practically by modelling puzzles and real life logic problems. The project is built in four phases. Each phase handles a different topic on epistemic logic. These four modules are:

1. A module to check for epistemic logic $S5_m$.
2. A module for public announcement logic.
3. A module for modelling unbounded models.
4. A module for common knowledge.

The project is developed in this order because the final phases require the previous ones to work correctly. It also makes the modelling more intuitive. This Msc thesis will describe the development of each one of the phases, showing some examples of the application and the correct use of the program.

Before starting to describe the project, a brief introduction to epistemic logic and model checking will be given. This introduction includes a definition of epistemic logic as well as the description of syntax and semantics. Here, the concept of "knowledge" will be explained, which is the core of the epistemic logic.

Section 1.2 a background on model checking is presented. Some advantages and disadvantages of model checking are analysed. It also includes a few examples of model checkers that have proven to be useful to model logic puzzles.

Section 1.3 presents some of the puzzles this project will be able to model, as well as a description of how other existing model checkers are able to handle them.

1.1 Epistemic Logic

The epistemic logic is known as the logic of 'knowledge'. This logic provides information about the properties of individual agents and tools to model complicated scenarios involving two or more agents and it is helping to improve the understanding of the dynamic of knowledge.

By studying epistemic logic, logicians have found that expressions like "knows that" contain semantic properties. Epistemic logic is not only useful to solve traditional philosophical problems, it also has many applications in computer science and economics. Examples of these applications are robotics, network security and cryptography applications.

Most of epistemic logic focuses on propositional knowledge. Logicians have found ways to formally handle a wide variety of knowledge claims in propositional terms. The language of epistemic logic can be considered the same as the language of propositional logic with the addition of the epistemic operator K_i such that

$$K_i\varphi \text{ reads "Agent } i \text{ knows } \varphi\text{".}$$

The set of states accessible to an agent depends on the information it knows at that instant. In order to understand the dependency between the states, it is necessary to introduce the accessibility relation, R_i.

This relation is usually written $R_i(s,t)$ and reads "state t is accessible from s". The state t is said to be an epistemic alternative to state s for agent i. With this semantical interpretation it is possible to conclude that, if a formula φ is true in all the states which an agent considers possible, then the agents knows φ.

A Kripke-model is defined as $M = \langle S, R_i, ...R_m, \pi \rangle$ where S is a non-empty set of states, $R_i, ...R_m$, are the accessibility relations and π is a truth assignment to the propositional atoms per state. The resulting semantics of these models are called Kripke-semantics [21]. An atomic propositional formula, p, is said to be true in a state s in a model M $(M, s) \vDash p$ iff s is in the set of possible states assigned to p.

Lemmon developed a helpful nomenclature [18] to catalogue the axioms for epistemic logic. Combining these axioms may result in the derivation of multiple logic systems. This project will only focus on the epistemic system $S5$ for multiple agents ($S5_m$).

1.2 Model Checking

Model checking is defined as a technique to formally verify finite state concurrent systems. The model checker expresses the system specifications as logic formulas. Symbolic algorithms are used to analyse the model defined by the system. These algorithms will check if the given specification holds in the system or not.

The first works on model checking were made by two pairs in the early eighties: Clarke and Emerson [5] and Queille and Sifakis [23].

Baier defines model checking as an automated technique that, given a finite-state model of a system and a formal property, systematically checks whether this property holds for that model [14].

Baier and Katoen mention in the book [2, Chapter 1] that the design of a model checker should include three different phases .

- *Modelling phase*

- *Running phase*
- *Analysis phase*

As every system or approach to problem solving, model checking techniques also have strengths and weaknesses. Some of the most relevant strengths mentioned by Baier and Katoen in [2, Chapter 1] are:

- It is a general verification approach that is applicable to a wide range of applications such as embedded systems, software engineering, and hardware design.
- It supports partial verification, i.e., properties can be checked individually, thus allowing focus on the essential properties first. No complete requirement specification is needed.
- It is not vulnerable to the likelihood that an error is exposed;
- It provides diagnostic information in case a property is invalidated; this is very useful for debugging purposes.
- It has a sound and mathematical underpinning; it is based on theory of graph algorithms, data structures, and logic.

Model checking techniques also have weaknesses. When developing a model checker one must take into account the following:

- It is mainly appropriate to control-intensive applications and less suited for data-intensive applications as data typically ranges over infinite domains.
- Its applicability is subject to decidability issues; for infinite-state systems, or reasoning about abstract data types (which requires logics), model checking is in general not effectively computable.
- It checks only stated requirements, i.e., there is no guarantee of completeness. The validity of properties that are not checked cannot be judged.
- It suffers from the state-space explosion problem, i.e., the number of states needed to model the system accurately may easily exceed the amount of available computer memory. Despite the development of several very effective methods to combat this problem, models of realistic systems may still be too large to fit in memory.
- It is not guaranteed to yield correct results: as with any tool, a model checker may contain software defects.

In recent years, model checking has become a very helpful tool for computer science. Some attempts, such as [27], [6] and [24], have been made to apply model checking to epistemic systems.

1.3 Model Checkers and Logic Puzzles

DEMO [6] is a model checker not based on a temporal epistemic architecture. DEMO is short for Dynamic Epistemic MOdelling. It allows modelling epistemic updates, graphical display of Kripke structures involved, and formula evaluation in epistemic states. DEMO is written in the functional programming language Haskell. The model checker DEMO implements the dynamic epistemic logic of Baltag [3].

DEMO has been used to model logic puzzles such as “The muddy children” [6, 25], “Sum and product” [28] and “What sum” [19, 26, 30].

This project aims to model these puzzles, as well as some others like the “Russian Cards Problem” [24], and real life problems, trying to give a finite representation of these models when possible.

Chapter 2

The Model Checker

The Model Checker is built in the Java programming language. It is developed in three phases, each one depending on the previous. Each phase handles different goals that this project wants to achieve. The Model Checker was planned this way since, unlike Haskell, Prolog and Lisp, Java is not a native logic programming language, meaning that Java is an object oriented programming language unlike the others that are declarative. They expressed the program logic in terms of relations represented as facts and rules.

In this chapter, the phases are briefly described, including some theoretical background. In later chapters the phases will be fully explained and will include some examples and results of running the model checker.

2.1 Module 1: Epistemic Logic S5.

$S5_m$ is considered by Goldbladtt [10] as one of the most well-known of epistemic logics and he considers it as the system characterized by the concept of logical necessity. The theory of computation has been focusing its attention on the system $S5_m$ in order to be able to represent knowledge and process information by robotic systems and other agents.

Some of the axioms for the $S5_m$ are:

- $K_i\varphi \rightarrow \varphi$
- $K_i\varphi \rightarrow K_iK_i\varphi$ ‘positive introspection’
- $\neg K_i\varphi \rightarrow K_i\neg K_i\varphi$ ‘negative introspection’

The first of these implies that, if the agent knows something, then that something is true. The second says that, if an agent knows something, then it knows that it knows it, while the third states that if it does not know something, then it knows that it does not know it.

The accessibility relations of this particular logic must include three properties:

- Reflexive

- Transitive
- Euclidean

Being reflexive means that each one of the states in the model have an accessibility relation that goes to themselves.

$$R \text{ is } \textit{reflexive} \text{ if } \forall s \in S(s,s) \in R$$

An accessibility relation is transitive whenever a state t is accessible from state s, and u turns to be accessible from state t, then u is accessible for s.

$$R \text{ is } \textit{transitive} \text{ if } \forall s,t,u \in S : (s,t) \in R \text{ and } (t,u) \in R \Rightarrow (s,u) \in R$$

The property of being euclidean is different from transitivity: both the euclidean property and transitivity infer a relation between t and u from relations between s and t and between s and u, but with different order in the relations. If a relation is symmetric, then the argument orders do not matter.

$$R \text{ is } \textit{euclidean} \text{ if } \forall s,t,u \in S : (s,t) \in R \text{ and } (s,u) \in R \Rightarrow (t,u) \in R$$

If a relation is euclidean and reflexive, it must also be symmetric and transitive. Such a relation is called “equivalence relation”.

This module will receive an $S5_m$ model and an epistemic formula as an input. What this module will do, is verify if the epistemic formula holds in a given state. The model checker will do this by verifying that the model and the formula fulfil the required characteristics of the S5 logic as well as the accessibility relations and axioms.

2.2 Module 2: Public Announcements.

A public announcement, as its name says, is an announcement that can be pronounced by an agent inside the model or by an agent that is not involved in it. It is possible that this announcement can help the rest of the agents to increase their own knowledge.

The language of public announcements $L_{K[]}$ defined by van Ditmarsch, van der Hoek and Kooi in the book [29, Chapter 4] is given by:

$$\varphi ::= p \mid \neg\varphi \mid (\varphi \wedge \psi) \mid K_i\varphi \mid [\varphi]\psi$$

The formula $[\varphi]\psi$ means that after truthful public announcement of φ, ψ will hold. It is important to notice that public announcements do not change facts, they simply help the agents to acquire more information about the model and themselves.

The principles relating announcements to knowledge are the axiom announcement and knowledge [25]. Other valid principles include:

- $[\varphi](\psi \rightarrow \chi) \rightarrow ([\varphi]\psi \rightarrow [\varphi]\chi)$
- $\langle\varphi\rangle\psi \rightarrow [\varphi]\psi$
- $[\varphi]\psi \leftrightarrow (\varphi \rightarrow [\varphi]\psi)$
- $[\varphi]p \leftrightarrow (\varphi \rightarrow p)$
- $[\varphi](\psi \wedge \chi) \leftrightarrow ([\varphi]\psi \wedge [\varphi]\chi)$
- $[\varphi](\psi \rightarrow \chi) \leftrightarrow ([\varphi]\psi \rightarrow [\varphi]\chi)$
- $[\varphi]\neg\psi \leftrightarrow (\varphi \rightarrow \neg[\varphi]\psi)$
- $[\varphi]K_i\psi \leftrightarrow (\varphi \rightarrow K_i[\varphi]\psi)$
- $[\varphi][\psi]\chi \leftrightarrow [\varphi \wedge [\varphi]\psi]\chi$

The dynamic modal operator that corresponds to the action of public announcements is $[\varphi]$. The semantics to handle public announcements are:

- $(M, s) \models p$ iff $s \in \pi_p$
- $(M, s) \models \neg\varphi$ iff $(M, s) \not\models \varphi$
- $(M, s) \models \varphi \wedge \psi$ iff $(M, s) \models \varphi$ and $(M, s) \models \psi$
- $(M, s) \models K_i\varphi$ iff $\forall t \in S : R_i(s, t)$ implies $(M, t) \models \varphi$
- $(M, s) \models [\varphi]\psi$ iff $(M, s) \models \varphi$ implies $(M \mid \varphi, s) \models \psi$

where $M|\varphi = \langle S', R', \pi' \rangle$ is defined as:

$$S' = [\![\varphi]\!]_M$$

$$R' = R_i \cap ([\![\varphi]\!]_M \times [\![\varphi]\!]_M)$$

$$\pi' = \pi \cap [\![\varphi]\!]_M$$

There is an extension for the semantics. It is represented by $\langle\varphi\rangle\psi$ and it means that after some announcement of φ, ψ holds. It is defined as:

- $(M, s) \models \langle\varphi\rangle\psi$ iff $(M, s) \models \varphi$ and $(M \mid \varphi, s) \models \psi$

The operator $\langle\varphi\rangle$ is the dual of $[\varphi]$ and what $\langle\varphi\rangle\psi$ means is that after some truthful public announcement φ, ψ will hold.

For the public announcement logic, the model checker can work in two ways. In the first way, requires the use of the *announce* function. The function will again receive an epistemic formula and an S5 model. The function will return a reduced model with the states were the announced formula holds. The second way is create a public announcement formula with the function *formula* and check if the formula holds in a given model with the function *check*. It is expected that all public announcements in the system are truthful announcements. False announcements are not handled by the system.

2.3 Module 3: Unbounded Models.

This module will analyse unbounded models. These are models which will follow a specific pattern capable of being represented with a regular expression. This means that the models are somehow dynamic and may grow indefinitely depending on the regular expression used to define them.

Figure 2.1 shows an example of an unbounded model. In the model checker this model will be created using the regular expression *"(pq)+"*. This expression means that the model will start with a state with the atomic proposition p followed by a state with the atomic proposition q. The plus sign, in the language of regular expressions means that the instruction between parenthesis should be repeated at least once.

The accessibility relations for the model should be deified separately. The reflexive relations will be created automatically but the outgoing relations must be defined for each one of the agents. The explanation of which regular expressions the model checker can handle, can be found in Chapter 6.3.

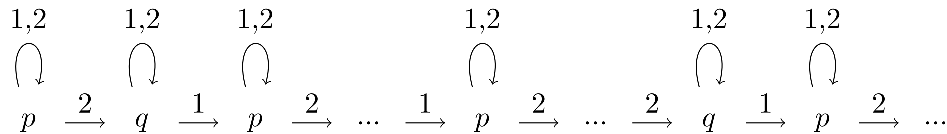

Figure 2.1: An unbounded model

2.4 Module 4: Common Knowledge.

Something can become common knowledge if every agent in the model knows, that every agent knows, that every agent knows and so on, for an infinite number of iterations. It calls for an extension of the $S5_m$ logic language and in order to define it, it is necessary to introduce two new logic operators, E which stands for "everybody knows", the notion of everybody knows is often called *general knowledge*, and C, which stands for "it is common knowledge that". The E operator is defined as:

- $E\varphi = K_1\varphi \wedge K_2\varphi \wedge ...K_m\varphi$

What the operator E says, is that all the agents in the model have acquired knowledge of a formula, in this case φ. The agents in the model may know it from the beginning, or they may have acquired the knowledge after some announcements. In any case all the agents now know φ. The common knowledge is achieved after everybody knows that everybody knows and so on. The common knowledge can only be understood intuitively since it goes to infinity and there is no absolute way to know when to stop.

Some of the axioms for the E and C operators are:

- $E\varphi \leftrightarrow K_1\varphi \wedge ... \wedge K_m\varphi$
- $C\varphi \rightarrow \varphi$

- $C\varphi \rightarrow EC\varphi$
- $C(\varphi \rightarrow \psi) \rightarrow (C\varphi \rightarrow C\psi)$
- $C\varphi \rightarrow (\varphi \wedge EC\varphi)$
- $(C\varphi \wedge C(\varphi \rightarrow \psi)) \rightarrow C\psi$
- $C(\varphi \rightarrow E\varphi) \rightarrow (\varphi \rightarrow C\varphi)$

The "Muddy children" puzzle described by, e. g. van Ditmarsch and Kooi in [25], models common knowledge. This puzzle shows that common knowledge can be acquired after $n-1$ iterations, where "n" is the total of muddy children.

Chapter 3

The Model

This chapter describes how the model checker should be used. In Section 3.1 the valid characters to define a formula are presented, then in Section 3.2 the main functions of the model checker are introduced including the explanation if each one of the functions as well as the expected result after executing them.

3.1 Valid Characters in Formulas

The model checker will have the feature to define epistemic formulas. The right definition of a logic formula in the model checker may include the following characters.

- *Alphanumeric strings*: Characters from $a - z$ and the digits from $0 - 9$ can be used to declare a logic proposition. Only lower case characters are allowed and they must start with a letter.
- !: The exclamation mark will work as the negation operator $\neg$.
- $($, $)$: The open and close parenthesis are used to group formulas, propositions and logic operators.
- &: The ampersand character will work as the conjunction operator $\wedge$.
- |: The pipe character will be used as the disjunction operator $p \vee q$, which is an abbreviation for $\neg(\neg p \wedge \neg q)$.
- $- >$: These two characters together will serve as the implication $\rightarrow$, which is an abbreviation for $\neg(p \wedge q)$.
- $< - >$: These three characters will be used as an abbreviation for $(p \rightarrow q) \wedge (q \rightarrow p)$.
- K: The capital K will be used as the knowledge operator and it must be followed by a number.
- $[,]$ The open and close brackets will be use to define a public announcement.
- C: The capital C will be used to determine the common knowledge.

- _: The underscore symbol will be used when creating a group of agents for common knowledge.

The combination of these characters will allow the definition of epistemic formulas in the model checker. The defined formulas can be used to be validated in a model or to be announced in the model.

3.2 Model Checker Functions

This section will describe definition and the syntax for the formulas allowed by the model checker. Bellow the formulas are listed as well as the use and the output of executing them. The formulas for the model checker are:

1. formula{p&q};
2. model{[(p,...,r)(..,..,..)(!p,...,!r)],[(1~2,1~3)(n~1,n~n)]};
3. announce{m,f};
4. check{m,f}; and check{m[1,2,...,n],f};
5. print{f};
6. delete{m};
7. longmodel{
propositions[p,...,q]
indices[1~100]
restriction[p < q]
relations[p+q=p+q]
};
8. longformula{
name of a previously defined formula
symbol of a logical operator
propositions[p,q]
indices[1~100]
restriction[p<q]
};
9. regex{
regular expression
};

3.2.1 Use of Functions

3.2.1.1 formula

The function *formula* will create a well-formed epistemic formula. The parameter that this function requires is string representing a well-formed formula. The characters accepted by this function are listed in Section 3.1. The output of this function will show

the written formula in a more traditional way. This output can be assigned to a variable to be used in later functions.

```
f = formula{p};
f=>
    p

f1 = formula{K1(p};
f1=>
    K1(p)

f2 = formula(p&q};
f2=>
    p&q

f3 = formula{q};
f3=>
    q
```

3.2.1.2 model

The function *model* will create an epistemic model with the received parameters. It needs two parameters to be executed. The first parameter defines the atomic propositions that are valid in each state.

It is important to notice that affirmations and negations must be explicitly defined, for instance, if a model M contains two states $s1$ and $s2$ and one wish to define the truth valuation for $s1$ as p and $s2$ as $\neg p$, the first parameter must be defined as $[(p)(!p)]$. The model checker will not accept empty valuations and also, it will not assume the negation of a valuation if it is not defined.

The second parameter will define the accessibility relations of each agent. For each agent, the user should open a parenthesis followed by the state of origin, then the relation symbol $\sim$ and the destination state. If the user wants to add another relation he should type a comma and then the new relation. Finally, when the user is done with the relations of the agent, he should close the parenthesis.

This process must be repeated for all the agents. Since the model checker is defined for the $S5$ logic, the reflexive relations can be omitted in the definition of the model since the model checker will create them automatically.

```
m   = model{[(p)(!p)(p)],[(1~2)(1~3)]};

m=>
     Agent: 1 [s1-s1,s1-s2,s2-s2,s3-s3]
     Agent: 2 [s1-s1,s1-s3,s2-s2,s3-s3]
     States[
        s1(p)  s2(!p)  s3(p)
    ]
```

3.2.1.3 announce

The function *announce* will receive two parameters. The first parameter is a model variable that must be previously defined. The second parameter its a previously defined formula. This function can give to types of outputs.

The first type will return a new model which is a reduced version of the input model containing only the states in which the given formula holds. The second type of output is

just a string indicating that the announcement was unsuccessful since it did not produce a change in the model.

```
a   = announce{m,f};
a=>
     Agent: 1 [s1-s1,s3-s3]
     Agent: 2 [s1-s1,s1-s3,s3-s3]
     States[
        s1(p)  s3(p)
    ]
```

3.2.1.4 check

The function *check* will validate if a formula holds in a state of the model. The parameters include the desired model as well as the formula that wants to be checked. The output will simply be a *true* or *false* statement depending if the given formula holds or not in the model.

This formula can be also used to check specific states of the model. This is done by writing the list of states between brackets after the name of the model.

```
check{m,f};
'f' holds for 'm' in:
     state 1 = true
     state 2 = false
     state 3 = true

check{m[2,3],f};
'f' holds for 'm' in:
     state 2 = false
     state 3 = true
```

3.2.1.5 print

The function *print* will show the content of a formula or a model.

```
print{f2};
f2=>
    p&q

print{m};
m=>
     Agent: 1 [s1-s1,s1-s2,s2-s2,s3-s3]
     Agent: 2 [s1-s1,s1-s3,s2-s2,s3-s3]
     States[
        s1(p)  s2(!p)  s3(p)
    ]
```

3.2.1.6 delete

The function *delete* will delete a variable that has been defined before.

```
delete{f};
Formula 'f' was deleted.

delete{m};
Model 'm' was deleted.
```

3.2.1.7 longmodel

The function *longmodel* must be used to create models that are too big and, therefore, it will be very hard to create them with the *model* function 3.2.1.2. The first parameter of the function is a list of atomic *propositions* that one wish to include in the model.

```
1   propositions[p,q]
```

Next, it is necessary to define the indices or combination of propositions that we want to include in the model, for instance if the *indices* is defined as:

```
1   indices[1~3]
```

The function will create states with all the possible combinations between the propositions and each of the combinations will create a different state in the model. The model will grow bigger depending on the number of propositions and the limit of the indices. The next example shows the result of the function with the same input indices but with different number of propositions.

```
1   propositions[p,q]
    indices[1~3]
3
    s1(p1 q1), s2(p1 q2), s3(p1 q3), s4(p2 q1), s5(p2 q2), s6(p2 q3), s7(p3 q1), s8(p3 q2), s9(p3 q3)
5
    propositions[p,q,r]
7   indices[1~3]

9   s1(p1 q1 r1),  s2(p1 q1 r2),  s3(p1 q1 r3),  s4(p1 q2 r1),  s5(p1 q2 r2),  s6(p1 q2 r3),  s7(p1 q3 r1),
    s8(p1 q3 r2),  s9(p1 q3 r3),  s10(p2 q1 r1), s11(p2 q1 r2), s12(p2 q1 r3), s13(p2 q2 r1), s14(p2 q2 r2),
11  s15(p2 q2 r3), s16(p2 q3 r1), s17(p2 q3 r2), s18(p2 q3 r3), s19(p3 q1 r1), s20(p3 q1 r2), s21(p3 q1 r3),
    s22(p3 q2 r1), s23(p3 q2 r2), s24(p3 q2 r3), s25(p3 q3 r1), s26(p3 q3 r2), s27(p4 q3 r3)
```

The next parameter of the function is the *restriction*. The restriction is used to remove states that are not of interest. It is possible to define as many restrictions as we want, the function will concatenate all this restrictions and will only leave the states that successfully fulfil all of the restrictions.

For instance, a function with two propositions and a indices form 1 to 3 will generate 9 states as seen in the previous example. If we want to keep only the states in which the p is smaller than q and $p+q$ is less or equal 4, we write the following restrictions.

```
    restriction[p<=q]
2   restriction[p+q<6]

4   s1(p1 q1), s2(p1 q2), s3(p1 q3), s4(p2 q2), s5(p2 q3)
```

The function now will show only the states in which the restrictions are fulfilled. The last parameter of the function is *relations*. This parameter will define the relations of each of the agents in the model. It is necessary to write as many relations as agents in the model, for example, if we want the model to have 3 agents involved, then it is necessary to use the relation parameter three times to define each one of the relations.

The relations will be created using the indices as reference, for instance, if we want to define the relations of an agent who cannot distinguish between the worlds which the sum of the indices of the origin state is the same as the sum of the indices of the destiny state, we write:

```
relations[pO+qO=pD+qD]
```

Note that even though the relation works with the indices that the state contains, in the definition of the relation it is used the name of the atomic proposition followed by a capital “O” for origin or a capital “D” for destiny. This is to map the values of the indices to the given propositions.

Many different relations can be defined. If one wish to define relations from one state to another which the sum of its indices is equal to 2 we write:

```
1     relations[qD+qD=2]
```

A complete definition of this function will look similar to the next example.

```
1     longmodel{
          propositions[p,q]
3         indices[1~3]
          restriction[p<=q]
5         restriction[p+q<6]
          relations[pO+qO=pD+qD]
7         relations[pD+qD=2]
      };
```

3.2.1.8 longformula

From a previously defined formula, the function *longformula* will create a formula concatenating the possible propositions combination using the submitted logical operator. The first parameter of the function is the name of the defined formula that one wish to repeat. The second parameter is the logical operator that one wish to use. It can be either &, |, $<->$ or $->$.

The definition of the rest of the parameters, *propositions*, *indices* and *restrictions*, is the same as the one specified for the *longmodel* function 3.2.1.7.

A complete definition of this function will look similar to the next example.

```
      f = formula{K1(p&q)};
2     lf = longformula{
          f
4         |
          propositions[p,q]
6         indices[1~3]
          restriction[p<=q]
8         restriction[p+q<6]
      };
```

3.2.1.9 regex

The *regex* will create unbounded models. Unbounded models are a collection of models which share similar characteristics and patterns. A more detailed explanation of this function as well as an introduction to regular expressions and how to build them, can be found in Chapter 6. An example of the use of the *regex* function is shown below.

```
1     u = regex{([!p,q]<1,2>|[q]);([p]<1>*)*};
      u=>
3         ([!p,q]<1,2>|[q]);([p]<1>*)*
```

Chapter 4

Model Checker for S5

In order to explain the model it is necessary to remember the Kripke semantics for the knowledge K operator.

$$(M, s) \models K_i\varphi \text{ iff } (M, t) \models \varphi \text{ for all } t \text{ with } (s, t) \in R_i$$

This means that an agent i knows φ in a state in a model (M, s) iff φ is true in all the accessible states to agent i, in other words, all states that the agent i considers possible. This means that if in a state s the agent is not sure about the true nature of the real world, it will consider several possible worlds t for which the relation $R(s, t)$ holds. If in all these possible worlds t with $R(s, t)$, φ holds, then the agent i has no doubts about φ any more. The agent certainly *knows* φ.

The model checker can create models with n number of states and m number of agents. Then it can verify if a logic formula holds and in which states of the model for each one of the agents. Consider the model in Figure 4.1;

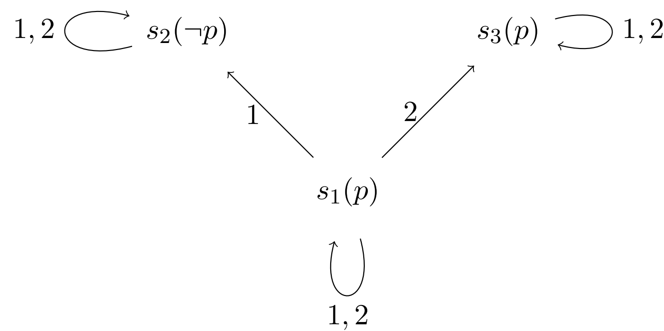

FIGURE 4.1: An $S5$ model with three worlds and two agents

This model can be created in the model checker with the following command.

```
1   m = model {[(p)(!p)(p)],[(1~2)(1~3)]};

3   m=>
        Agent: 1 [s1-s1,s1-s2,s2-s2,s3-s3,]
5       Agent: 2 [s1-s1,s1-s3,s2-s2,s3-s3,]
        States[
7           s1(p)  s2(!p)  s3(p)
      ]
```

Then the formulas that are going to be checked in the states of the model are defined.

```
f = formula {!K1 p};
f1 = formula {!K1 p & !K1 !p};
f2 = formula {K2 p};
f3 = formula {K1(K2 p | K2 !p )};
f4 = formula {!K2 !K1 p };

f=>
    !K1p

f1=>
    !K1p&!K1!p

f2=>
    K2p

f3=>
    K1(K2p|K2!p)

f4=>
    !K2!K1p
```

Once the model and the formulas are defined, it is possible to test them with the command check. The output of this function will show in which of the states of the model, the formula holds.

```
check{m,f};
'f' holds for 'm' in:
     state 1 = true
     state 2 = true
     state 3 = false
```

Agent 1 does not know that p in states $s1$ and $s2$.

- $(M, s_1) \vDash \neg K_1 p$
- $(M, s_2) \vDash \neg K_1 p$

```
check{m,f1};
'f1' holds for 'm' in:
     state 1 = true
     state 2 = false
     state 3 = false
```

Agent 1 does not know in $s1$ and $s3$ whether p holds.

- $(M, s_1) \vDash \neg K_1 p \wedge \neg K_1 \neg p$

```
check{m,f2};
'f2' holds for 'm' in:
     state 1 = true
     state 2 = false
     state 3 = true
```

Agent 2 knows that p in states $s1$ and $s3$.

- $(M, s_1) \vDash K_2 p$
- $(M, s_3) \vDash K_2 p$

```
check{m,f3};
'f3' holds for 'm' in:
     state 1 = true
     state 2 = true
     state 3 = true
```

Agent 1 knows that agent 2 knows whether p holds in states $s1$, $s2$ and $s3$.

- $(M, s_1) \vDash K_1(K_2 p \vee K_1 \neg p)$
- $(M, s_2) \vDash K_1(K_2 p \vee K_1 \neg p)$
- $(M, s_3) \vDash K_1(K_2 p \vee K_1 \neg p)$

```
1   check{m,f4};
    'f4' holds for 'm' in:
3        state 1 = true
         state 2 = false
5        state 3 = true
```

Agent 2 does not know that Agent 1 does not know that p in states $s1$ and $s3$.

- $(M, s_1) \vDash \neg K_2 \neg K_1 p$
- $(M, s_3) \vDash \neg K_2 \neg K_1 p$

Chapter 5

Public Announcements

The logic of public announcements is considered as an extension of multi-agent epistemic logic and it was first introduced by Plaza [22]. Public announcements can, in some cases, make common change of knowledge possible. The model checker is capable of handling the logic and semantics in public announcements described in Section 2.2.

In this chapter, some logic puzzles will be described and modelled in the model checker, showing the effects that public announcements have in the model.

5.1 The wise men

The wise men riddle was introduced by McCarthy in [20]. This riddle originally features three wise men trying to guess the colour of a spot on their forehead. A simplified version of McCarthy's wise men riddle goes like this:

> A certain king wishes to test his two wise men. He makes them stand in front of each other so they can see and hear the other wise man. The king says that he has two black hats and a white one and he is going to put one hat on each one of them. in this way a wise man can not see his own hat but can see the other person's hat.
>
> Then the king asks them sequentially if they know the colour of the hat on their own head. What will the wise men answer?

The first person can only answer "yes" to the king's question if he sees a white hat on the other person's head. This way he will know he has a black hat. Therefore the second person will answer "yes" to the question since he knows that the first person knows his hat colour. If the first person answer is "no" to the question, is because he is seeing a black hat on the other person's head, meaning that he himself can either be wearing a black or a white hat. However, this is enough information for the second person to know that he is wearing a black hat.

Figure 5.1 shows the model for this riddle and each state represents a possible colour combination of the hats.

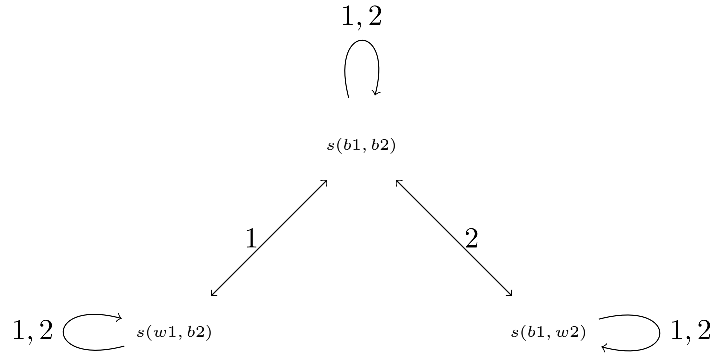

FIGURE 5.1: The model for the "Wise men riddle" where b stands for black hat and w for white hat. the number next to them represents the wise person.

This can be inserted in the model checker as follows:

```
1    wise = model {[(b1,b2)(w1,b2)(b1,w2)],[(1~2 ,2~1)(1~3 ,3~1)]};
     wise=>
3        Agent: 1 [s1-s1,s1-s2,s2-s2,s2-s1,s3-s3,]
         Agent: 2 [s1-s1,s1-s3,s2-s2,s3-s3,s3-s1,]
5        States[
            s1(b1,b2)  s2(w1,b2)  s3(b1,w2)
7       ]
```

The formula in which the first person knows the colour of both hats and the formula in which he does not know the colours can be written as:

```
1    yes = formula{K1(b1&w2)};
     no = formula{ !K1(b1|s1)&K1 b2};
3
     yes=>
5        K1(b1&w2)

7    no=>
         !K1(b1|s1)&K1b2
```

If these formulas are announced in the original model, they will yield different models. This can be done in the model checker with the *announce* function. After this, the second person will always know the colour of his hat.

```
     announceyes = announce {wise , yes };
2    announceno = announce {wise ,no };

4    announceyes=>
         Agent: 1 [s3-s3]
6        Agent: 2 [s3-s3]
         States[
8           s3(b1,w2)
        ]
10
     announceno=>
12       Agent: 1 [s1-s1,s1-s2,s2-s2,s2-s1]
         Agent: 2 [s1-s1,s2-s2]
14       States[
            s1(b1,b2)  s2(w1,b2)
16      ]
```

Another way to use the public announcement functionality of the model checker is by making the announcements in the defined formula. Let us say that the first person is wearing a white hat and then take the model *announceno*, defined in the previous

example, obtained after the first person says that he does not know the colour of his hat. If we want to check if the first person knows whether he is wearing a black or white hat after the second person announces that he knows his hat colour without using the *announce* function, it is possible to do it as follows:

```
iknow = formula{[K2(w1&b2)] K1 (w1&b2)};
iknow ==>
    [K2(b2|w2)]K1(b1|w1)
```

Then it is possible to check in which states, of the previously obtained *announceno* model, the formula holds.

```
check{announceno,iknow};

'iknow' holds for 'announceno' in:
     state 1 = false
     state 2 = true
```

This means that after the announcement, only state $s2$ will survive the announcement and it is the only state in which the first persons has a white hat and the second person a black one.

5.2 Muddy children

The muddy children puzzle has first been introduced by Barwise [4]. However, older variations of the same puzzle have been found, like the one described by Gamow and Stern involving cheating wives [9].

The muddy children puzzle goes like this:

> A group of children has been playing outside. After a while, their father told them to go back inside. The father notices that they are dirty and some of them may even have mud on their forehead. The children cannot see if they have mud on her own face but they are able to see the other children faces. The Father now says:
>
> “At least one of you has mud on your forehead.”
>
> Then the father says:
>
> “If you know you have mud on your face, step forward.”
>
> If nobody steps forward, the father will keep repeating the request. Assuming that they are the perfect children and they are truthful, intelligent and perfect logicians. What will happen after m announcements if m of n children are muddy?

There is a proof that the first $m-1$ times the father makes the announcement, no children will step forward, but in the mth time the muddy children will step forward. This proof can be found in, i.e. [21, 29].

We can model this puzzle in the model checker as follows:

Each of the muddy children will be represented as m followed by the a number. In our case we will create a model with three children $m1$, $m2$ and $m3$. If we want to say that a child is not muddy, we simply add the negation operator before the children. $!m1$.

The model must define a state for each possible combination involving the children. In case with three children, eight combinations are possible.

```
(!m1 & !m2 & !m3), (!m1 & !m2 & m3), (!m1 &  m2 & !m3), (!m1 &  m2 & m3),
(m1  & !m2 & !m3), (m1  & !m2 & m3), (m1  &  m2 & !m3), (m1  &  m2 & m3)
```

Finally it is necessary to include the relations between the states. These relations are given in the puzzle when it is mentioned that a child can not see his or her own face but can see the faces of the others. The command to insert all this information in the model checker is:

```
muddy = model{
    [
        (m1,m2,m3)(m1,m2,!m3)(m1,!m2,m3)
        (m1,!m2,!m3)(!m1,m2,m3)(!m1,m2,!m3)
        (!m1,!m2,m3)(!m1,!m2,!m3)
    ],
    [
        (1~5,5~1,2~6,6~2,3~7,7~2,4~8,8~4)
        (1~3,3~1,2~4,4~2,5~7,7~5,6~8,8~6)
        (1~2,2~1,3~4,4~3,5~6,6~5,7~8,8~7)
    ]
};
```

In this example, the first parameter, 8, represents the number of states in the model. The second one, 3, represents the number of agents in the model, in this case, the three children. The third parameter defines the possible combinations of children being muddy or not and the fourth parameter is the accessibility relation for each children. After submitting the function the output will look like this:

```
muddy=>
     Agent: 1 [
            s1-s1,s1-s5,s2-s2,s2-s6,s3-s3,s3-s7,s4-s4,s4-s8,
            s5-s5,s5-s1,s6-s6,s6-s2,s7-s7,s7-s2,s8-s8,s8-s4
        ]
     Agent: 2 [
            s1-s1,s1-s3,s2-s2,s2-s4,s3-s3,s3-s1,s4-s4,s4-s2,
            s5-s5,s5-s7,s6-s6,s6-s8,s7-s7,s7-s5,s8-s8,s8-s6
        ]
     Agent: 3 [
            s1-s1,s1-s2,s2-s2,s2-s1,s3-s3,s3-s4,s4-s4,s4-s3,
            s5-s5,s5-s6,s6-s6,s6-s5,s7-s7,s7-s8,s8-s8,s8-s7]
     States[
            s1(m1,m2,m3)  s2(m1,m2,!m3)  s3(m1,!m2,m3)  s4(m1,!m2,!m3)
            s5(!m1,m2,m3)  s6(!m1,m2,!m3)  s7(!m1,!m2,m3)  s8(!m1,!m2,!m3)
        ]
```

In a situation in which children 1 and 2 have muddy faces, the first announcement of the father, which is, "At least one of you has mud on his or her forehead.", is represented by the formula

$$(m_1 \vee m_2 \vee m_3)$$

means that child 1 or child 2 or child 3 is muddy. The model checker represents this formula as follows:

```
father1 = formula{m1 | m2 | m3};

father1=>
    m1|m2|m3
```

In order to announce this in the model it is necessary to call the function *announce*. This function will give another model as an output. The output model will be the same model as the one submitted as parameter without the states in which the announced formula did not hold.

```
firstannouncement = announce{muddy,father1};
firstannouncement=>
     Agent: 1 [
          s1-s1,s1-s5,s2-s2,s2-s6,s3-s3,s3-s7,s4-s4,
          s5-s5,s5-s1,s6-s6,s6-s2,s7-s7,s7-s2
      ]
     Agent: 2 [
          s1-s1,s1-s3,s2-s2,s2-s4,s3-s3,s3-s1,s4-s4,
          s4-s2,s5-s5,s5-s7,s6-s6,s7-s7,s7-s5
      ]
     Agent: 3 [
          s1-s1,s1-s2,s2-s2,s2-s1,s3-s3,s3-s4,s4-s4
          ,s4-s3,s5-s5,s5-s6,s6-s6,s6-s5,s7-s7
      ]
     States[
          s1(m1,m2,m3)  s2(m1,m2,!m3)  s3(m1,!m2,m3)  s4(m1,!m2,!m3)
          s5(!m1,m2,m3)  s6(!m1,m2,!m3)  s7(!m1,!m2,m3)
      ]
```

In this case the state $s8$, in which the formula did not hold because in this state all children were clean $s8(!m2, !m3, !m1)$, was removed. Then the father says for the first time "If you know you have mud on your face, step forward.". At his point, none of the children know yet whether he or she is muddy or not. This uncertainty of a child not knowing whether he or she is muddy will be represented with the formula:

$$\neg(K_1 m_1 \vee K_1 \neg m_1) \wedge \neg(K_2 m2 \vee K_2 \neg m_2) \wedge \neg(K_3 m_3 \vee K_3 \neg m_3)$$

and it is represented in the model checker as:

```
father2 = formula{!(K1m1 | K1!m1) & !(K2m2 | K2!m2) & !(K3m3 | K3!m3)};
father2=>
    !(K1m1|K1!m1)&!(K2m2|K2!m2)&!(K3m3|K3!m3)
```

This formula should be announced in the last model that was obtained.

```
secondannouncement = announce{firstannouncement,father2};
secondannouncement ==>
    secondannouncement=>
         Agent: 1 [s1-s1,s1-s5,s2-s2,s3-s3,s5-s5,s5-s1]
         Agent: 2 [s1-s1,s1-s3,s2-s2,s3-s3,s3-s1,s5-s5]
         Agent: 3 [s1-s1,s1-s2,s2-s2,s2-s1,s3-s3,s5-s5]
         States[
              s1(m1,m2,m3)  s2(m1,m2,!m3)  s3(m1,!m2,m3)  s5(!m1,m2,m3)
           ]
```

This time only four states survived the announcement. The father makes the announcement once again and this time children one and two step forward since this time they both know that they are muddy and only one state will survive at the end.

$$(K_1 m1 \vee K_1 \neg m_1) \wedge (K_2 m2 \vee K_3 m3)$$

```
father3 = formula{(K1m1 | K1!m1) & (K2m2 | K2!m2 )};
father3=>
    (K1m1|K1!m1)&(K2m2|K2!m2)

final = announce{secondannouncement,father3};
final=>
     Agent: 1 [s2-s2]
     Agent: 2 [s2-s2]
     Agent: 3 [s2-s2]
     States[
          s2(m1,m2,!m3)
    ]
```

With two muddy children, the model is reduced to the right states after two announcements. A similar situation can be simulated in the model that includes more children and more muddy children. In the end the number of necessary announcements to reduce the model to one state, will be the same as the number of muddy children.

5.3 Sum and product

The official sum and product riddle was first introduced by Freudenthal [7]. A translation of the original riddle publication can be found in [28] and it goes like this:

> One person tells to S and P: "I have chosen a pair of integers x and y such that $1 < x < y$ and $x + y \leq 100$. Now I am going to tell only to S the sum of x and y and only to P product of x and y. These announcements will remain private". After this, the following conversation between S and P takes place:
>
> 1. P: "I do not know the numbers."
> 2. S: "I knew you did not."
> 3. P: "Now I know the numbers."
> 4. S: "Now I know them too."
>
> Determine the numbers x and y.

This problem can be modelled as follows:

```
sumproduct = longmodel{
    propositions[x,y]
    indices[1~100]
    restriction[1<x]
    restriction[x<y]
    restriction[x+y<=100]
    relations[x0+y0=xD+yD]
    relations[x0*y0=xD*yD]
};
```

The function will create a model with 2352 states with a pair of x and y that will fulfil the conditions of the restrictions. Also, the function will create two agents for the model, one which cannot distinguish between states that have the same sum of x and y and another one which cannot distinguish between states that have the same product of x and y.

In this riddle, four announcements are made. The formulas for these announcements are represented by van Ditmarsch, Ruan and Verbrugge [28] as follows:

1. P: "I do not know the numbers.": $\neg K_P(x, y)$
2. S: "I knew did not.": $\neg K_S(\neg K_P(x, y))$
3. P: "Now I know the numbers.": $K_P(x, y)$
4. S: "Now I know them too.": $K_S(x, y)$

In the riddle, x and y are two integers such that $1 < x < y$ and $x+y \leq 100$. This can be defined as $I = \{(x, y) \in N \mid 1 < x < y \text{ and } x + y \leq 100\}$ [28] With this information the formulas can be rewritten as:

1. $\neg \bigvee_{(i,j)\in I} K_P(x_i \wedge y_j)$
2. $K_S \neg \bigvee_{(i,j)\in I} K_P(x_i \wedge y_j)$ or equivalent $K_S \bigwedge_{(i,j)\in I} \neg K_P(x_i \wedge y_j)$
3. $\bigvee_{(i,j)\in I} K_P(x_i \wedge y_j)$

4. $\bigvee_{(i,j)\in I} K_S(x_i \wedge y_j)$

Once the formulas have been defined, we can introduce them in the model checker with the commands shown in the next example. In this case Sum is represented by $K1$ and Product by $K2$

```
as1 =formula{K1(!K2(x&y))};
ap1 =formula{K2(x&y)};
as2 =formula{K1(x&y)};

announcesum1 = longformula{
    as1 &
    propositions[x,y]
    indices[1~100]
    restriction[1<x]
    restriction[x<y]
    restriction[x+y<=100]
};

announceprod1 = longformula{
    ap1 |
    propositions[x,y]
    indices[1~100]
    restriction[x<y]
    restriction[x+y<=100]
};

announcesum2 = longformula{
    as2 |
    propositions[x,y]
    indices[1~100]
    restriction[x<y]
    restriction[x+y<=100]
};
```

Finally, these announcements can be made in the model using the *announce* function. The first announcement must be made in the original model *sumpoduct*:

```
sumproduct1 = announce{sumproduct,announcesum1};
sumproduct2 = announce{sumproduct1,announceprod1};
sumproduct3 = announce{sumproduct2,announcesum2};
```

The output of this function is to big too be written here, but after the announcement the model is reduced from 2352 states to 145 only. Then, the second announcement, *announceprod1*, should be made in this new model.

```
sumproduct2 = announce{sumproduct1,announceprod1};
```

After this announcement only 86 states survive. The last announcement is made and this time the output is a model with a single state. The model checker will show the result as follows:

```
sumproduct3 = announce{sumproduct2,announcesum2};
sumproduct3 =>
     Agent: 1 [199-199]
     Agent: 2 [199-199]
     States[
        s199(x4,y13)
    ]
```

Chapter 6

Unbounded Models

For this project, unbounded models are a collection of models which share similar characteristics and patterns since they are created using a regular expression. Given a regular expression and an epistemic formula, the model checker will iterate through the collection of models defined by the regular expression, checking if the formula holds only in the first state of the current model. If the formula holds in the first state of the model, the model checker will stop iterating and will show the model in which the formula holds. If it is not possible that the formula holds for any of the models defined by the regular expression, the model checker will only show a message with this information.

In these Sections 6.1 to 6.4 we shall introduce important definitions of terms necessary to create regular expressions. These concepts include the "alphabet", a set of symbols, "strings", a list of symbols from an alphabet, and "language", a set of strings from the same alphabet. Section 6.5 for the regular expressions will be explained afterwards, also, basic examples of how to create unbounded models are shown. The next section shows how to model the "Two Generals Problem" [11] with a regular expression. The last section will show how to use the regular expressions in the model checker and how to evaluate formulas in the regular expressions.

6.1 Alphabet

An *alphabet* is a finite non-empty set of symbols Σ. Examples of commonly used alphabets include:

1. Binary alphabet: $\Sigma_b = \{0, 1\}$
2. Set of lower case letters: $\Sigma_l = \{a, b, ..., z\}$
3. The set of all ASCII characters.
4. The set of digits: $\Sigma_d = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$

The alphabet of the model checker is given by the set of symbols containing all the lower case alphabets, the digits, the square brackets, greater and less symbols, the negation symbol and the comma:

$$\Sigma_{MC} = \{a, b, ..., z, 0, 1, 2, ..., 9, [,], <, >, !, comma\}$$

6.2 Strings

Hopcroft in [13, Chapter 1] defines a *string* as finite sequence of symbols chosen from some alphabet.

6.2.1 Powers of an Alphabet:

Hopcroft mentions that if Σ is an alphabet, it is possible to express the set of all strings of a certain length from that alphabet by using an exponential notation. Therefore, Σ^k is defined as the set of strings of length k, each of whose symbols is in Σ. It is important to remember that $\Sigma^0 = \{\epsilon\}$, regardless the alphabet Σ, since ϵ is the only string of length 0.

The set of all possible strings over an alphabet Σ is defined as:

$$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup ...$$

6.2.2 Model checker strings

The strings in the model checker can be formed using the symbols defined in the alphabet Σ_{MC}. However, as in the English language, not all the strings, or words, that can be written with the set of symbols in the alphabet are well-formed in the model checker. Since the strings will be representing models or more precisely, states of a model, the strings that are well-formed for the model checker must follow certain rules:

1. Atomic propositions must be between square brackets and they must be separated by a comma. $[p, q, r]$
2. Atomic propositions can be preceded by the negation symbol '!'. $[!p, !q]$
3. Atomic propositions can be followed by numeric indices. $[p_1, p_2, p_3]$
4. After the propositions it is necessary to define accessibility relations for a group of agents. The agents must be between the less and greater symbols and they must be separated by a comma. $\langle 1, 2, 3\rangle$

With these rules it is possible to define strings that are well-formed for the model checker. Examples of well-formed strings are:

- $\sigma_1 = [p]\langle 1\rangle$
- $\sigma_2 = [p, q, ...]\langle 1, 2\rangle$
- $\sigma_3 = [p_1, p_2, .., p_n]\langle 1, 2, 3\rangle$
- $\sigma_4 = [p]\langle 1, 2, ..., m\rangle$

- $\sigma_5 = [p, q, ...]\langle 2, 3\rangle$
- $\sigma_6 = [p_1, p_2, .., p_n]\langle 1\rangle$

6.2.3 Languages

Hopcroft defines a *language* L as a (possibly infinite) set of strings, all of which are chosen from some alphabet Σ^* where Σ is a particular alphabet. If Σ is an alphabet, and $L \subseteq \Sigma^*$, then L is a language over Σ. Notice that a language over Σ might not include strings with all the symbols of Σ, so once it is established that L is a language over Σ, it is also known that it is a language over any alphabet that is a superset of Σ.

Some abstract examples of languages mentioned by Hopcroft are:

1. The set of strings of 0's and 1's with an equal number of each:

$$\{01, 10, 0011, 0101, 1001, ...\}$$

2. The set of binary numbers whose value is a prime:

$$\{10, 11, 101, 111, 1011, ...\}$$

3. Σ^* is a language for any alphabet Σ.

These are just general examples of languages, they do no represent regular languages.

6.3 Regular Expressions

Regular expressions are strings of symbols that describe strings that can be represented by finite automata. They are used in many common types of software, including tools to search for patterns in text. In this case the regular expressions will be used to create a collection of models that share similar patterns.

The idea of regular expressions and the proof of their equivalence to finite automata is the work of S. C. Kleene [15]. Kleene's theorem is as follows:

Kleene's Theorem: Let L be a language over an alphabet Σ. Then L is regular if and only if it is the language accepted by some finite automaton with alphabet Σ.

What this theorem implies is that regular expressions have the same expressive power as finite automata. We can translate any regular expression into a finite automaton that accepts its language and we can take a finite automaton and convert it into an equivalent regular expression.

Regular expressions denote languages. For instance, the regular expression $01^* \mid 10^*$ denotes the language consisting of all strings that are either a single 0 followed by any number of 1's or a single 1 followed by any number of 0's. In order to explain the symbols used in the regular expression and describe the regular expression notation it

is necessary to explain the three operations that can be performed on languages. These operations are defined by Hopcroft as follows:

Union: The union of two languages L and M , denoted $L \cup M$ is the set of strings that are either in L or M or both. For instance:

$$\text{If } L = \{001, 10, 111\} \text{ and } M = \{010\}, \text{ then } L \cup M = \{10, 001, 010, 111\}$$

The operator for the union of languages used in the model checker is the pipe '|'

Concatenation: The concatenation of languages L and M is the set of strings that can be formed by taking any string in L and concatenating it with any string in M. For instance:

$$\text{If } L = \{001, 10, 111\} \text{ and } M = \{010\} \text{ then } L; M \text{ is } \{001, 10, 111, 001010, 10010, 111010\}$$

The concatenation of languages in the model checker is denoted by the semicolon ';'.

Closure The closure of a language L is denoted L^* and represents the set of those strings that can be formed by taking any number of strings from L, possibly with repetitions (i.e., the same string may be selected more than once) and concatenating all of them. For instance:

$$\text{If } L = \{0, 1\}, \text{ then } L^* \text{ is the set of all strings of 0's and 1's.}$$

More formally, L^* is the infinite union $\cup_{1 \geq 0} L^i$, where $L^1 = L$ and L^i for $i > 1$ is $L; L; L; ... \; i$ times.

After defining the basic operations for regular expressions, it is possible to introduce the inductive definition of regular expressions given by Hopcroft in [13, Chapter 3]. There are four parts in the inductive step, one for each of the operators and one for the introduction of parenthesis:

1. If E and F are regular expressions, then $E \mid F$ is a regular expression denoting the union of $L(E)$ and $L(F)$. That is, $L(E \mid F) = L(E) \cup L(F)$.

2. If E and F are regular expressions, then $E; F$ is a regular expression denoting the concatenation of $L(E)$ and $L(F)$. That is, $L(E; F) = L(E); L(F)$.

3. If E is a regular expression, then E^* is a regular expression, denoting the closure of $L(E)$. That is, $L(E^*) = (L(E))^*$.

4. If E is a regular expression, then (E), a parenthesized E, is also a regular expression, denoting the same language as E. Formally; $L((E)) = L(E)$.

The definition of regular expressions in the model checker is given by:

$$E = \quad \sigma \quad | \quad E; E \quad | \quad E|E \quad | \quad E^* \quad | \, (E)$$

6.3.1 Priority of operators.

Just like the operators in regular algebra, the operators for regular expressions have a specific priority which means that the operators are associated with operands in a particular order.

The closure operator '*' is the one with the highest priority. This means that the operator is applied to the smallest sequence of symbols to its left.

The concatenation ';' has lower priority than the closure. This operator groups expressions that are adjacent to it without the intervention of another operator. In this case, the order of the operands is important since it is required to define the accessibility relations between the models.

The union operator '|' is the one with the lowest priority and it is an associative operator so the order of the operands is not really important, still, it is going to be grouped from left to right, for instance, the expression $E_1 \mid E_2 \mid E_3 \mid E_4$ will be grouped as $(((E_1 \mid E_2) \mid E_3) \mid E_4)$.

There may be occasions that we do not want to group expressions according to the priority of the operators. In these occasions the use of parentheses to group expressions is required. The use of parentheses is the same as in algebraic expressions and they can be used to group expressions even if the desired grouping is implied by the hierarchy of the operators.

6.4 Building Regular Expressions

In Section 6.2.2 was explained what kind of strings are accepted by the model checker. In order to write complex regular expressions accepted by the model checker let us take a look at the automaton in Figure 6.5. By following this automaton, we will be able to create a great variety of different regular expressions using the 3 basic operators.

An example of the most simple regular expression we can write with this automaton is $[p]\langle 1\rangle$. Regular expressions like this, are the base of unbounded models since each of this simple expressions represent a state in the model. This regular expression will create a model with a single state with the atomic proposition p and a single agent as shown in Figure 6.1.

FIGURE 6.1: Model created with the regular expression $[p]\langle 1\rangle$

The next step will be to use the concatenation operator, the semicolon. We can concatenate regular expressions like the previous one by just adding a semicolon between them: for instance, if we have 3 expressions, $[p]\langle 1\rangle$, $[q]\langle 2\rangle$ and $[p, q]\langle 1, 2\rangle$ we can concatenate them using the semicolon operator $[p]\langle 1\rangle; [q]\langle 2\rangle; [p, q]\langle 1, 2\rangle$.

The model that this regular expression creates, can be seen in Figure 6.2. It is important to notice that, as in the regular models, the reflexive relations will be created automatically in unbounded models. Also, by taking a closer look you may notice that the last string creates an accessibility relation for agents 1 and 2 but since there is no other expression to be concatenated with, the relation will just be ignored.

FIGURE 6.2: Model created with the regular expression $[p]\langle 1\rangle; [q]\langle 2\rangle; [p, q]\langle 1\rangle$

The use of the union operator, the pipe, is similar to the semicolon. By placing the union operator between expressions we will have the option to decide between one model or another, for instance, the regular expression $[p]\langle 1\rangle; [p]\langle 2\rangle; ([q]\langle 1\rangle | [!q]\langle 2\rangle); [p]\langle 1\rangle$ will create the model shown in Figure 6.3

FIGURE 6.3: Class of models created with the regular expression $[p]\langle 1\rangle; [p]\langle 2\rangle; ([q]\langle 1\rangle | [!q]\langle 2\rangle); [p]\langle 1\rangle$

The dotted lines represent the two possible options at which we can arrive when we use the union operator. If we use more union operators, we will have more choices. In this example, the two possible paths can only arrive at the same state because there are no more union operators.

The closure operator, the star, allows us to write a potentially infinite concatenation. Unlike the union and the concatenation, the closure is an unary operator, we only require a single expression to use it. Figure 6.4 shows an example of the use of the closure operator.

FIGURE 6.4: Class of models created with the regular expression $[p, q]\langle 2\rangle; ([q]\langle 1, 2\rangle; [r]\langle 1, 2\rangle)^*$

As mentioned before, the simplest way to insert regular expression in the model checker is by following the automaton in Figure 6.5. It will guide you step by step when writing regular expressions for the model checker.

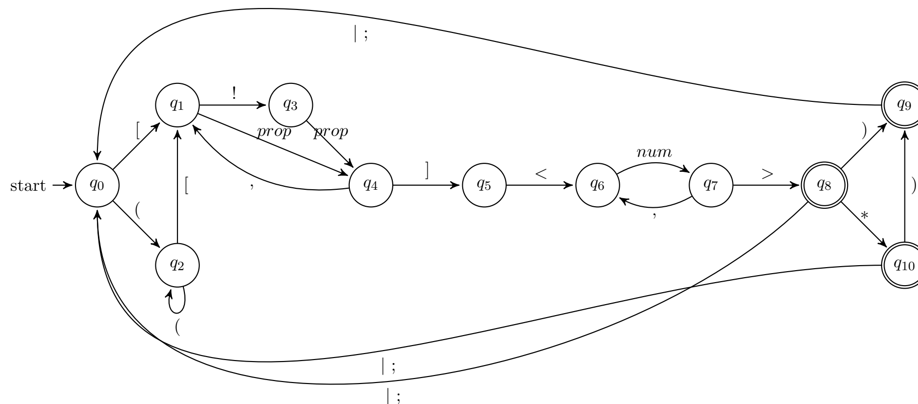

Figure 6.5: Automaton used to check well-formed regular expressions.

Regular expressions are a powerful tool when creating models and some of them may be more efficient than others. The processing time in the model checker will depend on how efficient the regular expression is. Try to avoid redundancy and the creation of potentially unnecessary states.

As seen in this chapter, the structure of the unbounded models created by a regular expression is linear. As in regular languages, regular expressions are used to define strings, therefore, the regular expressions in this model checker will create models with a string like structure resembling the strings created in regular languages.

6.5 Two Generals Problem

This problem was first introduced as the "Two Generals Paradox" by Gray in [11] and then a more general version was published by Lamport in [17]. A proof of the impossibility of this problem was published by Akkoyunlu in [1].

The original version of the problem by Gray goes like this:

> There are two generals on campaign. They have an objective (a hill) that they want to capture. If they simultaneously march on the objective they are assured of success. If only one marches, he will be annihilated.
>
> The generals are encamped only a short distance apart, but due to technical difficulties, they can communicate only via runners. These messengers have a flaw; every time they venture out of camp they stand some chance of getting lost (they are not very smart.)
>
> The problem is to find some protocol that allows the generals to march together even though some messengers get lost.

In this problem it is assumed that neither of the generals will attack without knowing for sure that the other one will attack too. If one of the generals, named 1, wants to coordinate a simultaneous attack, he will use a messenger who has to go through the enemy territory to deliver a message m to the other general.

For general 1 to be certain that the message was delivered, an acknowledgement K_2m must be sent back by the recipient general, named 2, if he receives the message, but there is also a chance that the acknowledgement gets lost. In order for the second general to be sure about the simultaneous attack he needs to have an acknowledgement of the arrival of his acknowledgement, i. e. he needs to know that K_1K_2m. Then again, the first general will need a confirmation of his acknowledgement, he needs to know that $K_2K_1K_2m$, and so on.

In this problem, as soon as a general sends an acknowledgement, he is in a state of uncertainty since he cannot distinguish between a state where the message has been delivered from a state in which the message was not delivered. Figure 6.6 shows an epistemic model for the two generals problem.

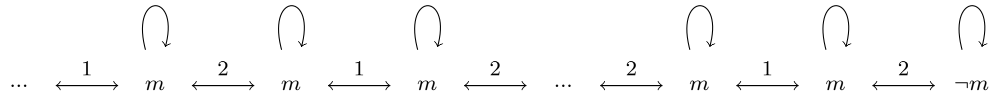

FIGURE 6.6: Epistemic representation of the "Two generals problem"

In order to create this model with a regular expression we should identify which strings are involved. In this case 3 different strings are necessary to create the model:

- $\sigma_a = [m]\langle 2\rangle$
- $\sigma_b = [m]\langle 1\rangle$
- $\sigma_c = [!m]\langle 1\rangle$

The strings σ_a and σ_b represent states with the atomic proposition m with the difference that σ_a will create relations for general 1 and σ_b will create the relations for general 2. σ_c will be the last state to be added to the model.

When creating a regular expression for a model such as the model for the two generals problem, we can take a look at the characteristics of the model and build the regular expression step by step as shown below:

1. The model can alternate between σ_a and σ_b for an unbounded number of times.

$$(\sigma_a; \sigma_b)^*$$

2. The model will always end with σ_c

$$(\sigma_a; \sigma_b)^*; \sigma_c$$

3. σ_c will never be preceded by σ_b.

$$(\sigma_b; \sigma_a)^*; \sigma_c$$

4. The model can start with either σ_a or σ_b

$$(\sigma_a \mid \sigma_b); (\sigma_b; \sigma_a)^*; \sigma_c$$

5. σ_b cannot be followed by another σ_b or σ_c (a|

$$(\sigma_a \mid (\sigma_b; \sigma_a)); (\sigma_b; \sigma_a)^*; \sigma_c$$

Just as the Kleene's theorem specifies, Figure 6.7 shows the automaton that corresponds to the obtained regular expression. Keep in mind that there may be other regular expressions that could define the two generals problem. Some representations may be more efficient than others.

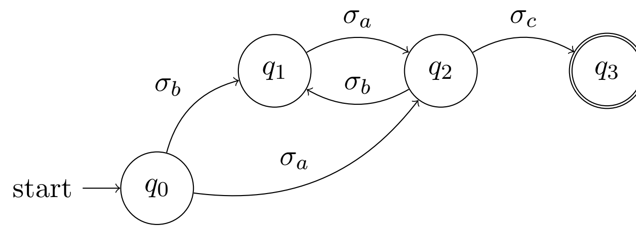

FIGURE 6.7: Automaton for the regular expression: $(\sigma_a \mid (\sigma_b; \sigma_a)); (\sigma_b; \sigma_a)^*; \sigma_c$

As it can be seen in the automaton, the collection of models that can be defined by this regular expression is potentially infinite due to the closure operation. Some examples accepted for the automaton in Figure 6.7 are:

- $\sigma_a \sigma_c$
- $\sigma_b \sigma_a \sigma_c$
- $\sigma_a \sigma_b \sigma_a \sigma_c$

The next section will show how the regular expressions can be used in the model checker as well as checking some epistemic formulas in models defined by regular expressions.

6.6 The *regex* function.

Once we have defined a regular expression, it is possible to introduce it to the model checker with the *regex*. Just as with the *formula* and *model* functions, it is possible to assign the value of this function to a variable. Once a regular expression is defined, we can use the *check* function on the expression. Consider the next example:

```
1   unbounded = regex{[p]<1,2>*;[p,q]<1>};
    unbounded=>
3       [p]<1,2>*;[p,q]<1>
```

This can create models of which the last state contains the atomic propositions p and q and is preceded by 0 or more states which contain a single proposition p. Now let us define the formulas we want to test in this regular expression and use the *check* function to test them in the regular expression:

```
1       f1 = formula{p};
        f2 = formula{K3 p };
3       f3 = formula{K1 r};
        f4 = formula{K1K1K2K2K1(p|q)};
5
        check{unbounded,f1};
7       check{unbounded,f2};
        check{unbounded,f3};
9       check{unbounded,f4};
```

The model checker will search for the shortest model in which the formula can hold in the first state. For the formula *f1*, the shortest model in which the formula can hold, is a model which contains a single state with the atomic propositions p and q. The output of the model checker is:

```
1   The formula f1 can hold in the first state of model =>
        $temp$=>
3        States[
            s1(p,q)
5       ]
```

The second and third formula *f2* and *f3* will not hold in the first state of any model defined by the regular expression *unbounded*. The reason is that the regular expression never defines a third agent and the formula *f2* is asking for agent 3. The formula *f3* will never hold because the proposition r is not included in any state of the regular expression. The output of the model checker for these two formulas is:

```
1       The formula 'f2' cannot hold in the first state of models defined by regular expression 'unbounded'

3       The formula 'f3' cannot hold in the first state of models defined by regular expression 'unbounded'
```

The last formula *f4*, may seem at first sight to require a model with many states in order to hold due to the level of knowledge required by the agents $K_1K_1K_2K_2K_1$, however, the output model contains only two states:

```
1   The formula f4 can hold in the first state of model =>
        $temp$=>
3        Agent: 1 [s1-s1,s1-s2,s2-s2,s2-s1]
         Agent: 2 [s1-s1,s1-s2,s2-s2,s2-s1]
5        States[
            s1(p)  s2(p,q)
7       ]
```

This does not mean that a formula can only hold in this model, the model checker only shows the *smallest* model in which the formula can hold. Notice also that the state (p, q) went from state 1 in formula *f1* to state 3 in formula *f4*.

The two generals problem can be modelled and tested in the same way. The next example shows a program that declares a regular expression for the problem and some formulas that can be tested.

```
1       generals = regex{([m]<2>|[m]<1>;[m]<2>);([m]<1>;[m]<2>)*;[!m]<1>};
        m1 = formula{K2(m | !m)};
3       m2 = formula{K1 m};
        m3 = formula{K1 m | K1 !m};
5       m4 = formula{K1K2K1 m};
        m5 = formula{!K2K1K2K1 m};
7
        check{generals,m1};
9       check{generals,m2};
        check{generals,m3};
11      check{generals,m4};
        check{generals,m5};
13
        generals=>
15          ([m]<2>|[m]<1>;[m]<2>);([m]<1>;[m]<2>)*;[!m]<1>
```

```
17      m1=>
            K2(m|!m)
19
        m2=>
21          K1m

23      m3=>
            K1m|K1!m
25
        m4=>
27          K1K2K1m

29      m5=>
            !K2K1K2K1m
31
        The formula m1 can hold in the first state of model =>
33          $temp$=>
             Agent: 1 [s1-s1,s2-s2,]
35           Agent: 2 [s1-s1,s1-s2,s2-s2,s2-s1,]
             States[
37              s1(m,)  s2(!m,)
            ]
39
        The formula m2 can hold in the first state of model =>
41          $temp$=>
             Agent: 1 [s1-s1,s2-s2,]
43           Agent: 2 [s1-s1,s1-s2,s2-s2,s2-s1,]
             States[
45              s1(m,)  s2(!m,)
            ]
47
        The formula m3 can hold in the first state of model =>
49          $temp$=>
             Agent: 1 [s1-s1,s2-s2,]
51           Agent: 2 [s1-s1,s1-s2,s2-s2,s2-s1,]
             States[
53              s1(m,)  s2(!m,)
            ]
55
        The formula m4 can hold in the first state of model =>
57          $temp$=>
             Agent: 1 [s1-s1,s2-s2,s2-s3,s3-s3,s3-s2,s4-s4,]
59           Agent: 2 [s1-s1,s1-s2,s2-s2,s2-s1,s3-s3,s3-s4,s4-s4,s4-s3,]
             States[
61              s1(m,)  s2(m,)  s3(m,)  s4(!m,)

63          ]

65      The formula m5 can hold in the first state of model =>
            $temp$=>
67           Agent: 1 [s1-s1,s2-s2,]
             Agent: 2 [s1-s1,s1-s2,s2-s2,s2-s1,]
69           States[
                s1(m,)  s2(!m,)
71          ]
```

Figure 6.8 shows the output model for the formulas $m2$ and $m3$. It may seem that these two formulas could hold on the second states of the model, meaning that there may be a smaller model in which the formula can hold. However a shorter model will not include agent 1, therefore, these two formulas will not hold in that model.

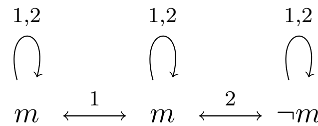

FIGURE 6.8: Model generated by testing $m2$ and $m3$ in the generals regular expression.

Figure 6.9 shows the shortest model in the collection defined by the regular expression $([m]\langle 1\rangle|[m]\langle 2\rangle;[m]\langle 1\rangle);([m]\langle 2\rangle;[m]\langle 1\rangle)^*;[!m]$. Although the regular expression contains two agents, the string $[m]\langle 1\rangle;[!m]$, which is the simplest one that can be obtained, does not include agent 2.

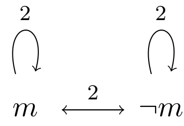

FIGURE 6.9: Model generated by testing $m1$ in the generals regular expression.

Chapter 7

Common Knowledge

Common knowledge is a special kind of knowledge for a group of agents. It was first introduced by Morris F. Friedell in [8]. Common Knowledge is useful for agents since it allows them to communicate or coordinate their behaviour. In order for the agents to have a successful interaction, agents require mutual understandings or background knowledge.

This chapter will first introduce the language and the semantics of common knowledge. This chapter will also cover the special case of common knowledge and public announcements. Finally this chapter will show how to use the common knowledge operator in the model checker.

7.1 The Language of Common Knowledge

The logic of knowledge with common knowledge is denoted $S5_C$. Common knowledge is a very strong notion, therefore, most of the time it can only be obtained for weak formulas φ. Another way to understand the notion of common knowledge is to realise in which situations $C_B\varphi$ does not hold for a group B. This is the case as long as someone, on the grounds of their knowledge, considers it a possibility that someone considers it a possibility that someone... that φ does not hold.

Van Ditmarsch, van der Hoek and Kooi define the language of common knowledge in [29, Chapter 2] as follows:

Let P be a set of atomic propositions, and A a set of agent-symbols. a, b, c, ... are variables over A, and B is a variable over coalitions of agents, i.e., subsets of A. The language L_{KC}, the language for multi-agent epistemic logic with common knowledge, is given by:

$$\varphi ::= p \mid \neg\varphi \mid (\varphi \wedge \varphi) \mid K_a\varphi \mid C_B\varphi$$

7.2 Semantics of Common Knowledge

The semantic definition for common knowledge given by Meyer and van der Hoek in the book [21, Chapter 2] says that, given a Kripke model $M = \langle S, R_1, ..., R_m, \pi \rangle$, let $s, t \in S$

1. Then, instead of $(s,t) \in R_i$ it is written $s \rightarrow_{R_i} t$, where $i \in B$.
2. Instead of $(s,t) \in R_1 \cup ... \cup R_m$ it is written $s \rightarrow t$. If there is a sequence $s = s_0 \rightarrow s_1 \rightarrow ... \rightarrow s_k = t$, for some $s_i (0 \leq i \leq k)$ where $[i,k] \in B$, it is written $s \rightarrow^k t$.
3. The relation $\twoheadrightarrow_B$ is the *reflexive-transitive closure* of the one step reachability relation $\rightarrow$. Thus $s \twoheadrightarrow_B t$ holds iff $s \rightarrow^k t$ form some $k \geq 0$.

With these semantics, let $M = \langle S, \pi, R_1, ..., R_m \rangle$ and $s \in S$ Let φ be an epistemic formula in the language L_{KC} for such formulas it is necessary to extend the language L_K with two clauses:

1. $(M,s) \vDash E_B\varphi \iff (M,t) \vDash \varphi$ for all $s \longrightarrow t$
2. $(M,s) \vDash C_B\varphi \iff (M,t) \vDash \varphi$ for all $s \twoheadrightarrow_B t$

The first clause implies that $E_B\varphi$ holds s if, and only if, φ holds in all worlds t that any agent can reach from s. The second clause implies that $C_B\varphi$ holds in s if, and only if, φ holds in all worlds t that any combination of agents can reach from s in zero or more steps.

7.3 Public Announcements and Common Knowledge

The epistemic logic language that the model checker supports is the language $L_{KC[]}$ which is the combination of the language of public announcements $L_{K[]}$, introduced in Section 2.2, and the language of common knowledge L_{KC}, explained in Section 7.1. The language $L_{KC[]}$ is given by:

$$\varphi ::= p \mid \neg\varphi \mid (\varphi \wedge \varphi) \mid K_i\varphi \mid C_B\varphi \mid [\varphi]\varphi$$

With this language we can create very rich formulas to test in the model checker and it gives us the possibility to combine formulas with common knowledge and public announcements. The interaction between announcement and knowledge can be formulated in various ways, however we have to be careful while trying to create equivalences. Take these two formulas as an example:

- $[\varphi]\neg\psi \leftrightarrow (\varphi \rightarrow \neg[\varphi]\psi)$
- $[\varphi]K_i\psi \leftrightarrow (\varphi \rightarrow K_i[\varphi]\psi)$

These formulas show that $[\varphi]\neg\psi$ and $[\varphi]K_i\psi$ are equivalent to $\varphi \rightarrow \neg[\varphi]\psi$ and $\varphi \rightarrow K_i[\varphi]\psi$ respectively. A full proof of the derivation can be found in [29, Chapter 4]. The problem comes when we try to include common knowledge in a similar equivalence:

- $[\varphi]C_B\psi \leftrightarrow (\varphi \to C_B[\varphi]\psi)$

This formula is invalid. In order to show why, consider the instance $[p]C_{1,2}q \leftrightarrow (p \to C_{1,2}[p]q)$ and the model shown in Figure 7.1.

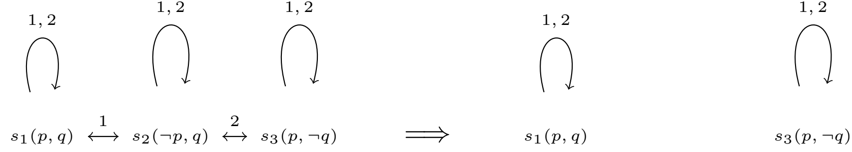

FIGURE 7.1: A model before and after the announcement of p. Taken from [29]

The left side of the formula is true, and the right side false, in state s_1 of the model on the left. We have that $[p]C_{1,2}q$ holds in state s_1 because $(M \mid p, s_1) \vDash C_{1,2}q$. The model after the announcement consists of two disconnected states. $(M \mid p, s_1) \vDash Cq$ holds because q is true in s_1 and there is no other reachable state for any agent.

On the other hand, $p \to C_{1,2}[p]q$ does not hold in state s_1, because $(M, s_1) \vDash p$ but $(M, s_1) \nvDash C_{1,2}[p]q$. The second part is due the fact that $s_1 \twoheadrightarrow_{1,2} s_3$ or $s_1 \to_1 s_2 \to_2 s_3$ and $[q]p$ will not hold in s_3. When we evaluate q in $(M \mid p)$ we are in another state that is disconnected where q is false.

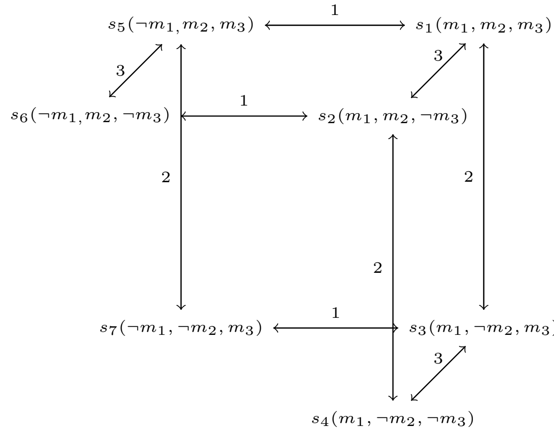

FIGURE 7.2: Muddy children model after the first announcement. Taken from [29]

Figure 7.2 shows the muddy children model after the first announcement and only three children. As mentioned in Section 5.2, the first announcement made by the father is "At least one of you is muddy" and it can be written as $(m_1 \vee m_2 \vee m_3)$. After the announcement, this formula became common knowledge for all three agents and it can be checked with the formula $C_{1,2,3}(m_1 \vee m_2 \vee m_3)$ in any state of the model. It is very important to keep in mind that it is not always the case that a formula will become common knowledge after it is announced as explained in Section 7.3.

The relation between announcement and common knowledge cannot be expressed in an equivalence. Fortunately there is a way to get common knowledge after an announcement. This principle and its proof are mentioned in [29, Chapter 4]. The principle goes like this:

Public announcement and common knowledge: If $\chi \to [\varphi]\psi$ and $(\chi \wedge \varphi) \to E_B\chi$ are valid, then $\chi \to [\varphi]C_B\psi$ is valid.

In order to explain the first premiss of this principle note that $\chi \to [\varphi]\psi$ is equivalent to $\chi \to (\varphi \to [\varphi]\psi)$ which is equivalent to $(\chi \wedge \varphi) \to [\varphi]\psi$. What this premiss means is that, given an arbitrary state in a model where χ and φ hold, if we restrict the model only to φ states, if we do a φ-step, then ψ holds in the resulting epistemic state.

What the second premiss implies is that, given an arbitrary state in the domain where χ and φ hold, if we do an arbitrary a-step in the domain, then we always reach an epistemic state where χ holds.

For the conclusion, note that $\chi \to [\varphi]C_B\psi$ is equivalent to $(\chi \wedge \psi) \to [\varphi]C_B\psi$. According to van Ditmarsch, van der Hoek and Kooi, the conclusion therefore says that, given an arbitrary state in the domain where χ and φ hold, if we do a φ-step followed by a B-path, we always reach a ψ-state.

This induction implies that, if we do a φ-step followed by an a-step, the path can be completed, because the premisses ensure that we can reach a state so that we can, instead, do the a-step first, followed by the φ-step.

In order to make things more simple, let us look at an example. Imagine a model M with an arbitrary number of states. In this model there may be states in which χ is true and some others in which $\neg\chi$ is not true as shown in Figure 7.3.

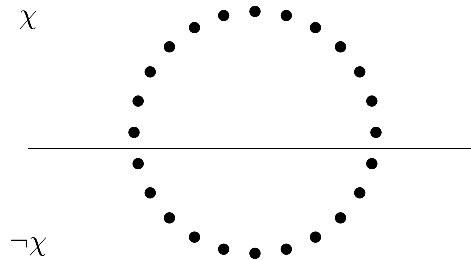

FIGURE 7.3: Model with χ and $\neg\chi$ worlds

Then, let us take only the worlds in which χ is true and then announce φ. The result of this announcement is shown in Figure 7.4

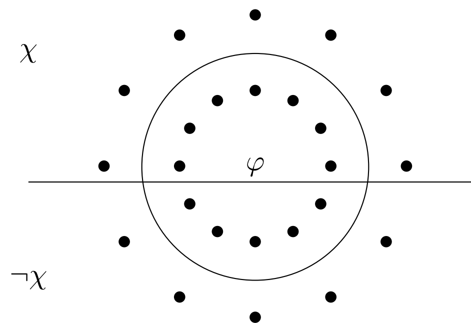

FIGURE 7.4: Model with χ and $\neg\chi$ worlds after the announcement of φ

In each surviving state of the model, we can say that φ holds and, if we take a state in which χ holds too and then follow a path defined by a group of agents B we will always

reach a state in which ψ holds thanks to the premisses of the 'Public announcement and common knowledge' principle. In other words, every B-path in a model M that runs along states where φ holds also runs along states where $[\varphi]\psi$ holds.

7.4 Common Knowledge in the Model Checker

This Section will show how to model common knowledge in the model checker. Consider the model shown in Figure 7.5.

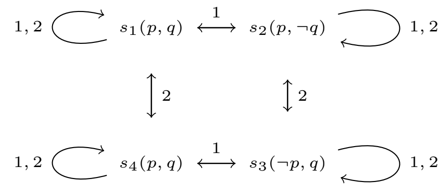

FIGURE 7.5: Example of model to test common knowledge.

This model can be inserted in the model checker with the next function:

```
1   m = model{
        [
3           (p,q) (p,!q) (!p,q) (p,q)
        ],[
5           (1~2,2~1,3~4,4~3)
            (1~4,4~1,2~3,3~1)
7       ]};
```

We want to check if it is possible to acquire common knowledge in a state of the model. For this we will use the formulas $C_B q$, $(C_B q \wedge C_B p)s$ and $C_B(p \vee \neg p)$. We insert these formulas as follows:

```
1   f1 = formula{Cq};
    f2 = formula{Cq&Cp};
3   f3 = formula{C(p|!p)};

5   check{m,f1};
    check{m,f2};
7   check{m,f3};
```

As in previous examples each one of the formulas can be tested in specific states by adding the state number after the model variable between square brackets. The output of this program shows the states in which these formulas hold.

```
1   'f1' holds for 'm' in:
        state 1 = false
3       state 2 = false
        state 3 = false
5       state 4 = false

7   'f2' holds for 'm' in:
        state 1 = false
9       state 2 = false
        state 3 = false
11      state 4 = false

13  'f3' holds for 'm' in:
        state 1 = true
15      state 2 = true
        state 3 = true
17      state 4 = true
```

The formula $C_B(p \vee \neg p)$ holds in all the states because it does not matter in which state we decide to star. All the states that are reachable from that first state contain either p or $\neg p$.

Now, for public announcements and common knowledge, consider the example shown in Section 7.3 in Figure 7.1. In this example, it was mentioned that the formulas $[p]C_B q$ and $(p \rightarrow C_B[p]q)$ are not equivalent. This can be easily checked with the next program.

```
1  ckandpa = model{
       [
3          (p,q) (!p,q) (p,!q)
       ],[
5          (1~2,2~1)
           (2~3,3~2)
7      ]};

9  f1 = formula{[p]Cq};
   f2 = formula{(p->C[p]q)};
11
   check{ckandpa ,f1};
13 check{ckandpa ,f2};
```

If these formulas were equivalent, they should have yielded the same result, however, as it is shown in the next lines, we obtain different results after running both formulas in the same model.

```
1  'f1' holds for 'ckandpa' in:
        state 1 = true
3       state 2 = false
        state 3 = false
5
   'f2' holds for 'ckandpa' in:
7       state 1 = false
        state 2 = true
9       state 3 = false
```

To show how to test the knowledge of a group, consider the muddy children model after the first announcement obtained in Section 5.2 and shown in Figure 7.2. The formulas that will be tested are $C_B(m_1 \vee m_2 \vee m_3)$, $C_{1,3}(m_2)$, $C_{2,3}(!m_1)$ and $C_{1,2}(m_3) \wedge C_{1,3}(!m_2)$. In order to insert the group of agents in the model checker, we have tor write the underscore symbol _ after the C operator followed by the agent number we want to include in the group as shown below.

```
1  group1 = formula{C(m1 | m2 | m3)};
   group2 = formula{C_1_3(m2)};
3  group3 = formula{C_2_3(!m1)};
   group4 = formula{C_1_2(m3) | C_1_3(m2)};
5
   check{muddy1,group1};
7  check{muddy1,group2};
   check{muddy1,group3};
9  check{muddy1,group4};
```

We are testing these formulas in a previously defined model of the muddy children after the first announcement. Check Section 5.2 for more details. The result of this program is as follows:

```
1  'group1' holds for 'muddy1' in:
        state 1 = true
3       state 2 = true
        state 3 = true
5       state 4 = true
        state 5 = true
7       state 6 = true
        state 7 = true
9
   'group2' holds for 'muddy1' in:
11      state 1 = true
        state 2 = true
```

```
13        state 3 = false
          state 4 = false
15        state 5 = true
          state 6 = true
17        state 7 = false

19    'group3' holds for 'muddy1' in:
          state 1 = false
21        state 2 = false
          state 3 = false
23        state 4 = false
          state 5 = true
25        state 6 = true
          state 7 = true
27
      'group4' holds for 'muddy1' in:
29        state 1 = true
          state 2 = true
31        state 3 = false
          state 4 = false
33        state 5 = true
          state 6 = true
35        state 7 = false
```

The first formula implies that it is common knowledge that at least one of the children is muddy. This formula involves all of the agents, in this case the children, and all of them consider possible that at least one of them is muddy in all the states.

The second and third formula involves just two of the agents. In this case since one of the agents is left out, his relations are not considered when building the knowledge path. Therefore, the path will not reach as many states as if all the agents where involved. This can help to achieve common knowledge a bit easier.

The last formula shows that it is possible to include two different groups of agents in the same formula. The order in which the agents of the group are listed is irrelevant.

Common knowledge can also be tested in unbounded models. Consider the model shown in Figure 7.6

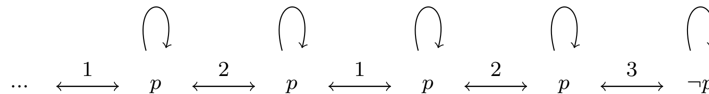

FIGURE 7.6: unbounde models and common knowledge

One of the possible ways to create models similar to the one shown in the previous figure is with the regular expression $([p]\langle 1\rangle; [p]\langle 2\rangle)*; [p]\langle 3\rangle; [!p]$. As with the normal models, we can define formulas using common knowledge and test thems in unbounded models. The formulas that will be tested in this model are $C_B p$, $\neg C_B p$, $C_B \neg p$ and $C_{1,2} p$.

```
1     unbounded = regex{([p]<1>;[p]<2>)*;[p]<3>;[!p]};

3     u1 = formula{Cp};
      u2 = formula{!Cp};
5     u3 = formula{C!p};
      u4 = formula{C_1_2 p};
7
      check{unbounded ,u1};
9     check{unbounded ,u2};
      check{unbounded ,u3};
11    check{unbounded, u4};
```

The output of this program shows whether it is possible for these formulas to hold at some point giving the regular expression. In the case of the first and third formula, the model checker shows that it is not possible for these formulas to hold in the first state

of the collection of models defined by the regular expression. In the case of the second and fourth formula, the model checker shows the shortest model in which the formula can hold.

```
1  The formula 'u1' cannot hold in the first state of models defined by regular expression 'unbounded'

3  The formula 'u2' can hold in the first state of model =>
       $temp$=>
5       Agent: 3 [s1-s1,s1-s2,s2-s2,s2-s1]
        States[
7           s1(p)  s2(!p)
       ]
9
   The formula 'u3' cannot hold in the first state of models defined by regular expression 'unbounded'
11
   The formula 'u4' can hold in the first state of model =>
13     $temp$=>
        Agent: 1 [s1-s1,s1-s2,s2-s2,s2-s1,s3-s3,s4-s4]
15      Agent: 2 [s1-s1,s2-s2,s2-s3,s3-s3,s3-s2,s4-s4]
        Agent: 3 [s1-s1,s2-s2,s3-s3,s3-s4,s4-s4,s4-s3]
17      States[
            s1(p)  s2(p)  s3(p)  s4(!p)
19     ]
```

When testing formulas in an unbounded model, the length of the model depends on the knowledge depth of the formula we want to check. The knowledge depth of a formula can be calculated by using the nested knowledge K operators in the formula. However this is not the case with the common knowledge operator C.

When the model checker is asked to check in an unbounded model a formula that contains common knowledge, it starts testing with the shortest model possible and in case of failure, it tests it in the next model. Each run is saved and the model checker will continue with this operation until it decides that the result will not change no matter how large the model can grow.

The model checker analyses which parts of the model are not allowing the formula to become true and checks if it is possible to change it using the rules of the regular expression, for instance, let us use the regular expression $[p]\langle 1,2\rangle *; ([!p]\langle 1,2\rangle \mid [p]\langle 1,2\rangle)$ and the formula $C_B p$. The regular expression can create models that start with an unbounded number of p states and end in either a p or a $\neg p$ state.

When checking the formula $C_B p$, the model checker first analyses the models that start with an unbounded number of p states and end with $\neg p$. In these models the formula will never hold because it is always possible to reach the $\neg p$ state. The model checker analyses the regular expression and checks which part of it or which operator is not allowing the formula not to hold, in this case, $[!p]\langle 1,2\rangle$, then the model checker 'looks' if it is possible to change the state that is not allowing the formula to hold. In this case, since we have the union operator, it is possible to choose a model in which the formula can hold.

Another example will be the regular expression $[p]\langle 1,2\rangle; [!p]\langle 1,2\rangle > *; [p]\langle 1,2\rangle *$ and the same formula $C_B p$. This time, the part of the regular expression that does not allow the formula to hold is $[!p]\langle 1,2\rangle > *$. The model checker analyses this situation and decides to, use the operator zero times. By doing this, the formula $C_B p$ can hold.

If it is not possible to get a model that falls into the specifications of the regular expression that can achieve common knowledge, the model checker will show a message similar to the ones seen in the previous example.

The last example will show if it is possible for the two generals to acquire common knowledge and synchronize their attacks. We will use the regular expression defined at

the end of Section 6.6. In order for the generals to start a coordinated attack, common knowledge of m is necessary. This is modelled with the formula $C_B m$.

```
1  generals = regex{([m]<1>|[m]<2>;[m]<1>);([m]<2>;[m]<1>)*;[!m]};
   success1 = formula{C m};
3  success2 = formula{C_1_2 m};
   check{generals,success1};
5  check{generals,success2};
```

Unfortunately for the generals they will not be able to synchronize their attacks since, no matter how big the model grows, there will always be a reachable state that contains $\neg m$. This is consistent with the results obtained in [12], [1] and [29, Chapter 2] that show that common knowledge cannot be achieved by the generals in this way.

```
1  The formula 'success1' cannot hold in the first state of models defined by regular expression 'generals'

3  The formula 'success2' cannot hold in the first state of models defined by regular expression 'generals'
```

Chapter 8

Examples

In previous chapters we have shown examples of how to define formulas and test them in our models. We solved the muddy children puzzle with 3 agents, the wise man riddle and the two generals problem. This chapter includes extra examples of models and formulas that are more complex and how can we try them in the model checker.

8.1 Example 1

The first example consists of two parts and it was taken from [21, Chapter 2] Exercise 2.2.10 and goes like this:

1. Let $E^k\varphi$ be defined by $E^0\varphi = \varphi$, $E^{k+1}\varphi = E(E^k\varphi)$. Let M be an $S5_m$ Kripke model with n states. Prove that $M \vDash E^n\varphi \leftrightarrow C\varphi$.
2. Devise an $S5_m$ Kripke model M and a formula φ such that for every $n \geq 0$ it holds that $M \nvDash E^n\varphi \leftrightarrow C\varphi$.

For the first part of this example let us assume we have 2 agents in our model and make $n = 3$ and $\varphi = p$. The model for this example is shown in Figure 8.1. The formula we would like to test is $EEEp \leftrightarrow Cp$. At the present time, the model checker does not support the E operator so we can rewrite the formula in terms of the K operator as shown below:

$$K_1(K_1(K_1p \wedge K_2p) \wedge K_2(K_1p \wedge K_2p)) \wedge K_2(K_1(K_1p \wedge K_2p) \wedge K_2(K_1p \wedge K_2p)) \leftrightarrow Cp$$

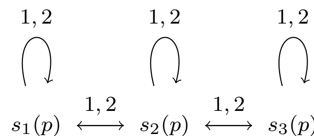

FIGURE 8.1: Model exercise 1.1

We can input this model and the formula in the model checker as follows:

```
meyer = model{
    [(p)(p)(p)],
    [
        (1~2,2~1,2~3,3~2)
        (1~2,2~1,2~3,3~2)
    ]};

exf1 = formula{
    K1(K1(K1 p & K2 p) & K2(K1 p & K2 p)) & K1(K1(K1 p & K2 p) & K2(K1p & K2 p)) <-> C p
};

check{meyer ,exf1};
```

As the number of states in the model grows, the knowledge becomes deeper, allowing us to reach one more state. This example was made with a knowledge depth of 3 but it is possible to increase it with a more complex formula. The output of the model checker for $n = 2$ is shown below.

```
'exf1' holds for 'meyer' in:
     state 1 = true
     state 2 = true
     state 3 = true
```

This result shows that it does not matter in which state of the model we are, we will always reach a p state.

For the second part of this example we need to find a model in which the formula $E^n\varphi \leftrightarrow C\varphi$ will not hold. Consider again $n = 3$, the instance $\varphi = p$ and the previously defined formula *exf1*. The model that we will be using is shown in Figure 8.2.

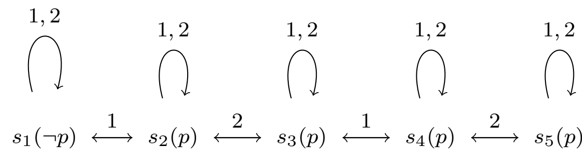

FIGURE 8.2: Model exercise 1.2

The model, the formula and their output are shown below:

```
meyer2 = model{
    [(!p)(p)(p)(p)(p)],
    [
        (1~2,2~1,3~4,4~3)
        (2~3,3~2,4~5,5~4)
    ]};

check{meyer2 ,exf1};

exf1 'holds for 'meyer2' in:
     state 1 = true
     state 2 = true
     state 3 = true
     state 4 = true
     state 5 = false
```

We found a model in which the formula $E^3p \leftrightarrow Cp$ does not hold in state s_5. The formula does not hold in the state s_5 because, with E^3p we are not able to reach a $\neg p$ state from s_5, therefore E^3p is *True* in S_5, however, Cp will always reach a $\neg p$-state, meaning that the result of checking the formula is *True* $\leftrightarrow$ *False* which results in *False*. The formula holds in the rest of the states since E^3p can always reach the $\neg p$-state, therefore, in the rest of the states we get *False* $\leftrightarrow$ *False* which results in *True*.

This example was created using a fixed n, more specifically $n = 3$. The second part of this example is actually asking for a model in which the formula $E^n\varphi \leftrightarrow C\varphi$ will not hold for every n. The answer for this, is a model with an infinite number of states. The model checker at the present time cannot work with infinite models since, even though an unbounded model can be very large, it still contains a finite number of states. Nevertheless the example shows that it is possible to define such a model. Future versions of the model checker will deal with infinite models, see Section 9.2.

8.2 Example 2

This example implements the principle of common knowledge and public announcements listed in Section 7.3. For this exercise consider the model shown in Figure 8.3 and for the formulas χ, φ and ψ consider the instances $\chi = K_1\neg p$, $\varphi = \neg K_2 q$ and $\psi = K1(K_2(p \wedge \neg q))$.

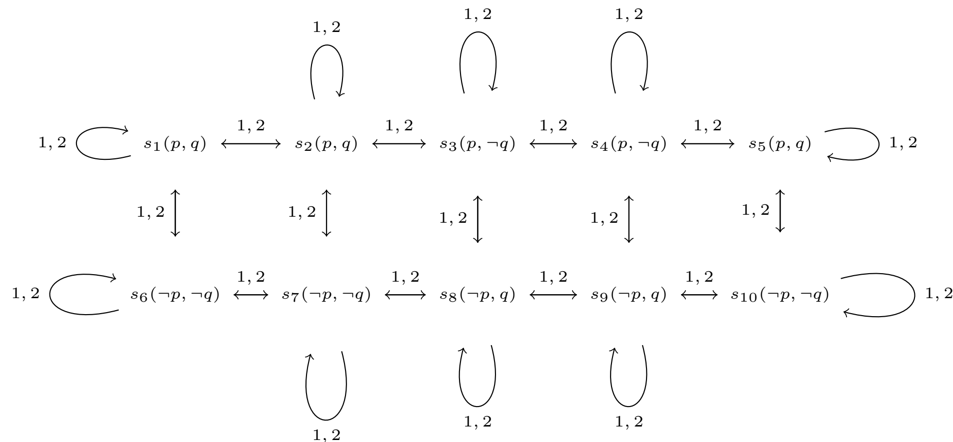

FIGURE 8.3: Model exercise 2

The principle says that, if $\chi \rightarrow [\varphi]\psi$ and $(\chi \wedge \varphi) \rightarrow E_B\chi$ are valid, then $\chi \rightarrow [\varphi]C_B\psi$ is valid, which means that the formula $(\chi \rightarrow [\varphi]\psi) \wedge ((\chi \wedge \varphi) \rightarrow E_B\chi)$ and the formula $\chi \rightarrow [\varphi]C_B\psi$ must yield the same result.

Below it is shown how we can input these formulas, with the appropriate substitutions mentioned before, into the model checker.

```
1   common = model{
        [
3           (p,q)(!p,!q)
            (p,q)(!p,!q)
5           (p,!q)(!p,q)
            (p,!q)(!p,q)
7           (p,q)(!p,!q)
        ],[
9           (1~2,1~6,2~1,2~3,2~7,3~2,3~4,3~8,4~3,4~5,4~9,5~4,5~10)
            (1~2,1~6,2~1,2~3,2~7,3~2,3~4,3~8,4~3,4~5,4~9,5~4,5~10)
11      ]};

13  exf1 = formula{(K1!p->[!K2q](K1K2(p&!q)))&((K1!p&!K2q)->K1K1!p & K2K1!p)};
    exf2 = formula{K1!p->[!K2q]C(K1K2(p&!q))};
15
    check{common,exf1};
17  check{common,exf2};
```

Below is the result of executing the previous program. It can be seen that both formulas are equivalent since both of them yielded the same result, therefore, the principle of common announcements and common knowledge can be checked in the model checker.

```
'exf1' holds for 'common' in:
     state 1 = true
     state 2 = true
     state 3 = true
     state 4 = true
     state 5 = true
     state 6 = false
     state 7 = true
     state 8 = false
     state 9 = true
     state 10 = false

'exf2' holds for 'common' in:
     state 1 = true
     state 2 = true
     state 3 = true
     state 4 = true
     state 5 = true
     state 6 = false
     state 7 = true
     state 8 = false
     state 9 = true
     state 10 = false
```

8.3 Example 3

This example is about public announcements in unbounded models. A nice thing about this combination is that public announcements can help us to "get rid of" the star operator, making the models more simple. Take, for instance, the unbounded model $([p]\langle 2,3\rangle;[p]\langle 1\rangle)*;([!p]\langle 2\rangle;[!p]\langle 1\rangle)*;[!p]$. This regular expression will create models similar to the one shown in Figure 8.4.

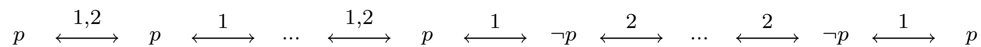

FIGURE 8.4: Model exercise 3, based on [16]

If we announce the formula $\neg p$ and then check if $\neg p$ became common knowledge, we can use the formula $[\neg p]C\neg p$ in the model checker as shown below.

```
example3 = regex{([p]<2,3>;[p]<1>)*;([!p]<2>;[!p]<1>)*;[p]};
exf1 = formula{[!p]C!p};
check{example3,exf1};
```

If there is the possibility that the formula $[\neg p]C\neg p$ can hold in the first state of a model inside the collection of models defined by the regular expression, the model checker will show the shortest one. Below is the result of running the previous program.

```
The formula exf1 can hold in the first state of model =>
    $temp$=>
     Agent: 1 [s1-s1,s2-s2,s2-s3,s3-s3,s3-s2]
     Agent: 2 [s1-s1,s1-s2,s2-s2,s2-s1,s3-s3]
     States[
        s1(!p)  s2(!p)  s3(p)
    ]
```

In case we want to verify that the result is actually correct, we can insert the obtained model manually and then run the formula on it.

```
verify = model{
    [(!p)(!p)(p)],
    [
        (s1~s1,s2~s2,s2~s3,s3~s3,s3~s2)
        (s1~s1,s1~s2,s2~s2,s2~s1,s3~s3)
    ]};

exf1 = formula{[!p]C!p}
check{verify,exf1};
```

The advantage of verifying whether the model given by the regular expression was correct, is that we can check if it is possible that the formula can hold in state other than the first one as the result of the program shows.

```
'exf1' holds for 'verify' in:
     state 1 = true
     state 2 = true
     state 3 = false
```

Also, it is possible to see how the model looks after the announcement if we use the function *announce* instead of running the formula directly.

```
exf2 = formula{!p};
verifyannounce = announce{verify ,exf2};

verifyannounce=>
     Agent: 1 [s1-s1,s2-s2]
     Agent: 2 [s1-s1,s1-s2,s2-s2,s2-s1]
     States[
        s1(!p)  s2(!p,)
    ]
```

Chapter 9

Conclusions and Future Work

9.1 Conclusions

Verification of multi-agent systems has become an active area of research. For many logics of interest, model checking can be efficiently automated. This has led to widespread interest in model checking as a technique for verifying properties of systems and the development of model checking tools.

The success of model checking in the computer aided verification community has led to a growth of interest in the use of model checking in artificial intelligence. Automated verification technologies are increasingly relevant for safety and reliability of autonomous systems.

Epistemic logic has grown from its philosophical beginnings to find diverse applications in computer science as a mean of reasoning about the knowledge and belief of agents. In this project we have provided a model checking technique to perform unbounded model checking on an epistemic language.

Unbounded models can be helpful to represent a very large collection of models with a single regular expression, therefore, it is possible to analyse the models in the collection in a simple way in a single run. The model checker will try a formula in the shortest model possible defined by the regular expression. If the formula holds in this model, the analysis will stop and the model checker will output this model. If the formula did not hold, the model checker will try the next model in the collection. This process is repeated indefinitely either until the model checker detects that the formula holds in a model or until it realises that it never holds, as explained in Section 7.4.

It is important to realize that the collection of models defined by a regular expression can be very large and potentially infinite. On the other hand, all the models inside the collection can be formed by a great number of states but still the number of states is finite. This allows the model checker to always find a result when checking a formula and the result will be either a model in which the formula holds or a message saying that the formula will never hold.

Another advantage of using unbounded models is that a solution can be found before checking all the models in the collection. If an acceptable solution is found in one model

of the collection, the model checker will show this result and will stop the verification, making the process more efficient.

With the help of the unbounded models we can say how expressive common knowledge is. Some epistemic operators can be expressed in terms of other operators as seen in Section 8.1 where the everybody knows operator E can be expressed in terms of K. However, not all epistemic operators can be expressed in other terms, an example of this is the common knowledge operator C.

The common knowledge operator does not have an equivalent formula that uses the K or any other operator, that is one of the reasons that when we give the definition of common knowledge we use the semantic definition. In [29, Chapter 8] van Ditmarsch, van der Hoek and Kooi, explain the concept of expressivity. They mention that two formulas are equivalent if and only if they are true in the same states. Using unbounded models we can see that, when using common knowledge in a formula, there is no formula that is equivalent to it using only K operators. Therefore, the language of $S5_m^C$ is more expressive than the language $S5_m$

Unbounded models will become a useful tool when testing multi-agent systems. They can provide a simple way when defining very large models providing the most simple way in which a particular problem can be solved.

9.2 Future Work

Since this project is the first one to work with unbounded models, there is still a lot of work to do. One of the most important features to improve in the model checker, will be to increase the number of classes and structures that unbounded models can define. This version of the model checker can only analyse linear unbounded models. Future versions will introduce more elaborated regular expressions with extra operators that will allow the user to create unbounded models that are not linear.

Currently, the model checker can only test formulas in the first state of an unbounded model. Future versions will allow the user to check his formulas in the nth state of an unbounded model. Also, after checking a formula in an unbounded model, the model checker returns only the smallest model in which the formula can hold. It may be the case that the user does not want the smallest model but a model of a specific length so it can be closer to a real life situation. A future version will allow the users to submit the length of the model they want as an output.

After making the unbounded models more solid, the next step will be the support for infinite models that have a finite representation or models that have an infinite number of agents, so problems such as the second question in Section 8.1 can be answered by the model checker. It is important that infinite models can be finitely represented, either by a regular expression or by a different technique, so that it can be inserted into the model checker.

Other model checkers have the capability to give a counter-example when asked. This model checker lacks this feature. Counter-examples can be very helpful when solving logic problems. Therefore, a future version of the model checker will be capable to give a counter-example if requested by the user.

Then again, after the model checker is more solid and stable with all the previous features in the domain of the $S5$ logic, other well known logics such as $S4$, G, etc., can be implemented. Being able to implement other logics, the range of the problems that the model checker can solve will increase.

The previous features are related to logic itself so that the model checker can help logicians to model and solve problems more easily. The next features are improvements to help the user to analyse the results and to make the coding over the model checker more friendly. The first of these features is the implementation of the Everybody Knows operator E, the belief operator B, the implicit knowledge operator I and the Goal operator G. Some of them, such as E, can be inserted in the model checker using the K operator equivalent as seen in Chapter 8, but the formulas can get very complicated so it will be good to have a shortcut.

Sometimes when analysing the output models of the model checker, they may not be easy to read. For this reason, it will be possible to export the models to DOT (graph description language), that can later be imported to a DOT interpreter such as Gvedit, and will change the input into an image of the model making it easier to analyse.

Finally, another thing that can be improved is the way to submit programs into the model checker. Currently the programs must be written in the application itself, but many users prefer to upload a file. Future versions will allow users to insert the programs in the form of a text file.

Appendix A

The Interpreter

The Interpreter is the part of the program that checks the instruction for the model checker are valid, looks for invalid characters and ensures that the parameters sent to each formula, are the correct type. The interpreter consists of three parts:

1. Lexicon analyser;
2. Syntactic analyser;
3. Semantic analyser.

Next, each part of the Interpreter is explained.

A.1 Lexicon analyser

This analyser reads character by character of the submitted code creating a table of tokens that latter will be the input parameter for the syntactic analyser. These tokens are a numeric interpretation of the read characters. In case the analyser finds a string, it will check if it is a reserved word for the model checker or a variable.

The valid characters for the Interpreter alphabet are:

{ } [ ] , ; = ! ( ) & | - > < K C ~ _ + - * /

In addition to the previous characters all the letters from *a* to *z* and the digits from 0 to 9 are also included in the alphabet. Table A.1 shows de numerical equivalence of the lexemes defined by the analyser.

In case the analyser finds an invalid character or an invalid string, it will throw an exception.

```
1    throw new LexicException("Lexic Error: The character:
     '"+invalid+"' Does not exist in the alphabet.");
```

Once the characters are validated and the table of tokens is created, the table is sent to the syntactic analyser.

Lexeme	#	Lexeme	#	Lexeme	#
String	1	Number	2	{	3
}	4	[	5	]	6
,	7	;	8	=	9
<=	10	>=	11	!	12
(	13	)	14	&	15
|	16	->	17	<->	18
K	19	C	20	~	21
+	22	-	23	*	24
/	25	<	26	>	27
_	28	formula	29	model	30
announce	31	check	32	print	33
delete	34	longmodel	35	propositions	36
sequence	37	restriction	38	relations	39
longformula	40				

Table A.1: Token lexemes

A.2 Syntactic analyser

With the table of tokens as input, the syntactic analyser will read the tokens one by one. The analyser will check if the tokens were written in the right order. To do this, the analyser will use an automaton to validate the submitted code. Figures A.1, A.2, A.3, A.4 and A.5 show the automaton required depending on the submitted function.

Figure A.1: Automaton for function *formula*

In case the table of tokens does not fulfil the requirements of the automaton, the interpreter will throw an exception.

```
throw new SintacticException("Line must start with
a variable or a function.");
```

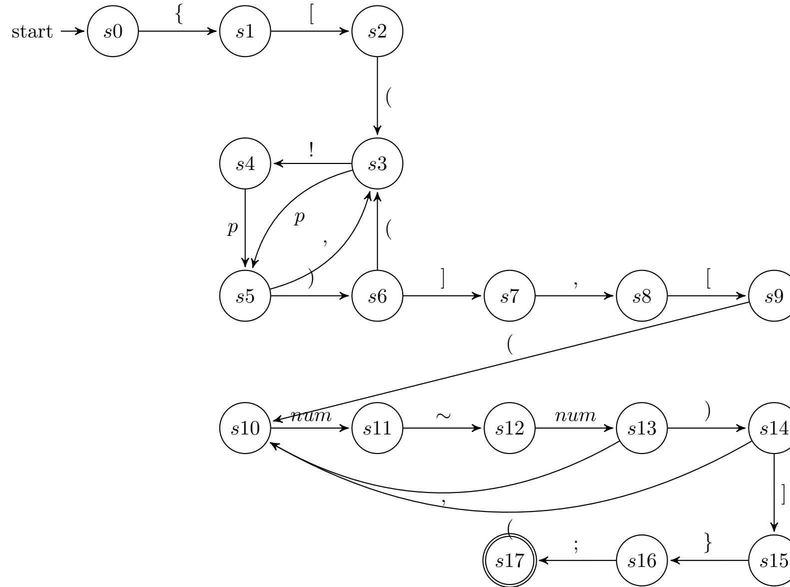

FIGURE A.2: Automaton for function *model*

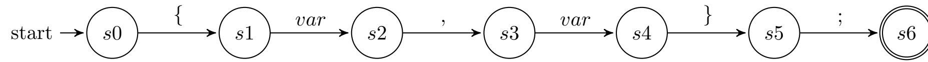

FIGURE A.3: Automaton for function *announce*

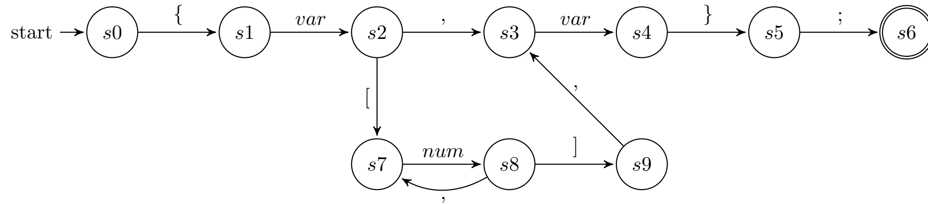

FIGURE A.4: Automaton for function *check*

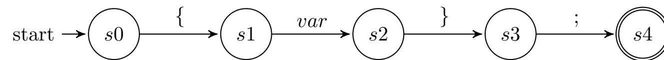

FIGURE A.5: Automaton for functions *print* and *delete*

A.3 Semantic analyser

The semantic analyser will validate if a certain variable has been declared or that the parameters for a certain function are the right ones, for instance, the function *announce*, requires a *model* variable and a *formula* variable. The analyser will check that, in fact, the parameters are a model and a formula and will throw an exception otherwise.

```
throw new SemanticException("The variable '"+var)
+"' is not a Model object.");
```

This analyser also takes a submitted logic formula in traditional notation, infix, and turns it into a reverse polish notation or infix. This process makes it easier to evaluate the logic formulas by removing the parentheses and executing the operations in the right

Operator	Priority
$< - >$	0
$- >$	1
$\vert$	2
&	3
!	4
K	4
C	4
[]	4

TABLE A.2: Operators priority for logic formulas

Operator	Priority
$<$	0
$>$	0
$>=$	0
$<=$	0
$=$	0
$+$	1
$-$	1
$*$	2
$/$	2

TABLE A.3: Operators priority for restrictions and relations

order. In order to do this, the operators were assigned with a priority value shown in table A.2.

When a formula is submitted, it looks like $K1(K2p|K2!p)$. After the infix notation is applied, it looks like $p2K!2Kp|1K$.

To verify if the formulas defined in the *restriction* and *relation*, parameters for *longformula* and *longmodel* functions described in Section 3.2, are well-formed, we have to follow a process similar to the one previously described. The table A.3 shows the priority of the operators for these functions.

Appendix B

Installation

This appendix includes the instructions to run the model checker in your own machine. For this you need:

- Java 5 or higher;
- Tomcat server 6;
- ModelChecker.war.

B.1 Installation in Windows

B.1.1 Steps for installing Java

1. Go to the download page of Java http://www.oracle.com/technetwork/java/javaseproducts/downloads/index.html
2. Select the version for Windows and click through the license acceptance. After two pages, you will be able to download the EXE file for installing Java on windows. Look for the SDK version.
3. Download and run the EXE installation program.
4. You will need to accept the license agreement again.
5. Use the suggested directory for installing Java.
6. You may use the remaining default settings for the installation.

B.1.2 Installing Tomcat

1. Go to the Tomcat download page http://tomcat.apache.org/download-60.cgi.
2. Scroll down until you see Binary Distributions.
3. Click on the link according to your windows system.
4. Extract the downloaded zip file in your preferred directory.

B.1.3 Setting the environment variables

1. Open the control panel under the start menu.
2. Double-click on System.
3. Click on the Advanced tab.
4. Click on the Environment Variables button.
5. Under System Variables, click on the New button.
6. For variable name, type: JAVA_HOME
7. For variable value, type the path where you installed Java.
8. Click OK to exit the dialogue window.
9. Click on the New button one more time.
10. For variable name, type: CATALINA_HOME
11. For variable value, type the path where you unzipped the tomcat server.
12. Click OK to exit the dialogue window.

B.1.4 Deploying the Model Checker

1. Download the Model Checker application from `http://modelcheckerjorge.appspot.com/Downloads.html`
2. Go to your tomcat installation directory and then go inside the webapps folder and paste ModelChecker.war
3. From the command line, go to your tomcat installation directory and then inside the bin folder.
4. Run the startup.bat file.
5. Go to your browser and type in the url http://localhost:8080/ModelChecker/

B.2 Installation on Linux

B.2.1 Steps for installing Java

1. Go to the download page of Java `http://www.oracle.com/technetwork/java/javaseproducts/downloads/index.html`
2. Select the version for Linux and click through the license acceptance. After two pages, you will be able to download the RPM file for installing Java on Linux.
3. From command line run the following command rpm -i java.rpm.
4. You will need to accept the license agreement.

B.2.2 Installing Tomcat

1. Go to the Tomcat download page http://tomcat.apache.org/download-60.cgi.
2. Scroll down until you see Binary Distributions.
3. Click on the link with the tar.gz extension.
4. Unpack the tomcat server with the command
   tar zxvf /tmp/apache-tomcat-6.x.x.tar.gz.
5. Copy the server and change the ownership with the next command
   chown -R tomcat.tomcat /var/lib/apache-tomcat-6.x.x

B.2.3 Setting the environment variables

1. From command line, use the following commands to set the environment variables:
   (a) export JAVA_HOME=path where you installed Java;
   (b) export CATALINA_HOME=path of your tomcat server;
   (c) export PATH=$JAVA_HOME/bin:$PATH.

B.2.4 Deploying the Model Checker

1. Download the Model Checker application from
   http://modelcheckerjorge.appspot.com/Downloads.html
2. Go to your tomcat installation directory and then go inside the webapps folder and paste ModelChecker.war
3. From the command line, run this command
   su -p -s /bin/sh tomcat $CATALINA_HOME$/bin/startup.sh.
4. Go to your browser and type in the url http://localhost:8080/ModelChecker/

Bibliography

[1] E. Akkoyunlu, K. Ekanadham, and R. Huber. Some Constraints and Tradeoffs in the Design of Network Communications. *SIGOPS Oper. Syst. Rev.*, 19:67–74, 1975.

[2] C. Baier. and J. P. Katoen. *Principles of Model Checking.* MIT Press, London, England, 2008.

[3] A. Baltag, L. Moss, and S. Solecki. The Logic of Public Announcements, Common Knowledge, and Private Suspicions. In *Proceedings of the 7th Conference on Theoretical Aspects of Rationality and Knowledge*, pages 43–56. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998.

[4] J. Barwise. Scenes and other Situations. *The Journal of Philosophy*, 78:369–397, 1981.

[5] E. M. Clarke and E. A. Emerson. Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic. In O. Grumberg and H. Veith, editors, *25 Years of Model Checking*, pages 196–215. Springer Berlin Heidelberg, London, UK, 2008.

[6] J.van Eijck. DEMO: A demo of Epistemic Modelling. In J. van Benthem, B. Lowe, and D. M. Gabbay, editors, *Interactive Logic. Selected Papers from the 7th Augustus de Morgan Workshop*, pages 303–362. Amsterdam University Press, 2007.

[7] H. Freudenthal. Formulation of the Sum and Product Problem. *Nieuw Archief voor Wiskunde*, 18:152, 1962.

[8] M. Friedell. On the Structure of Shared Awareness. *Behavioral Science*, 14:28–39, 1969.

[9] G. Gamow and M. Stern. *Puzzle-Math.* Macmillan, London, 1958.

[10] R. Goldblatt. *Logics of Time and Computation.* Center for the Study of Language and Information, Stanford, CA, USA, 1987.

[11] J. Gray. Notes on Data Base Operating Systems. In R. Bayer, R. Graham, and G. Seegmller, editors, *Operating Systems*, pages 393–481. Springer Berlin Heidelberg, 1978.

[12] J. Halpern and Y. Moses. Knowledge and Common Knowledge in a Distributed Environment. *J. ACM*, 37:549–587, 1990.

[13] J. Hopcroft, D. Ullman, and R. Motwani. *Introduction to Automata Theory, Languages, and Computation.* Pearson Education, United States of America, 2008.

[14] J. P. Katoen. *Concepts, Algorithms, and Tools for Model Checking.* IMMD, Nuernberg, Germany, 1999.

[15] S. C. Kleene. Representation of Events in Nerve Nets and Finite Automata. In *C. E. Shannon and J. McCarthy*, pages 3–42. Princeton University Press, 1956.

[16] L. B. Kuijer. APC vs ar, 2013.

[17] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4:382–401, 1982.

[18] E. J. Lemmon. The Lemmon Notes: An Introduction to Modal Logic. *American Philosophical Quarterly Monograph Series*, 11, 1977.

[19] A. Liu. Problem section: Problem 182. *Math Horizons*, 11:324, 2004.

[20] J. McCarthy. Formalization of Two P Involving Knowledge. In V. Lifschitz, editor, *Formalizing Common Sense: Papers by John McCarthy, Ablex Series in Artificial Intelligence.* Ablex Publishing Corporation, Norwood, New Jersey, 1990.

[21] J.-J. C. Meyer and W. van der Hoek. *Epistemic Logic for AI and Computer Science.* Cambridge University Press, Cambridge, United Kingdom, 1995.

[22] J. Plaza. Logics of Public Communications. In M. Hadzikadic M. L. Emrich, M. S. Pfeifer and Z.W. Ras, editors, *Proceedings of the 4th International Symposium on Methodologies for Intelligent Systems*, pages 201–216. North-Holland Press, 1989.

[23] J. P. Queille and J. Sifakis. Specification and Verification of Voncurrent Systems in CESAR. In M. Dezani-Ciancaglini and U. Montanari, editors, *International Symposium on Programming*, pages 337–351. Springer Berlin Heidelberg, 1982.

[24] H. P. van Ditmarsch. The Russian Cards Problem. *Studia Logica*, 75:31–62, 2003.

[25] H. P. van Ditmarsch and B. Kooi. The Secret of my Success. *Synthese*, 151:201–232, 2006.

[26] H. P. van Ditmarsch and J. Ruan. Model Checking Logic Puzzles. In *Quatrièmes Journées Francophones - MODÉELES FORMELS de l'INTERACTION.* Cahiers du Lamsade, 2007.

[27] H. P. van Ditmarsch, J. Ruan, and R. Verbrugge. Model Checking Sum and Product. In S. Zhang and R. Jarvis, editors, *AI 2005: Advances in Artificial Intelligence*, pages 790–795. Springer Berlin Heidelberg, 2005.

[28] H. P. van Ditmarsch, J. Ruan, and R. Verbrugge. Sum and Product in Dynamic Epistemic Logic. *Journal of Logic and Computation*, 3809:790–795, 2005.

[29] H. P. van Ditmarsch, W. van der Hoek, and B. Kooi. *Dynamic Epistemic Logic.* Springer, Dordrecht, The Netherlands, 2007.

[30] H. P. van Ditmarsch, W. van der Hoek, and B. Kooi. Dynamic Epistemic Logic and Knowledge Puzzles. In *Conceptual Structures: Knowledge Architectures for Smart Applications*, pages 45–58. Springer, 2007.