Indian Statistical Institute
Kolkata

Master's Thesis

Multi-Agent Systems: Model-checking in Logics of Protocols

Supervisor:
Dr. Sujata Ghosh

Author:
Pinaki Chakraborty
CS1824

A thesis submitted in fulfillment of the requirements
for the degree of Master of Technology
in
Computer Science

January 25, 2021

Declaration of Authorship

I, Pinaki Chakraborty, declare that this thesis titled, "Multi-Agent Systems: Model-checking in Logics of Protocols" and the work presented in it, submitted to Indian Statistical Institute, Kolkata, is a bonafide record of the study carried out in the partial fulfillment for the award of the degree Master of Technology in Computer Science. I confirm that:

- No part of this thesis has previously been submitted for a degree or any other qualification at this University.
- I have acknowledged all relevant sources of help.

Signed:

Date: January 25, 2021

Indian Statistical Institute,Kolkata
Kolkata, West Bengal, India, PIN - 700108

This is to certify that this thesis, "Multi-Agent Systems: Model-checking in Logics of Protocols" submitted by Pinaki CHAKRABORTY to Indian Statistical Institute, Kolkata, fulfills all the requirements of this Institute.

Signed:

Dr. Sujata Ghosh
Associate Professor
Computer Science Unit
Indian Statistical Institute, Chennai

Dedicated to My Parents

ACKNOWLEDGMENTS

It is a great pleasure for me to express my respect and deep sense of gratitude to my supervisor Dr. Sujata Ghosh, Computer Science Unit, Indian Statistical Institute, Chennai, for her vision, expertise, guidance, enthusiastic involvement, persistent encouragement, immense patience and unwavering faith in me during the planning and development of this work, and for her painstaking efforts in thoroughly going through and improving the manuscripts. I also gratefully acknowledge her contribution towards introducing me to the formal study of Logic.

I am highly obliged to Dr. Ansuman Banerjee, Advanced Computing and Microelectronics Unit, Indian Statistical Institute, Kolkata for initiating me to the field of Model Checking and encouraging me throughout this endeavor.

I am highly grateful to Dr. Malvin Gattinger, Department of Artificial Intelligence, University of Groningen for his valuable feedback in the early stages of this work.

I am highly obliged to my parents for their blessings and for providing me with their best.

I wish to express my obligation towards Dr. Malay Bhattacharya, Machine Intelligence Unit, Indian Statistical Institute, Kolkata for his moral support. I also wish to express my appreciation towards my friend Mr. Spandan Das for his encouragement and helping hand.

I would also like to express my deep and sincere thanks to all other persons whose names do not appear here, for helping me.

Finally, I am indebted and grateful to Indian Statistical Institute, Kolkata for hosting me as an M. Tech. student.

(P. C.)

ABSTRACT

A multi-agent system can often be described as a protocol based interacting system wherein the information flow, inter-agent communication and agent behavior can be naturally modeled with dynamic and epistemic logics which are different variants of modal logics. Such protocols may either be known beforehand to each agent or be unknown to any agent at the start. In the later situation, such protocols are called hidden protocols. When an agent learns of a hidden protocol, it is led to have some expectations about future observations and updates its knowledge of the state by matching its actual observations with the expected ones. In their paper "Hidden Protocols: Modifying our expectations in an evolving world", Hans van Ditmarsch, Sujata Ghosh, Rineke Verbrugge and Yanjing Wang studied how agents perceive such protocols and introduced the notion of epistemic expectation models and a propositional dynamic logic-style epistemic logic, Epistemic Protocol Logic for reasoning about knowledge via matching agents' expectations to their observations, updates of protocols and fact-changing actions. This is of particular interest to modeling scenarios where security aspects mandate knowledge of protocols to be hidden to some or all agents beforehand or at all times. In this project we will focus upon theory and implementation of a model checker for Epistemic Protocol Logic incorporating Epistemic Expectation Models and study formal methods towards a symbolic model checking approach to this end.

Contents

List of Figures

List of Acronyms/Abbreviations

BNF	Backus–Naur form
POL	Public Observation Logic
EPL	Epistemic Protocol Logic
ETL	Epistemic Temporal Logic
DEL	Dynamic Epistemic Logic
BDD	Binary Decision Diagram
EPDL	Epistemic Propositional Dynamic Logic
DFA	Deterministic Finite Automaton
ADT	Algebraic Data Type

List of Symbols

$\mathbb{N}$	The set of Natural numbers
$\mathbf{P}$	Powerset / Partition
$\mathcal{U}, \mathcal{V}$	Vocabulary

Chapter 1

Introduction

On October 29, 2018 a Boeing 737 MAX aircraft, *Lion Air Flight 610* crashed killing 189 people on-board, and on March 10, 2019 another Boeing 737 MAX aircraft, *Ethiopian Airlines Flight 302* crashed killing 157 people on-board. Investigations revealed that the crashes were caused by an improper design of the *The Maneuvering Characteristics Augmentation System*, which is an embedded software in the Boeing 737 MAX aircraft control system. In another incident, on May 6, 2010, the Dow Jones Industrial Index slumped nearly 1,000 points, wiping out more than $862 billion off the American stock market immediately. The government regulator identified the cause as an unintentional behavior of an automated algorithmic *trading strategy* employed by Waddell & Reed, a US mutual fund.

The above incidents highlight the importance of verification of a computerized system against its specifications to ensure proper behavior. To this end any system involving either a single computational agent (for example the aircraft embedded software) or a group of agents communicating with each other (for example in the algorithmic trading strategy scenario —buyer and seller agents are mutually interacting), needs to be formally modeled and checked against a formal specification of its behaviors in order to eliminate unintended 'side-effects'.

Usually, a logical formalism suitable to the corresponding problem domain is employed to model the structure as well as to specify the behavior and desired properties of a system. In general, the systems in consideration have finite states or finite state abstractions. For example, such systems include, but are not limited to, digital circuits, network protocols, embedded software etc. The task of modeling such systems and specifying their behaviors, along with designing frameworks to check the models against corresponding specifications, is known as *Model Checking* (also known as *Property Checking*).

In the late 1950's and early 1960's, Model Checking was associated chiefly with traditional hardware design. The methodologies, which were built upon *propositional logic*, were found to be quite suitable for verification of combinatorial circuits and their behaviors. But as hardware systems became more complicated with the introduction of large scale integrated circuits, the shortcomings of propositional

logic to describe properties of such systems became apparent.

In another direction, in the 1960's and 1970's, first order logic was used with considerable success in the domain of *Theorem proving* [Eme08]. This paradigm for verification, which is based upon proof-theoretic reasoning using formal axioms and inference rules, is oriented chiefly towards sequential programs. But these methods were not suitable to express parallelism characteristic to hardware systems consisting of very large scale integrated circuits and the emerging concurrent software systems. Moreover theorem proving in first-order logic is undecidable and hence often relies upon heuristics.

These underscored the need to avoid the difficulties with manual deductive proofs and to come up with methods that are decidable, thereby motivating alternative paradigms of formalism based upon modal logic in which the relevant property specifications (such as *correctness,fairness* etc) could be naturally expressed.

Amir Pnueli, in his seminal paper "The Temporal Logic of Programs", introduced *temporal logic* as one such suitable tool for modeling these concurrent systems [Pnu77]. Informally, any system comprising rules and symbolism for representation of and reasoning about propositions which are qualified with the modality of time is called a temporal logic.

The methodologies, which were developed upon temporal logic up to the early 1980's, suffered from the *state explosion problem*. As the number of state variables in the system increases, the size of state space grows exponentially. In the late years of the 1980's, symbolic methods for model checking that avoided explicit representation of the system state, thereby mitigating the state explosion problem, were developed [BCM+92].

In the subsequent decade, model checking based upon temporal logic was employed widely in many applications, such as improving real-time systems for ensuring the stability of buildings during earthquakes [CW96], and these techniques became industry standards. But the vast majority of existing tools are not quite useful for modeling situations or problems involving knowledge.

Another variant of modal logic, namely *epistemic logic* [FHMV95], which is chiefly concerned with logical approaches to knowledge, belief and related notions, naturally models such scenarios. While deeply rooted in philosophical traditions, the idea of a formal logical analysis of reasoning about knowledge is more recent. The impetus came from various research problems in the fields of economics, linguistics, artificial intelligence etc. It must be clarified that it is possible to incorporate epistemic operators in the framework of temporal logic [PR03].

Epistemic Logic is also particularly useful for modeling and analyzing situations

involving multiple agents. Informally stated, Multi-agent systems are those systems that include multiple autonomous entities with either diverging information or diverging interests, or both. A multi-agent system is typically characterized by communication between agents, which can change the epistemic states of agents. [Azi10]. Pragmatic concerns about the relationship between knowledge and action of an agent (or a group of agents) requires an analysis of change in the epistemic states of the corresponding agents.

Now let us consider a few examples that illustrate the concept of epistemic states of agents. In an affected area, a group of autonomous robotic agents are coordinating between themselves to accomplish a disaster response goal. Depending upon the degree of cooperation and nature of the goal, a formal analysis of the strategies being employed by the agents, requires different notions of epistemic states.

If the goal is to asses the damage incurred, for example, then the epistemic state of each agent incorporates the knowledge of the facts in its immediate surroundings. On the other hand, if the goal is to assist in evacuation and rescue operations, then the epistemic state of each agent must incorporate not only the knowledge of the facts in its immediate surroundings, but also the knowledge regarding the epistemic states of other agents in its vicinity.

As another motivating example, consider a bargaining situation. The seller of a car must consider what the potential buyer knows about the car's value. The buyer must also consider what the seller knows about what the buyer knows about the car's value, and so on.

The previous two examples illustrate that, different notions of epistemic states naturally arise in a multi-agent situation.

In some scenarios, it is imperative to consider the case wherein for a group of agents $\mathcal{I}$, each agent $i \in \mathcal{I}$, knows a proposition φ, and each agent i knows that another agent $j \in \mathcal{I}$ knows φ, and so on. In the literature, this is called *common knowledge of* φ in a group of agents $\mathcal{I}$.

For example, the convention that green light mandates the action "go" and red light for "stop" in a traffic control scenario, is presumably common knowledge among the drivers.

A related notion is that of a **protocol**. Informally stated, rules which prescribe the actions of agents with respect to their knowledge states are called protocols in the context of epistemic logic. In the traffic control scenario, for example, the convention that a green light mandates to perform the action "go" and red light mandates to perform the action "stop" is a protocol.

It is not necessarily the case that protocols are common knowledge to all agents. In some situations agents may have partial knowledge of protocols. In those cases, based upon its partial knowledge about the protocols and its observations of other agents' actions, an agent tries to reason about the facts in its world.

In this work, the methodologies for model checking different logic of protocols [vDGVW14], which are a part of the broad spectrum of epistemic logic, will be discussed and the implementation of a model checking tool for such logic will be presented. The work is organized and structured in the form of the following chapters:

i) **Introduction:** The need and scope for model checking variants of epistemic logic are discussed along with a presentation of the formal semantics of epistemic logic on Kripke Models. Dynamic Epistemic Logics are also briefly discussed to illustrate dynamic operations upon epistemic states.

ii) **Logic of Protocols:** The different logic of protocols, namely Public Observation Logic (POL) and Epistemic Protocol Logic (EPL) are discussed in detail.

iii) **Model Checking Logic of Protocols:** In this chapter, the problem of model checking is introduced formally and then a methodology for implementation of a model checking tool for POL and EPL is formulated.

iv) **Model Checking Tool for Logic of Protocols:** In this chapter, the details of a Haskell implementation of a model checking tool based upon the ideas in the previous chapter is described. An example of model checking EPL formulas with respect to a given scenario is also presented.

v) **Conclusions:** This chapter concludes the work and the scopes for future works are mentioned.

1.1 Introduction to Epistemic Logic

As mentioned before, epistemic logic is the study of knowledge (and related epistemic attitudes such as belief) using formal languages and mathematical models. In the literature, approaches rooted in *modal logic* are widely used to formalize epistemic logic.

In this section, we first very briefly discus the syntax and semantics of epistemic logic, and then in the next section, syntax and semantics of a related logic, called Dynamic Epistemic Logic, is briefly discussed.

We begin by defining some preliminary concepts. Let $\mathcal{I}$ be a finite set of agents, $\mathcal{I} = \{1, \ldots, n\}$. Let us further assume that a countably infinite set of atomic propositional variables exist. A finite subset of these variables is called a vocabulary and is denoted by $\mathcal{V}$.

Definition 1.1.1. *The language $\mathcal{L}_B(\mathcal{V})$ of Boolean formulas over a vocabulary $\mathcal{V}$ is given by the following BNF :*

$$\varphi ::= \top \mid p \mid \neg\varphi \mid \varphi \wedge \varphi \quad \textit{where } p \in \mathcal{V}$$

Throughout this work, the following connective symbols, $\bot \coloneqq \neg\top$, $\varphi \vee \psi \coloneqq \neg(\neg\varphi \wedge \neg\psi)$, $\varphi \rightarrow \psi \coloneqq \neg\varphi \vee \psi)$ and $\varphi \leftrightarrow \psi \coloneqq (\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi)$ are used.

Definition 1.1.2. *A Boolean assignment for a vocabulary $\mathcal{V}$ assigns to each atomic proposition a truth value. An assignment $\mathcal{S}$ is a function of the form $\mathcal{S} : \mathcal{V} \rightarrow \{True, False\}$.*

For a fixed vocabulary, $\mathcal{S}$ is identified with the subset of atomic propositions which are true in it. Thus $\mathcal{S} = \{p \in \mathcal{V} \mid \mathcal{S}(p) = True\}$. A formula, φ is termed satisfiable, if it is possible to find an assignment $\mathcal{S}$, such that φ is true over $\mathcal{S}$. It is denoted as $\mathcal{S} \vDash \varphi$, and is defined recursively, as following:

1. $\mathcal{S} \vDash \top$
2. $\mathcal{S} \vDash p$ iff $p \in \mathcal{S}$
3. $\mathcal{S} \vDash \neg\varphi$ iff not $\mathcal{S} \vDash \varphi$
4. $\mathcal{S} \vDash \varphi \wedge \psi$ iff $\mathcal{S} \vDash \varphi$ and $\mathcal{S} \vDash \psi$

A formula φ is called *valid* iff it satisfies all assignments and this is denoted as $\vDash \varphi$.

1.1.1 Epistemic Logic

We begin by defining the general syntax of a formula in epistemic logic.

Definition 1.1.3 (Epistemic Logic Syntax). *Given a set of agents $\mathcal{I}$ and a vocabulary $\mathcal{V}$, the language of epistemic logic $\mathcal{L}(\mathcal{V})$ extends the Boolean language $\mathcal{L}_B(\mathcal{V})$, and is expressed by the following BNF:*

$$\varphi ::= \top \mid p \mid \neg\varphi \mid \varphi \wedge \varphi \mid K_i\varphi \mid C_\Delta\varphi$$

Where $p \in \mathcal{V}$, $i \in \mathcal{I}$ and $\Delta \subseteq I$.

The formula $K_i\varphi$ is read as "agent i knows that φ is true", and the formula $C_\Delta\varphi$ says that φ is common knowledge among agents in the group Δ.

Before discussing the semantics of a formula, the model upon which the truth of a formula is evaluated, is discussed in the following two definitions.

Definition 1.1.4 (Kripke Frame). *A **Kripke Frame** for a set of agents $\mathcal{I} = \{1, \ldots, n\}$ is a tuple $\mathcal{M} = \langle \mathcal{W}, \mathcal{R} \rangle$, where $\mathcal{W}$ is a finite set of possible worlds and $\mathcal{R}$ is a family of binary relations over $\mathcal{W}$ indexed by agents i.e $\mathcal{R}_i \subseteq \mathcal{W} \times \mathcal{W}$ for each $i \in \mathcal{I}$.*

Definition 1.1.5 (Kripke Model). *A **Kripke Model** for a set of agents $\mathcal{I}$ and a vocabulary $\mathcal{V}$ is a tuple $\mathcal{M} = \langle \mathcal{W}, \Pi, \mathcal{R} \rangle$, where $\langle \mathcal{W}, \mathcal{R} \rangle$ is a Kripke Frame for $\mathcal{I}$ and $\Pi : \mathcal{W} \to \mathscr{P}(\mathcal{V})$ is a valuation function and $\mathscr{P}(\mathcal{V})$ is the powerset of $\mathcal{V}$.*

For any group of agents $\Delta \subseteq \mathcal{I}$, the transitive closure of the union of their relations is denoted by $\mathcal{R}_\Delta$ which is defined as $(\cup_{i \in \Delta} \mathcal{R}_i)^*$. A **pointed Kripke Model** is a pair $(\mathcal{M}, w)$ where w is a world of $\mathcal{M}$. The interpretation of the relation $\mathcal{R}_i$ is that it relates worlds which the agent i considers possible in its current knowledge state.

Knowledge is defined in terms of this possibility: agent i knows φ at a world w iff at all the worlds that i considers possible with respect to w, φ is true. Assuming $\mathcal{R}_i$ to be an equivalence relation: i knows something iff it is true at all those worlds that i cannot distinguish from the actual world w.

In Definition 1.1.6, a special class of Kripke Models called **S5 Kripke Models**, is defined.

Definition 1.1.6 (S5 Kripke Model). *For a group of agents $\mathcal{I}$ and a Kripke Model $\mathcal{M} = \langle \mathcal{W}, \Pi, \mathcal{R} \rangle$, if each $\mathcal{R}_i$ is an equivalence relation, then $\mathcal{M}$ is called to be a S5 Kripke Model.*

Now the semantics of epistemic logic is introduced in Definition 1.1.7.

Definition 1.1.7 (Epistemic Logic Semantics). *Semantics for $\mathcal{L}(\mathcal{V})$ on pointed S5 Kripke models are as follows:*

1. $(\mathcal{M}, w) \vDash \top$
2. $(\mathcal{M}, w) \vDash p$ *iff* $p \in \Pi(w)$
3. $(\mathcal{M}, w) \vDash \neg\varphi$ *iff not* $(\mathcal{M}, w) \vDash \varphi$
4. $(\mathcal{M}, w) \vDash \varphi \wedge \psi$ *iff* $(\mathcal{M}, w) \vDash \varphi$ *and* $(\mathcal{M}, w) \vDash \psi$

5. $(\mathcal{M}, w) \vDash K_i\varphi$ *iff for all* $w_j \in W$, *if* $\mathcal{R}_i w w_j$, *then* $(\mathcal{M}, w_j) \vDash \varphi$

6. $(\mathcal{M}, w) \vDash C_\Delta\varphi$ *iff for all* $w_j \in W$, *if* $\mathcal{R}_\Delta w w_j$, *then* $(\mathcal{M}, w_j) \vDash \varphi$

Note that, since $\top$ is trivially satisfied in any model, we refrain from including it while defining the semantics throughout this work.

1.1.2 Example of Kripke Model

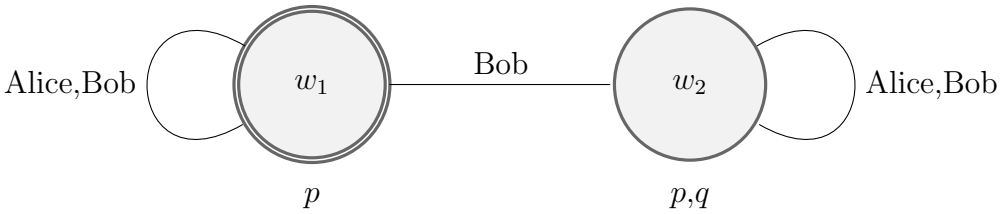


Figure 1.1: A Kripke Model

Figure 1.1 shows a Kripke model $\mathcal{M}$. The model consists of two worlds w_1 and w_2, and describes the Epistemic state of two agents called Alice and Bob. The actual world w_1 is marked with a double border in Figure 1.1. The semantics, in this scenario, is illustrated with the following true statement:

- $(\mathcal{M}, w_1) \vDash p \wedge \neg q \wedge C_{Alice,Bob}p \wedge K_{Alice}\neg q \wedge \neg K_{Bob}q$

1.2 Dynamic Epistemic logic

In this section a very brief overview of Dynamic Epistemic Logic which permits modeling change in epistemic state of an agent is presented. Update of epistemic states was introduced in [Pla07], which is known as Public Announcement Logic (PAL) in the literature. This extends epistemic logic with a modality to describe incoming information in the form of a truthful public announcement, made by a trusted authority, which is received and accepted by every agent.

But knowledge can be changed by many other forms of communications such as semi-private, private or secret announcements. Epistemic states of agents can also be altered by occurrence of events.

As an example, if in a room with two persons, a coin is flipped at random, and the result is made known to one of the persons then the knowledge of that person regarding the state of affairs changes but that of the other person remains unchanged.

In [BMS98], **Action Models** were introduced to model such complex communications and events. This framework came to be known as Dynamic Epistemic Logic (DEL) [DvdHK07].

The syntax and semantics of Dynamic Epistemic Logic formulas are as following:

Definition 1.2.1 (Action Model). *Let $\mathcal{V}$ be a vocabulary. An action model is a tuple $\mathcal{A} = (A, \mathcal{R}_A, pre)$ where A is a set of atomic events, $\mathcal{R}_A$ is a family of relations $R_i \subseteq A \times A$ for each i, $pre : A \to \mathcal{L}(\mathcal{V})$ is a function which assigns to each event a formula called the precondition.*

Definition 1.2.2 (Product Update). *Given a Kripke model $\mathcal{M}$ and an action model $\mathcal{A}$ using the same vocabulary, their product is $\mathcal{M} \times \mathcal{A} \coloneqq \langle \mathcal{W}^{new}, \Pi^{new}, \mathcal{R}_i^{new} \rangle$, where $\mathcal{W}^{new} \coloneqq \{(w, a) \in \mathcal{W} \times \mathcal{A}^A \mid (\mathcal{M}, w) \vDash pre(a)\}$, $\mathcal{R}_i^{new} \coloneqq \{((w, a), (v, b)) \mid \mathcal{R}_i^{\mathcal{M}} wv$ and $\mathcal{R}_i^{\mathcal{A}} ab\}$ and $\Pi^{new}((w, a)) \coloneqq \Pi(w)$.*

Definition 1.2.3 (Action). *An action is a pair $(\mathcal{A}, a)$ where $a \in \mathcal{A}^A$. To update a pointed Kripke model with an action the following is defined: $(\mathcal{M}, w) \times (\mathcal{A}, a) \coloneqq (\mathcal{M} \times \mathcal{A}, (w, a))$.*

Definition 1.2.4 (DEL Syntax). *Given a vocabulary $\mathcal{V}$, the language of Dynamic Epistemic logic $\mathcal{L}_D(\mathcal{V})$ with dynamic operators for action models extends $\mathcal{L}(\mathcal{V})$ and is given by the following BNF:*

$$\varphi \coloneqq \top \mid p \mid \neg\varphi \mid \varphi \wedge \varphi \mid K_i\varphi \mid C_\Delta\varphi \mid [\mathcal{A}, a]\varphi$$

Here $p \in \mathcal{V}$, $i \in \mathcal{I}$, $\Delta \subseteq \mathcal{I}$ and $(\mathcal{A}, a)$ is an action. Since the only new operator is the one denoting application of an action, only the semantic for that operator is defined in Definition 1.2.5.

Definition 1.2.5 (DEL Semantics). *The semantic of $[\mathcal{A}, a]$ operator is as following: $(\mathcal{M}, w) \vDash [\mathcal{A}, a]\varphi$ iff $(\mathcal{M}, w) \vDash pre(a)$ implies $(\mathcal{M} \times \mathcal{A}, (w, a)) \vDash \varphi$*

We conclude this section with an example of update with action models:

1.2.1 An example of update with action model

. Let us consider the following scenario. A coin lies on a table in a dark room. There are two persons a and b, in that room. a is facing the table, whereas b is looking in the opposite direction relative to the table. Initially none of the persons are aware of the fact that the coin lies heads up. Let the proposition that this coin is heads up be denoted with p. Now a light-bulb is lit in the room is lit by pressing

a switch from outside. Since only, a is facing the table, a becomes aware of the fact p immediately but b's ignorance regarding the situation still persists.

Figure 1.2 depicts a Kripke model of the initial situation, an action model representing the event of lighting the room and the resultant Kripke model. The actions are represented with rectangles. The precondition is prefixed with the symbol '?'.

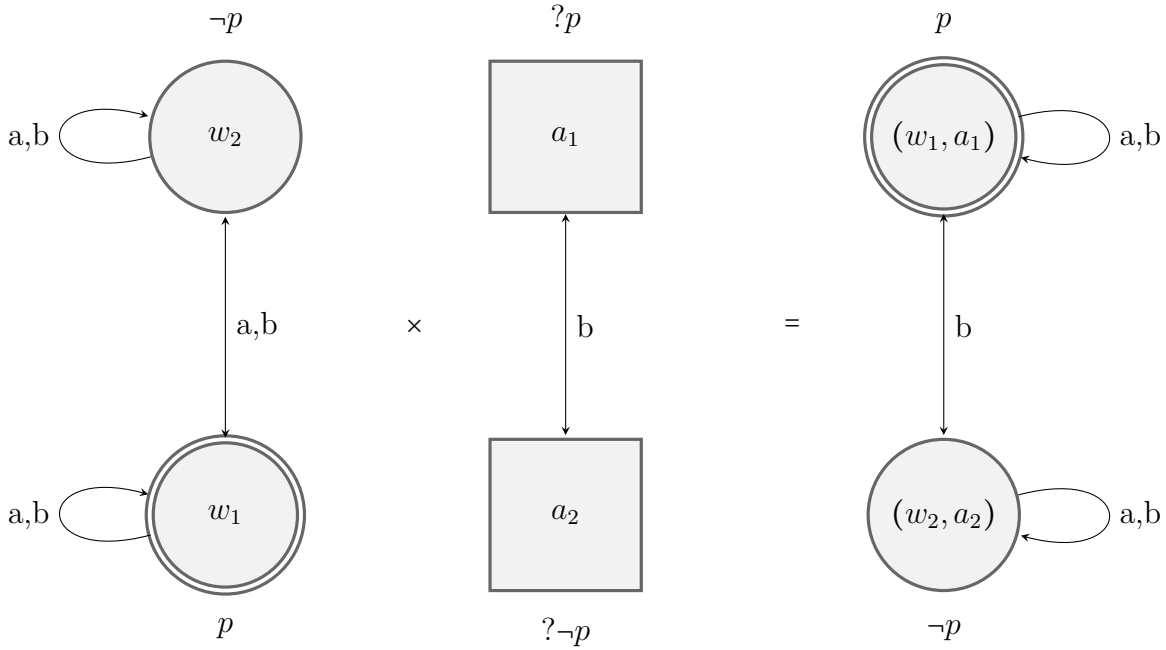


Figure 1.2: Coin Heads or Tails: Update with Action Model

Chapter 2

Logic of Protocols

In this chapter the notion of protocol, mentioned in Chapter 1 will be made more precise and the different logics of protocols as introduced in [vDGVW14] will be discussed. The notion of protocol in this context, following the convention in [vDGVW14], emphasizes upon understanding the underlying meaning of the actions induced by a protocol.

As an illustrative example consider an epistemic scenario wherein the agents are not only uncertain about the factual state of the world but also about the protocol that can be executed given some factual state, depicted as Figure 2.1: Here s and t

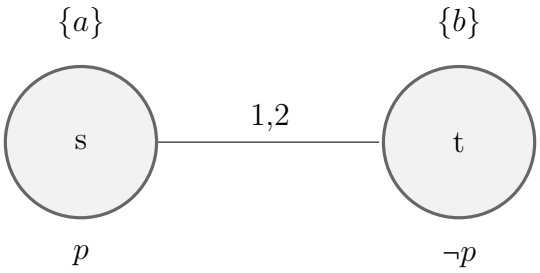


Figure 2.1: An epistemic scenario with state-dependent protocols

are labels of possible worlds, 1 and 2 are two agents, p is a proposition, and a and b are expected actions (the notion of action here is related to but not exactly the same as those on page 8). The uncertainty of the agents regarding the protocol is denoted by a state-dependent protocol assigning singleton action sets $\{a\}$ to s and $\{b\}$ to t. The reflexive arrows are omitted in this figure.

In the above example model, intuitively it is possible to reclaim some form of common knowledge of the protocol, by describing the protocol as follows: if p then perform a and if $\neg p$ then perform b. In the following sections a precise notion of such intuitive description, namely **observation expressions** and **protocol expressions** are presented.

The formalism discussed in the following sections does not incorporate other aspects of protocols such as how these protocols are designed in the first place and how the agents come to agree to use them.

2.1 Expectation and Observation

In order to reason about update of knowledge of an agent via matching its observations with its expectations, the concept of an *Epistemic Expectation Model*, which is an augmented form of the Kripke Model (Section 1.1.1), is utilized. An agent observes events that occur around it and reasons based upon these observations.

In general observation of actions such as 'going to the right', are the focus of this framework rather that of facts, such as 'the chair is red'. An observation can be thought of as a finite sequence of actions and an agent may expect to observe any one among (infinitely many) different observations at any given state in the world. Therefore in order to succinctly represent the potential observations of an agent, the notion of an observation expression (a form of regular expressions) is used.

Definition 2.1.1 (Observation Expression). *Given a finite set of action symbols Σ, the language $\mathcal{L}_{obs}$ of observation expressions is defined by the following BNF:*

$$\pi ::= \delta \mid \epsilon \mid a \mid \pi \cdot \pi \mid \pi + \pi \mid \pi^*$$

where δ stands for the empty set $\varnothing$ of observations, ϵ represents the empty string, and $a \in \Sigma$. The semantics for observation expressions are given by sets of observations (which are strings over Σ), in a similar way to that of regular expressions.

The notion of observation is formalized in Definition 2.1.2 in terms of observation expression.

Definition 2.1.2 (Observation). *Let Σ be a set of action symbols and π be an observation expression over Σ. The corresponding set of observations, $\mathcal{L}(\pi)$ is the set of finite strings over Σ, which are defined recursively,*

1. $\mathcal{L}(\delta) = \varnothing$
2. $\mathcal{L}(\epsilon) = \{\epsilon\}$
3. $\mathcal{L}(a) = \{a\}$
4. $\mathcal{L}(\pi_1 \cdot \pi_2) = \{wv \mid w \in \mathcal{L}(\pi_1) \text{ and } v \in \mathcal{L}(\pi_2)\}$
5. $\mathcal{L}(\pi_1 + \pi_2) = \mathcal{L}(\pi_1) \cup \mathcal{L}(\pi_2)$
6. $\mathcal{L}(\pi^*) = \{\epsilon\} \cup \bigcup_{n>0} \mathcal{L}(\pi^n)$

The definition of an Epistemic Expectation Model, which can be thought of as an epistemic model (Kripke Model) augmented by a set of expected observations for each world, expressed as an observation expression, follows:

Definition 2.1.3 (Epistemic Expectation Model). *A tuple $\langle \mathcal{S}, \mathcal{V}, \sim, Exp \rangle$ where $\langle \mathcal{S}, \mathcal{V}, \sim \rangle$ is an epistemic model (a S5 kripke model) and $Exp : S \to \mathcal{L}_{obs}$ is an expected observation function that assigns to each state $s \in \mathcal{S}$ an observation expression π such that $\mathcal{L}(\pi) \neq \varnothing$ is called an Epistemic Expectation Model $\mathcal{M}_{exp}$.*

Similar to the pointed Kripke Models in page 5, an epistemic expectation state is a pointed epistemic expectation model $\langle \mathcal{M}_{exp}, s \rangle$ where $s \in \mathcal{S}$. The subscript is often dropped if the meaning is clear from the context.

Remark 2.1.4. *An epistemic model $\mathcal{M}$ can be thought of as an epistemic expectation model $\mathcal{M}_{exp}$ where for all $s \in \mathcal{S}$, $Exp(s) = \Sigma^*$.*

2.1.1 Example of an Epistemic Expectation Model

In this example, we consider the following scenario from [vDGVW14]: *In the Netherlands, people often greet each other by kissing three times on the cheek (left-right-left) while in the rest of Europe, people usually kiss each other only twice.* We can reason whether a person is 'Dutch-related' by observing her behavior. Let p_D be the proposition meaning 'Mary is Dutch-related', a and b are two actions denoting kissing the left cheek and the right cheek, respectively. The following is the corresponding Epistemic Expectation Model: The equivalence relation in

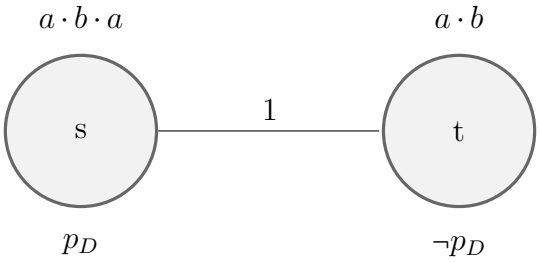


Figure 2.2: Dutch or not Dutch problem

Figure 2.2 depicts that agent 1 does not know whether p_D. In the next section a logic, called Public Observation Logic which is suitable for reasoning based on actual observations is presented.

2.2 Public Observation Logic

Before introducing the syntax and semantics of Public Observation Language formulas (POL), update of epistemic expectation models according to some observation $w \in \Sigma^*$ is defined. The underlying assumption is that agents are inherently

equipped with the capability of distinguishing between possible and impossible scenarios by observing sequence of actions.

Definition 2.2.1 (Update by Observation). *Let w be an observation over Σ^* and $\mathcal{M} = \langle \mathcal{S}, \mathcal{V}, \sim, Exp \rangle$ be an epistemic expectation model. The updated model is $\mathcal{M}_{|w} = \langle \mathcal{S}', \mathcal{V}', \sim', Exp' \rangle$ where the components are given as, $\mathcal{S}' \coloneqq \{s \mid Exp(s)\backslash w \neq \delta\}$, $\sim'_i \coloneqq \sim_{i|(\mathcal{S}' \times \mathcal{I} \times \mathcal{S}')}$, $\mathcal{V}' \coloneqq \mathcal{V}_{|\mathcal{S}'}$ and $Exp'(s) \coloneqq Exp(s)\backslash w$. The notation $\pi\backslash w$ denotes the observation expression corresponding to the set $\{v \mid wv \in \mathcal{L}(\pi)\}$ and $\mathcal{I}$ is the set of agents.*

Note that intuitively if $w = \epsilon$ then the model should not change at all. In Remark 2.2.2, this notion is made precise.

Remark 2.2.2. *$\pi\backslash w$ is defined with the following auxiliary output function $\mathcal{O} \coloneqq$*

1. $\pi = \mathcal{O}(\pi) + \sum_{a \in \Sigma}(a \cdot \pi\backslash a)$
2. $\mathcal{O}(\epsilon) \coloneqq \epsilon$
3. $\mathcal{O}(\delta) \coloneqq \mathcal{O}(a) = \delta$
4. $\mathcal{O}(\pi_1 + \pi_2) \coloneqq \mathcal{O}(\pi_1) + \mathcal{O}(\pi_2)$
5. $\mathcal{O}(\pi_1 \cdot \pi_2) \coloneqq \mathcal{O}(\pi_1) \cdot \mathcal{O}(\pi_2)$
6. $\mathcal{O}(\pi^*) \coloneqq \epsilon$
7. $\delta\backslash a = \epsilon\backslash a = b\backslash a \coloneqq \delta \ \ (a \neq b)$
8. $a\backslash a \coloneqq \epsilon$
9. $(\pi_1 + \pi_2)\backslash a = \pi_1\backslash a + \pi_2\backslash a$
10. $(\pi_1 \cdot \pi_2)\backslash a = (\pi_1\backslash a) \cdot \pi_2 + \mathcal{O}(\pi_1) \cdot (\pi_2\backslash a)$
11. $\pi^*\backslash a = \pi\backslash a \cdot \pi^*$
12. $\pi\backslash a_0 \ldots a_n = \pi\backslash a_0 \ldots \backslash a_n$
13. $\pi\backslash \epsilon \coloneqq \pi$

Definition 2.2.3 (POL Syntax). *The formulas in POL are expressed by the following BNF* $\coloneqq$

$$\varphi \coloneqq \top \mid p \mid \neg\varphi \mid \varphi \wedge \varphi \mid K_i\varphi \mid [\pi]\varphi$$

Here $p \in \mathcal{P}$ is the set of propositional symbols, $i \in \mathcal{I}$ is the set of agents and $\pi \in \mathcal{L}_{obs}$. The modal operator $[\pi]$ is called the *observation update operator*. As before in Section 1.1, other propositional connectives are defined in the usual manner.

Definition 2.2.4 (POL Semantics). *The semantics of POL formulas are as following. Given an epistemic expectation model $\mathcal{M} = \langle \mathcal{S}, \mathcal{V}, \sim, Exp \rangle$, a state $s \in \mathcal{S}$, and a POL-formula φ,*

1. $\mathcal{M}, s \vDash p$ *iff* $p \in \mathcal{V}(s)$
2. $\mathcal{M}, s \vDash \neg\varphi$ *iff not* $\mathcal{M}, s \vDash \varphi$
3. $\mathcal{M}, s \vDash \varphi_1 \wedge \varphi_2$ *iff* $\mathcal{M}, s \vDash \varphi_1$ *and* $\mathcal{M}, s \vDash \varphi_2$
4. $\mathcal{M}, s \vDash K_i\varphi$ *iff for all* $t : s \sim_i t, \mathcal{M}, t \vDash \varphi$
5. $\mathcal{M}, s \vDash [\pi]\varphi$ *iff for all* $w \in \mathcal{L}(\pi)$, *if* $w \in init(Exp(s))$ *then* $\mathcal{M}_{|w}, s \vDash \varphi$ *where* $w \in init(\pi)$ *iff existsv* $\in \Sigma*$ *such that* $wv \in \mathcal{L}(\pi)$ *i.e.,* $\mathcal{L}(\pi \backslash w) \neq \varnothing$

In Figure 2.2, for example, $\mathcal{M}, s \vDash [a \cdot b](\neg K_1 p_D \wedge [a] K_1 p_D)$. In Section 2.3, a formal framework for reasoning about how expectation is acquired and changed by agents, called Epistemic Protocol Logic ([vDGVW14]), is presented.

2.3 Epistemic Protocol Logic

As a starting point, the notion of protocols and protocol models as sources for expected observations is defined as following:

Definition 2.3.1 (Protocol Expression). *The language $\mathcal{L}_{prot}$ is defined by the following BNF:*

$$\eta ::= \delta \mid \epsilon \mid a \mid ?\varphi \mid \ \eta + \eta \mid \eta \cdot \eta \mid \eta^*$$

where $a \in \Sigma$, $\varphi \in \mathcal{L}_B(\mathcal{P})$, $\mathcal{P}$ is a set of propositions.

Remark 2.3.2. *The test condition (?) in protocol expressions is used to describe the conditions under which certain observations may occur.*

For example, referring to Figure 2.2, the underlying protocol can be expressed as $?p_D \cdot a \cdot b \cdot a + ?\neg p_D \cdot a \cdot b$.
The semantics for protocol expressions are as following:

Definition 2.3.3 (Semantics of Protocol Expressions). *The set $\mathcal{L}_g(\eta)$ of guarded observations of the form $\rho_0 a_0 \ldots \rho_k a_k$, where $p_j \in \mathcal{P}$, associated to a protocol η, encoding the conditions for later observations to be witnessed, is defined inductively as following* :=

1. $\mathcal{L}_g(\delta) = \varnothing$
2. $\mathcal{L}_g(\epsilon) = \{\, \rho \mid \rho \subseteq \mathcal{P} \,\}$
3. $\mathcal{L}_g(a) = \{\, \rho a \rho \mid \rho \subseteq \mathcal{P} \,\}$
4. $\mathcal{L}_g(?\varphi) = \{\rho a \rho \mid \rho \vDash \varphi,\ \rho \subseteq \mathcal{P}\}$
5. $\mathcal{L}_g(\eta_1 \cdot \eta_2) = \{\, w \diamond v \mid w \in \mathcal{L}_g(\eta_1),\ v \in \mathcal{L}_g(\eta_2) \,\}$
6. $\mathcal{L}_g(\eta_1 + \eta_2) = \mathcal{L}_g(\eta_1) \cup \mathcal{L}_g(\eta_2)$
7. $\mathcal{L}_g(\eta^*) = \{\, \rho \mid \rho \subseteq \mathcal{P} \cup \bigcup_{n>0} \mathcal{L}_g(\eta^n) \,\}$

where $w \diamond v = w'\rho v'$ *when* $w = w'\rho$ *and* $v = \rho v'$*, and not defined otherwise;* $\rho \vDash \varphi$ *iff* φ *is true under the valuation induced by* ρ*.*

In Definition 2.3.4, a function to convert a protocol expression to an observation expression is presented.

Definition 2.3.4. *The set of observations which are expected under the same condition, denoted by the set of propositions* ρ *according to a protocol* η *are expressed by the following conversion function* $f_\rho : \mathcal{L}_{prot} \to \mathcal{L}_{obs}$

1. $f_\rho(\delta) = \delta$
2. $f_\rho(\epsilon) = \epsilon$
3. $f_\rho(a) = a$
4. $f_\rho(?\varphi) =$ *if* $\rho \vDash \varphi$ *then* ϵ *else* δ
5. $f_\rho(\eta_1 + \eta_2) = f_\rho(\eta_1) + f_\rho(\eta_2)$
6. $f_\rho(\eta_1 \cdot \eta_2) = f_\rho(\eta_1) \cdot f_\rho(\eta_2)$
7. $f_\rho(\eta^*) = (f_\rho(\eta))^*$

In Definition 2.3.5, an Epistemic Protocol Model is presented,

Definition 2.3.5 (Epistemic Protocol Model). *An Epistemic Model* $\mathcal{A}$ *is a triplet* $\langle \mathcal{T}, \sim, Prot \rangle$*, where* $\mathcal{T}$ *is a domain of abstract objects,* $\sim$ *stands for a set of equivalence relations* $\{\, \sim_i \mid i \in \mathcal{I} \,\}$*, and* $Prot : \mathcal{T} \to \mathcal{L}_{prot}$ *assigns to each domain object a protocol expression.*

A pointed Epistemic Protocol Model is known as an Epistemic Protocol (analogous to the definition of an action on page 8).

Definition 2.3.6 (Protocol Update). *Given an Epistemic Expectation Model,* $\mathcal{M}_{exp} = \langle \mathcal{S}, \mathcal{V}, \sim, Exp \rangle$ *and an Epistemic Protocol Model* $\mathcal{A} = \langle \mathcal{T}, \sim, Prot \rangle$*, the product* $\mathcal{M}_{exp} \times \mathcal{A} = \langle \mathcal{S}', \mathcal{V}', \sim', Exp' \rangle$ *is defined as following:*

1. $\mathcal{S}' = \{ (s,t) \in \mathcal{S} \times \mathcal{T} \mid \mathcal{L}(f_{\mathcal{V}_{\mathcal{M}}(s)}(Prot(t))) \neq \varnothing \}$
2. $(s,t) \sim_i' (s',t')$ *iff* $= s \sim_i s'$ *and* $t \sim_i t'$
3. $\mathcal{V}'(s,t) = \mathcal{V}(s)$
4. $Exp'((s,t)) = f_{\mathcal{V}_{\mathcal{M}}(s)}(Prot(t))$

Now the framework of Epistemic Protocol Logic(EPL), which is a DEL-style logic based upon an extension of POL (Section 2.2) and is suitable to describe the 'change' of protocols, together with the effect of observations of agents, based upon the current protocol, is presented:

Definition 2.3.7 (EPL Syntax). *The formulas of EPL are given by the following BNF:*

$$\varphi ::= \top \mid p \mid \neg\varphi \mid \varphi \wedge \varphi \mid K_i\varphi \mid [\pi]\varphi \mid [!\mathcal{A}_e]\varphi$$

where $p \in \mathcal{P}$*,* $i \in \mathcal{I}$*,* $\pi \in \mathcal{L}_{obs}$ *and* $\mathcal{A}_e$ *is an Epistemic Protocol with designated state* e*.*

Remark 2.3.8. *EPL formulas are evaluated with respect to an Epistemic Expectation Model. It is assumed that the protocol models are finite. Given any EPL formula* φ *that does not contain the modal operator* $[!\mathcal{A}_e]$ *and a pointed Epistemic Expectation Model* $\mathcal{M}_{exp}$ *with a designated state* s*, the truth conditions are the same as that of POL formulas in Section 2.2. Therefore the following definition only expresses the truth condition for this new modal operator* $[!\mathcal{A}_e]$*.*

Definition 2.3.9 (EPL Semantics). *Given an EPL formula* $[!\mathcal{A}_e]\varphi$ *and a pointed Epistemic Expectation Model* $\mathcal{M}_{exp}$ *with a designated state* s*, we have*

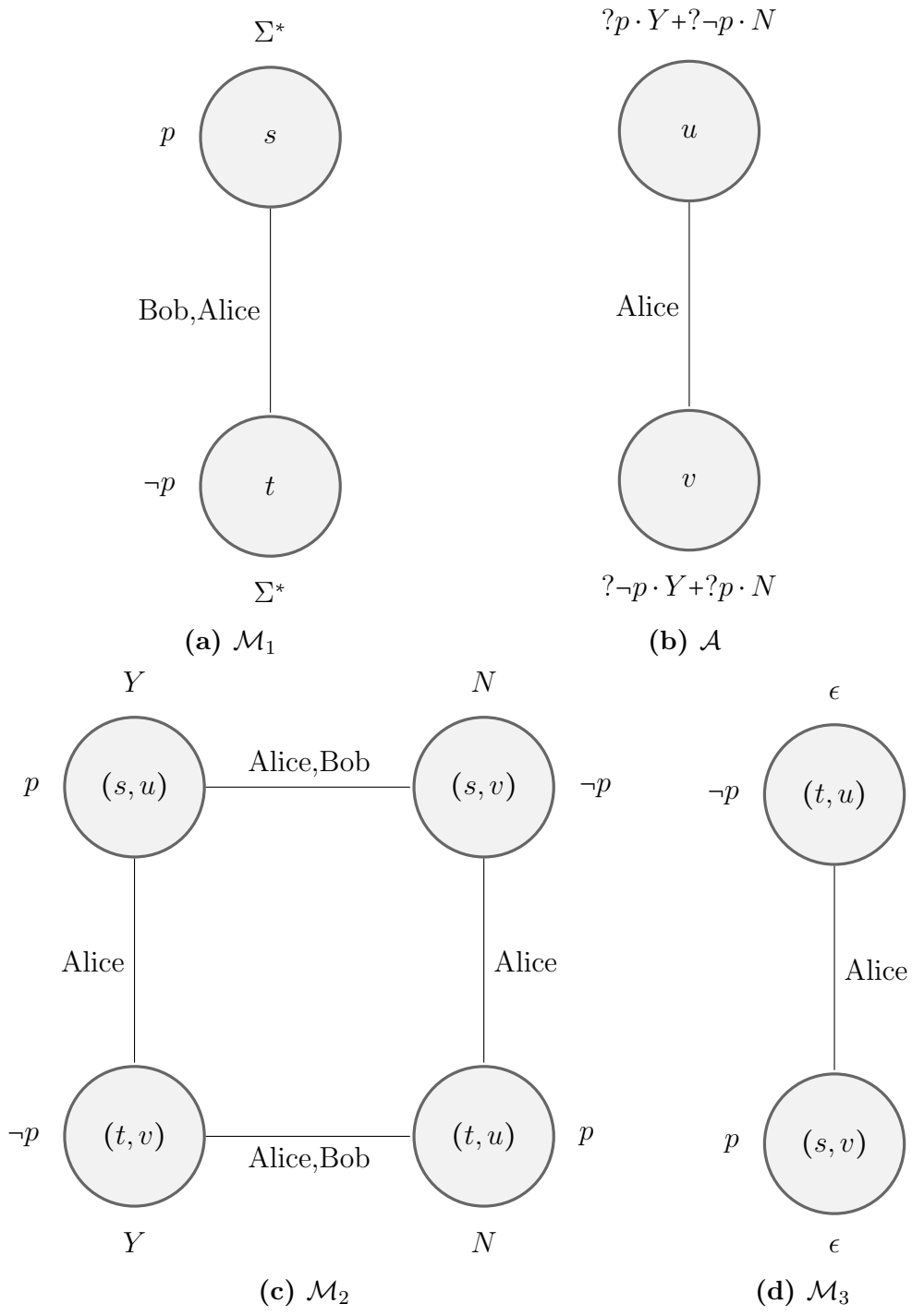
$$\mathcal{M}_{exp}, s \vDash [!\mathcal{A}_e]\varphi \text{ iff } \mathcal{L}(f_{\mathcal{V}(s)}(Prot(e))) \neq \varnothing \text{ implies } \mathcal{M} \times \mathcal{A}, (s,e) \vDash \varphi$$

2.3.1 An Example of Protocol Update

Consider the following scenario: *Three friends Carl, Bob and Alice are sitting in a pub. Carl says to Bob, who is his childhood friend, "On Valentine's day I went to this pub with Mike and Sara. It was a crazy night!". This immediately catches the attention of Alice, who is secretly in love with Mike. She asks: 'What happened?' Carl winks to Bob, who know each other like the back of their hands,*

and replies, "Nothing". Bob immediately realizes that indeed nothing had occurred, whereas Alice, who doesn't know what that wink stands for, becomes unsure about the events of that night.

A depiction of the initial expectation model, the protocol model and the updated expectation model is depicted in Figure 2.3. Here p stands for the fact that 'Some-



(a) $\mathcal{M}_1$ **(b)** $\mathcal{A}$ **(c)** $\mathcal{M}_2$ **(d)** $\mathcal{M}_3$

Figure 2.3: The Valentine's Day problem

thing has happened involving Mike and Sara on Valentine's night', while Y encodes

Carl answering affirmatively to Alice's question, and N encodes Carl answering negatively.

Initially(Figure 2.3a), in model $\mathcal{M}_1$ both Alice and Bob were expecting any possible observation(denoted by Σ^*) and considered the states s (where p holds) and t (where $\neg p$ holds) equally possible. The ignorance of Alice regarding the meaning underlying Carl's wink is depicted in *Figure 2.3b* as $\mathcal{A}$.

Here at states u and v with valuation p and $\neg p$ respectively, the coresponding protocol expressions are $?p \cdot Y + ?\neg p \cdot N$ and $?\neg p \cdot Y + ?p \cdot N$ in that order, meaning if p holds at that state then Alice considers Y otherwise she considers N.

Note that it in this protocol (indicated by the reflexive arrows for Bob, which are omitted) Bob is not ignorant about the meaning of Carl's action and hence the equivalence class induce by the accessibility relation pertaining to him, consists of only singleton sets.

The updated epistemic expectation model $\mathcal{M}_2$, after the installation of this protocol is shown in Figure 2.3c. In Figure 2.3d, $\mathcal{M}_3$ depicts the model after the observation N.

It can be easily verified that in this example, $\mathcal{M}_1, t \vDash [!\mathcal{A}_u][N](K_{Bob}\neg p \wedge \neg K_{Alice}\neg p)$, but $\mathcal{M}_1, t \vDash [!\mathcal{A}_u][N]K_{Alice}(K_{Bob}p \vee K_{Bob}\neg p)$.

Chapter 3

Model Checking Logic of Protocols

In Chapter 1, the model checking problem was described informally. If both the system description (in the form of a model) and the required specification can be formalized in a logical language, then the model checking problem is equivalent to asking whether the system fulfills its specification. A formal description of the problem can be expressed as the following question: Given a model, $\mathcal{M}$ and a formula φ, is φ true in $\mathcal{M}$? In the context of this work, $\mathcal{M}$ is an Epistemic Expectation Model and $\varphi \in \mathcal{L}_{POL}$ (or $\varphi \in \mathcal{L}_{EPL}$).

This is a trivial question from the perspective of mathematical logic, since if the semantics of a logic is properly defined, then the model checking problem can be answered using the definition of satisfaction for a formula recursively, where the number of steps involved is usually not more than the size of the formula.

But from the perspective of automating this procedure, the problem becomes hard, since it is not apparent, in concrete terms, what data structures are suitable for succinct representation of the model, and what framework is to be used in order to efficiently determine satisfiability.

As mentioned before in Chapter 1, naive implementation of standard logical semantics usually results in the State Explosion Problem, wherein the number of worlds increases exponentially in terms of agents and propositions thereby increasing inefficiency. A solution to this predicament was first presented by Randal Byrant who introduced Binary Decision Diagrams [Bry86].

This led to emergence of Symbolic Model Checking wherein a description of a model that is capable of evaluating all the formulas of interest is provided as input in the form of a symbolic (ideally compact) representation along with a formula to be evaluated. However the problem of formulating compact symbolic representation is not always tractable.

In this chapter, we will focus upon the algorithmic aspects of model checking the logics of protocols, and formulate an explicit model checking framework. Chapter 4 focuses upon an implementation of the ideas in this chapter. We leave symbolic model checking of POL and EPL as future works.

3.1 Related Work

A first step towards symbolic model checking epistemic modality was [SSL07] which presents a symbolic model checking framework for temporal logics of knowledge by a Boolean translation of the knowledge operators but does not cover dynamic operators such as announcements. On the other hand, frameworks that are capable of Although there are no specific prior work upon model checking dynamic logics of epistemic protocols that we know of, general frameworks for explicit model checking DEL do exist, notably **DEMO-S5** [vE14]. We mention the case of DEL because the data structures that are used to design explicit model checking tools for DEL are quite relevant for our purpose, since Protocol Models are analogous to Event Models in DEL with respect to their corresponding structures. For example, the idea to represent equivalence relations with partitions, in order to compactly store the accessibility relations of an agent, was implemented in **DEMO-S5**. However it is to be noted that in the worst case, when for each agent each state of the world is only related to itself, this idea is not better than using list of tuples.

3.2 Model Checking Framework

Before discussing the methodology employed for formulation of a model checking framework, it is explored whether a suitable transformation of the Epistemic Expectation Models yields an algorithm for model checking POL (and thereby EPL). The intuition behind such an initiative is that the observation expressions are analogous to program constructs in Propositional Dynamic Logic (PDL) [TB19] in terms of their corresponding syntax. Thus it is imperative to consider whether a direct transformation from Epistemic Expectation Models to Epistemic Temporal Models, if achievable, is fruitful towards implementing a model checking framework for POL by translating a POL formula accordingly. In Section 3.2.1 it is discussed whether such a transformation, if exists, can be directly leveraged or not in formulation of our framework.

3.2.1 Translation to Epistemic Propositional Dynamic Logic

An epistemic temporal model is a Kripke model with both epistemic and temporal binary relations between possible worlds. It can be shown that Epistemic Expectation Models (EEMs) are compact representations of a particular form of Epistemic Temporal Models (ETMs) as presented in Definition 3.2.1:

Definition 3.2.1 (Generated Epistemic Temporal Model). *Let $\mathcal{M}$ be an Epistemic Expectation Model $\langle \mathcal{S}, \sim_i, \mathcal{V}, Exp \rangle$. The $\mathcal{M}$-generated Epistemic Temporal Model $\mathcal{ET}(\mathcal{M})$ is defined as, $\langle \mathcal{H}, \xrightarrow{a}, , \sim'_i, \mathcal{V}' \rangle$. The components of $\mathcal{ET}(\mathcal{M})$ are defined as following* :=

1. $\mathcal{H} = \{ (s, w) \mid s \in \mathcal{S}, w = \epsilon \text{ or } w \in \mathcal{L}(Exp(s)) \}$
2. $(s, w) \xrightarrow{a} (t, v)$ *iff* $s = t$ *and* $v = wa, a \in \Sigma$
3. $(s, w) \sim'_i (t, v)$ *iff* $s \sim_i t$ *and* $w = v$
4. $p \in \mathcal{V}'(s, w)$ *iff* $p \in \mathcal{V}(s)$

The semantics of POL formulas (only the modal operators are shown) over these Epistemic temporal Models are as following:

1. $\langle \mathcal{N}, h \rangle \vDash K_i \varphi$ iff for all h' such that $h \sim_i h'$, $\langle \mathcal{N}, h' \rangle \vDash \varphi$
2. $\langle \mathcal{N}, h \rangle \vDash [\pi]\varphi$ iff for each $w \in \mathcal{L}(\pi)$, for all h' such that $h \xrightarrow{w} h'$, $\langle \mathcal{N}, h' \rangle \vDash \varphi$

The logic thus defined is called Epistemic Propositional Dynamic Logic (EPDL) in [vDGVW14].

The equivalence of between Epistemic Expectation Models and Epistemic Temporal Models is made precise in Theorem 3.2.2:

Theorem 3.2.2 (Equivalnce of EEM with ETM).

$$\langle \mathcal{M}, s \rangle \vDash \varphi \text{ iff } \langle ET(\mathcal{M}), (s, \epsilon) \rangle \vDash_{EPDL} \varphi$$

where $\langle \mathcal{M}, s \rangle$ is a pointed Epistemic Expectation Model and φ is a POL formula.

Proof. Refer to page 12 in [vDGVW14]. ■

Thus the problem of model checking a POL formula against an Epistemic Expectation Model reduces to the problem of model checking that formula (viewed as an EPDL formula) against the generated Epistemic Temporal Model. But from Definition 3.2.1 it is evident that the number of states in a generated Epistemic Temporal Model is potentially infinite. Therefore the ideas from Model Checking temporal logic can not be utilized to provide a model checking procedure for POL.

In Section 3.2.3, it is shown that POL is decidable. For the sake of convenience, in Section 3.2.2, syntax and semantics of POL is very briefly mentioned.

3.2.2 Syntax and Semantics of POL

In Chapter 2, the syntax and semantics of POL formulas were discussed in detail, which is, again, briefly presented here for convenience.

POL formulas are interpreted over the class of Epistemic Expectation Models. An Epistemic Expectation Model is a Kripke Model augmented with an Expectation function that associates an Observation Expression with each of the states.

The syntax is given by the following BNF,

$$\varphi ::= \top \mid p \mid \neg\varphi \mid \varphi \wedge \varphi \mid K_i\varphi \mid [\pi]\varphi$$

The modal operator $[\pi]$, is called the observation update operator in this thesis.

The semantics is as following: Given an epistemic expectation model $\mathcal{M} = \langle \mathcal{S}, \mathcal{V}, \sim, Exp \rangle$, a state $s \in \mathcal{S}$, and a POL-formula φ,

1. $\mathcal{M}, s \vDash p$ iff $p \in \mathcal{V}(s)$
2. $\mathcal{M}, s \vDash \neg\varphi$ iff not $\mathcal{M}, s \vDash \varphi$
3. $\mathcal{M}, s \vDash \varphi_1 \wedge \varphi_2$ iff $\mathcal{M}, s \vDash \varphi_1$ and $\mathcal{M}, s \vDash \varphi_2$
4. $\mathcal{M}, s \vDash K_i\varphi$ iff for all $t : s \sim_i t, \mathcal{M}, t \vDash \varphi$
5. $\mathcal{M}, s \vDash [\pi]\varphi$ iff for all $w \in \mathcal{L}(\pi)$, if $w \in init(Exp(s))$ then $\mathcal{M}_{|w}, s \vDash \varphi$ where $w \in init(\pi)$ iff $existsv \in \Sigma*$ such that $wv \in \mathcal{L}(\pi)$ i.e., $\mathcal{L}(\pi \backslash w) \neq \varnothing$

3.2.3 Decidability of POL

Since only the introduction of the formula $[\pi]\varphi$ extends standard epistemic logic to POL, and the model checking problem of standard epistemic logic is decidable, we need to consider the semantics of this formula only for deciding the model checking problem of POL. But first some notations are presented in Definition 3.2.3 and Definition 3.2.4.

Definition 3.2.3 (Compliant Observation). *Let Σ be a set of action symbols. $w \in \Sigma^*$ is said to be **compliant** with an observation expression π iff $w \in init(\pi)$ i.e $\pi \backslash w \neq \delta$. This is denoted as $w \bowtie \pi$.*

Definition 3.2.4 (Pre-sequences of Observation Expressions). *Let π be an observation expression over a set of action symbols Σ. The set of all the pre-sequences of π is $init(\pi) = \{ w \mid w \bowtie \pi \}$.*

Lemma 3.2.5 shows that for an observation expression π, the set $init(\pi)$ can be finitely partitioned. In [Wan11], a quite similar lemma is proved for a scenario involving a PDL-style logic interpreted over pointed S5 kripke models.

Lemma 3.2.5. *Let π be an observation expression over a finite set of action symbols Σ. There exists a minimal natural number k such that the set $init(\pi)$ can be partitioned into k sets, each expressed by observation expressions $\pi_0, \ldots, \pi_k$, where for any two $w, v \in \mathcal{L}(\pi_i)$ for some $i \in \{1, \ldots, k\}$, $\pi\backslash w = \pi\backslash v$.*

Proof. Observation Expressions are a domain specific format of Regular Expressions. For any regular expression a corresponding minimal Deterministic Finite Automaton (DFA) can be constructed [HMU07]. Let the minimal DFA corresponding to the observation expression π be $\mathcal{A} = \{\mathcal{Q}, \Sigma, q_0, \mathcal{T}, \mathcal{F}\}$, where $\mathcal{Q} = \{q_0, \ldots q_k\}$, $\mathcal{F} \subseteq \mathcal{Q}$ and q_0 be the start state. Let $q \in \mathcal{Q}$ be a state in $\mathcal{A}$. q is said to be **$\mathcal{F}$-admissible** if there exists at least one state $f \in \mathcal{F}$ such that f is reachable from q. For each $i \leq k$ such that q_i is $\mathcal{F}$-admissible, let π_i be the observation expression corresponding to the DFA, $\mathcal{A}_i = \{\mathcal{Q}, \Sigma, q_0, \mathcal{T}, \{q_i\}\}$. Since each $\mathcal{A}_i$ is deterministic, the family of sets $\{\mathcal{L}(\pi_i) \mid i \in \{1, \ldots k\}\}$ constitutes such a partitioning of $init(\pi)$. ■

Following the convention in [Wan11], we call these partitions ***pre-derivatives***. Now the equivalence relation induced by pre-derivatives upon action strings (strings formed by action symbols) is defined in Definition 3.2.6.

Definition 3.2.6 (Pre-derivative Equivalence). *Let τ be an observation expressions over a set of action symbols Σ. $u, v \in \Sigma^*$ are said to be pre-derivative equivalent with respect to τ iff $u, v \in \mathcal{L}(\tau_i)$ for a pre-derivative τ_i of τ. It is denoted as $u \sim^{\tau} v$.*

Now Theorem 3.2.7 shows that the language of all POL formulae, $\mathcal{L}_{POL}$ is decidable.

Theorem 3.2.7. *Given an Epistemic Expectation scene $\langle \mathcal{M}, s \rangle$ and a formula $[\pi]\varphi \in \mathcal{L}_{POL}$, the problem of determining whether $\langle \mathcal{M}, s \rangle \vDash [\pi]\varphi$ is decidable.*

Proof. In Definition 2.2.4 the semantics of $[\pi]\varphi$ over Epistemic Expectation Models is provided, according to which for all $w \in \mathcal{L}(\pi)$ such that $w \in init(Exp(s))$, determining whether φ is satisfied at $\langle \mathcal{M}_{|w}, s \rangle$ is necessary. According to Lemma 3.2.5, there are finitely many pre-derivatives of $Exp(s)$. Let $\mathcal{D}^{Exp} = \{D_1^{Exp}, \ldots, D_k^{Exp}\}$ be the set of pre-derivatives of $Exp(s)$. Thence from Definition 2.2.1 it follows that for any $i \in \{1, \ldots, k\}$ for any two $u, v \in D_i^{Exp}$, $\mathcal{M}_{|u} \equiv \mathcal{M}_{|v}$. Furthermore $\mathcal{L}(\pi) \cap init(Exp(s)) = \bigcup_i (\mathcal{L}(\pi) \cap D_i^{Exp})$. Hence from Definition 3.2.6 it follows that

$\sim^{Exp(s)}$ induces a finite partition upon $\mathcal{L}(\pi) \cap init(Exp(s))$. Therefore it is sufficient to pick an element w from each $\mathcal{L}(\pi) \cap D_i^{Exp}$ and check whether φ is satisfied at $\langle \mathcal{M}_{|w}, s \rangle$. Thus this problem is decidable. ■

Remark 3.2.8. *For an Epistemic Expectation scene $\langle \mathcal{M}, s \rangle$ and a formula $[\pi]\varphi \in \mathcal{L}_{POL}$, Theorem 3.2.7 also provides us with an algorithm to check whether $\langle \mathcal{M}, s \rangle \models [\pi]\varphi$ as described in Algorithm 1. Note that, while there is a minimal natural number k corresponding to an observation expression π such there are not fewer than k pre-derivatives of π, we can still bi-partition the language of a pre-derivative of π to produce another two pre-derivatives. This corresponds to the case when we do not use the minimal DFA in Lemma 3.2.5. This is particularly helpful in construction of an algorithm.*

Remark 3.2.9. *Note that decidability of EPL follows from decidability of POL and the fact that the protocol update operator transforms an Epistemic Expectation Model into another Epistemic Expectation Model.*

In the following sections we present the relevant algorithms for model checking POL (and EPL).

3.3 Algorithms for Model Checking POL and EPL over EEMs

At first the procedure for construction of a DFA from an observation expression is presented. Afterwards auxiliary algorithms for some functionalities upon DFAs will be presented, and then algorithms for evaluation of POL (and EPL) formulas will be discussed.

The following two remarks discuss some notions that are utilized in our method for DFA construction.

In Remark 3.3.1, a modified form of observation expressions [ALSU06], which are required for construction of DFA, is presented.

Remark 3.3.1. *Let π be an observation expression over a vocabulary $\mathcal{V}$. An augmented observation expression is of the form, $\pi \cdot \#$ where $\#$ is an end marking symbol. It is ensured that $\# \notin \mathcal{V}$.*

In Remark 3.3.2, syntax trees for observation expressions are discussed.

Remark 3.3.2. *A syntax tree for an observation expression is a such a tree that each leaf node corresponds to either the empty string symbol or a character literal*

value. Non-leaf nodes correspond to the operators in the expression. Each leaf node is marked with a single position value. The position value starts from 1 and increases from left to right. For example if two leaf nodes are siblings and the leaf at the left has position value n then its sibling at the right side will have position value n + 1. The set of position values for a non-leaf node n is the collection of all position values of the leaf nodes in the subtree rooted at n. The set of position values for a symbol 'a' is the collection of position values of those leaves that contain the symbol 'a'.

In Figure 3.1, a syntax tree is shown. In the next subsection i.e. Section 3.3.1, some functions upon syntax trees that are required in DFA construction, are discussed.

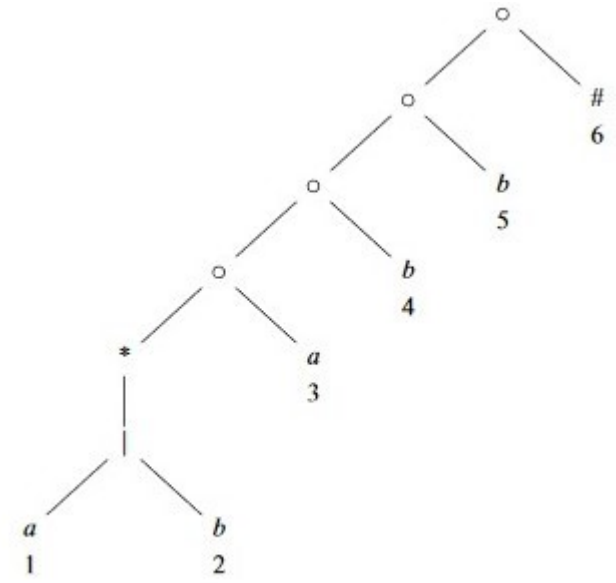


Figure 3.1: A syntax tree for the expression $(a+b)^* \cdot a \cdot b \cdot b \cdot \#$

3.3.1 Functions from Syntax Tree

For a syntax tree $\mathcal{T}$ representing an augmented observation expression $\tau = \pi \cdot \#$, the following four functions are computed to facilitate construction of a DFA.

- nullable(n) is true for a syntax-tree node n if and only if the subexpression represented by the subtree rooted at n has ϵ in its language.
- firstpos(n) is the set of positions in the subtree rooted at n that correspond to the first symbol of at least one string in the language of the subexpression rooted at n.

- lastpos(n) is the set of positions in the subtree rooted at n that correspond to the last symbol of at least one string in the language of the subexpression rooted at n.

- followpos(p), for a position p,is the set of positions q in the entire syntax tree such that there is some string $x = a_1 a_2 \dots a_n$ in $\mathcal{L}(\tau)$ such that for some i there is a way to explain the membership of x in $\mathcal{L}(\tau)$ by matching a_i to position p and a_{i+1} to position q.

In Figure 3.2, nullable values for a given syntax tree, in Figure 3.3, firstpos and lastpos sets for that syntax tree, and in Figure 3.4, followpos sets for the same tree are shown respectively.

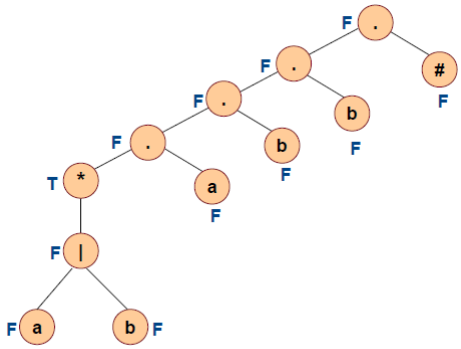


Figure 3.2: nullable values for each node in a syntax tree for $(a+b)^* \cdot a \cdot b \cdot b \cdot \#$

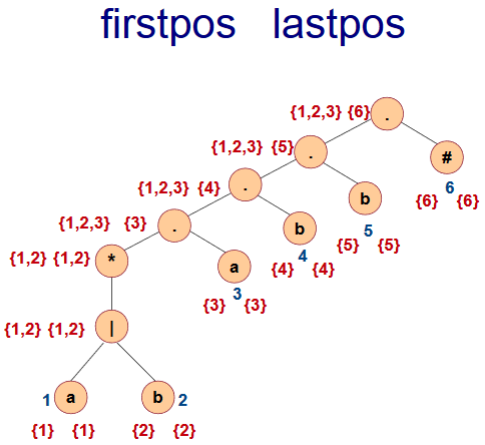


Figure 3.3: firstpos and lastpos sets for each node in a syntax tree for $(a+b)^* \cdot a \cdot b \cdot b \cdot \#$

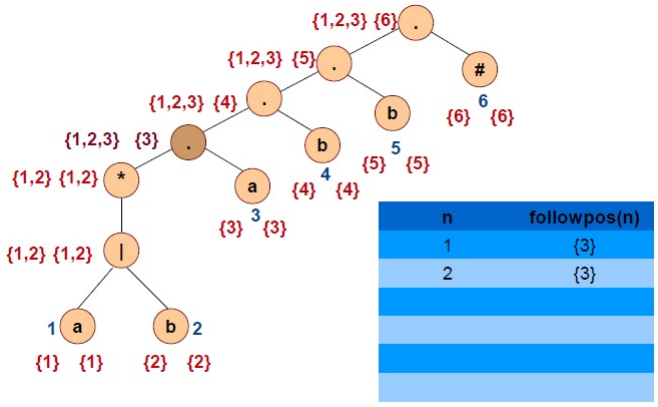


Figure 3.4: followpos sets for two leaf nodes in a syntax tree for $(a+b)^* \cdot a \cdot b \cdot b \cdot \#$

For a given syntax tree, the functions nullable, firstpos and lastpos are computed essentially by a straightforward recursion on the height of that tree. The basis and inductive rules for nullable and firstpos are summarized in Figure 3.5. The rules for lastpos are essentially the same as that for computation of firstpos, but the roles of children c_1 and c_2 have to be swapped for a node representing $\cdot$ (the concatenation operator). We leave the rules for computing followpos in this work. This can be found on **p. 177** in [ALSU06].

NODE n	$nullable(n)$	$firstpos(n)$
A leaf labeled ϵ	**true**	$\emptyset$
A leaf with position i	**false**	$\{i\}$
An or-node $n = c_1 \mid c_2$	$nullable(c_1)$ **or** $nullable(c_2)$	$firstpos(c_1) \cup firstpos(c_2)$
A cat-node $n = c_1 c_2$	$nullable(c_1)$ **and** $nullable(c_2)$	**if** ($nullable(c_1)$) $firstpos(c_1) \cup firstpos(c_2)$ **else** $firstpos(c_1)$
A star-node $n = c_1{}^*$	**true**	$firstpos(c_1)$

Figure 3.5: Rules for computing nullable, firstpos functions as presented in p. 177, "Compilers: Principles, Techniques, & Tools" by Aho, Lam, Sethi, Ullman

3.3.2 Method of DFA Construction

The method presented below is from Chapter 3 in [ALSU06]. In order to compute a DFA from an observation expression the following steps in order are to be performed.

1. Construct a syntax tree for the augmented expression $\tau = \pi \cdot \#$.
2. Compute nullable, firstpos, lastpos and followpos functions using the rules as described in Figure 3.5.
3. Construct a finite labeled transition graph, D' (start state and set of final states is not designated) using Algorithm 1
4. Mark $firstpos(root) \in D'_S$ as the start state in for a DFA, D, where $root$ is the root of the syntax tree and D'_S is the set of states in D'.
5. Mark those sets, which contain the position '#' as final states thereby constructing D from D'.

The procedure to obtain a labeled finite transition graph from a syntax tree is presented in Algorithm 1. $\varnothing$ represents an empty set in that algorithm.

In the following subsection, two algorithms are presented. Algorithm 2 returns a list of DFAs, each representing a pre-derivative of the observation expression corresponding to the input DFA. Algorithm 3 returns a string that is in the language of the input DFA.

3.3.3 Algorithms for some auxiliary functions upon DFAs

The procedure to obtain the DFAs corresponding to pre-derivatives is shown in Algorithm 2. The procedure to obtain a string that is acepeted by a given DFA is shown in Algorithm 3. The function 'findPath' is essentially a depth first search based procedure that returns a path from the start state to a final state in a DFA. It is implicitly assumed that every edge has weight 1 in this procedure. Here a depth first search is performed to extract a path from the start state to a final state.

3.3.4 Evaluation of Formula

The algorithms for evaluation of EPL formulas are shown in Algorithm 4 to Algorithm 9. The procedure **Convert** in Algorithm 9 is defined in Definition 2.3.4. It uses a straightforward recursion upon a protocol expression to obtain an observation expression.

Algorithm 1: Procedure to Obtain a Labeled Finite Transition Graph from a Syntax Tree

```
1  function: Obtain-Transitions (𝒯)
   Input  : A Syntax Tree 𝒯
   Output: A labeled finite transition diagram D′
2  S ← ∅
3  D′ ← INIT-LABELED-FTD()
4  s ← firstpos(root(𝒯))
5  mark(s) ← False
6  S.insert(s)
7  while there exists t ∈ S such that mark(t) == False do
8  |  mark(t) ← True
9  |  foreach a ∈ 𝒯.symbols do
10 |  |  p ← positionValuesOf(a)
11 |  |  foreach pv ∈ p do
12 |  |  |  sf ← ⋃ followpos(pv)
13 |  |  end
14 |  |  D′.transition[s, a] = sf
15 |  |  if sf ∉ S then
16 |  |  |  S.insert(sf)
17 |  |  |  mark(sf) ← False
18 |  end
19 end
20 return D′
```

Algorithm 2: Procedure to Obtain List of DFAs corresponding to Pre-derivatives

```
1  function: DFA-PRE-DERIVATIVE (𝒟)
   Input  : A DFA 𝒟
   Output: A list of DFAs DFALIST
2  DFALIST ← []
3  foreach p ∈ (𝒟.States \ 𝒟.Finals) do
4  |  𝒟′ ← 𝒟
5  |  𝒟′.Finals ← {}
6  |  insert(𝒟′.Finals, p)
7  |  DFALIST.insert(𝒟′)
8  end
9  DFALIST.insert(𝒟)
10 return DFALIST
```

Algorithm 3: Procedure to Obtain a String in the Language of a Given DFA

function: **FindString** ($\mathcal{D}$)
Input : A DFA $\mathcal{D}$
Output: A string w
$w \leftarrow \epsilon$
$finalPath \leftarrow findPath(\mathcal{D})$
for $i = 1$ *to* $finalPath.length$ **do**
  $(u, v) \leftarrow finalPath[i]$
  $w.concatenate(readLabel(u, v))$
end
return w

Algorithm 4: Procedure to Model Check an EPL Formula — Proposition

function: **Evaluate** ($\langle \mathcal{M}, s \rangle, p$)
Input : An Epistemic Expectation Scene $\langle \mathcal{M}, s \rangle$ and an EPL-formula p, where p is a proposition
Output: $True$ or $False$
if $p \in \mathcal{V}(s)$ **then**
  return $True$
else
  return $False$
end

Algorithm 5: Procedure to Model Check an EPL Formula — Negation

function: **Evaluate** ($\langle \mathcal{M}, s \rangle, \varphi$)
Input : An Epistemic Expectation Scene $\langle \mathcal{M} = \{\mathcal{W}, s\rangle$ and an EPL-formula $\varphi = \neg\psi$
Output: $True$ or $False$
if $Evaluate(\langle \mathcal{M}, s \rangle, \psi) == True$ **then**
  return $False$
else
  return $True$
end

Algorithm 6: Procedure to Model Check an EPL Formula — Conjunction

```
1  function: Evaluate (⟨M, s⟩, χ)
   Input  : An Epistemic Expectation Scene ⟨M = {W, s⟩ and an
            EPL-formula χ = φ ∧ ψ
   Output: True or False
2  if (Evaluate(⟨M, s⟩, φ) == True) && ((Evaluate(⟨M, s⟩, ψ) == True)
    then
3  |  return True
4  else
5  |  return False
6  end
```

Algorithm 7: Procedure to Model Check an EPL Formula — Knowledge Operator

```
1  function: Evaluate (⟨M, s⟩, ψ)
   Input  : An Epistemic Expectation Scene ⟨M = {W, s⟩ and an
            EPL-formula ψ = Kiφ where i is an agent
   Output: True or False
2  status ← True
3  for t ∈ Related(∼, i, s) do
4  |  status = status∧ Evaluate(⟨M, t⟩, φ)
5  |  if status == False then
6  |  |  return False
7  |  else
8  |  |  continue
9  |  end
10 end
11 return True
```

Algorithm 8: Procedure to Model Check an EPL formula — Observation Update Operator

function: **Evaluate** $(\langle \mathcal{M}, s \rangle, \psi)$
Input : An Epistemic Expectation Scene $\langle \mathcal{M}, s \rangle$ and an EPL-formula $\psi = [\pi]\varphi$
Output: $True$ or $False$
$status \leftarrow True$
$\mathcal{D}_{Exp(s)} \leftarrow$ **BuildDFA**$(Exp(s))$
$\mathcal{D}_{\pi} \leftarrow$ **BuildDFA**(π)
$\{\mathcal{D}_{\tau_1}, \ldots, \mathcal{D}_{\tau_k}\} \leftarrow$ **DFA-PRE-DERIVATIVE**$(init(Exp(s))$
for $i \leftarrow 1$ ***to*** k **do**
  $\mathcal{D}_{\pi \times \tau_i} \leftarrow \mathcal{D}_{\pi} \times \mathcal{D}_{\tau_i}$
  if $\mathcal{L}(\mathcal{D}_{\pi \times \tau_i}) \neq \varnothing$ **then**
    $w =$ **FindString**$(\mathcal{D}_{\pi \times \tau_i})$
    $mark =$ **Evaluate**$(\mathcal{M}_{|w}, \varphi)$
    $status = status \wedge mark$
  else
    continue
  end
end
return *status*

Algorithm 9: Procedure to Model Check an EPL Formula — Protocol Update Operator

function: **Evaluate** $(\langle \mathcal{M}, s \rangle, \psi)$
Input : An Epistemic Expectation Scene $\langle \mathcal{M}, s \rangle$ and an EPL-formula $\psi = [!\mathcal{A}_e]\varphi$ where $\mathcal{A}_e$ is a protocol
Output: $True$ or $False$
if ***Convert*** $(\mathcal{V}(s), Prot(e)) == \delta$ **then**
  return $True$
else
  $updatedM \leftarrow \mathcal{M} \times \mathcal{A}$
  $state \leftarrow$ **FindState**$(updatedM, s, e)$
  return ***Evaluate***$(\langle updatedM, state \rangle, \varphi)$
end

Chapter 4

Model Checking Tool for Logic of Protocols

In Chapter 3, algorithmic aspects of explicit model checking Public Observation Logic and Epistemic Protocol Logic were discussed. In this chapter, an implementation in the form of a Haskell library which is based upon the algorithms in Chapter 3 is discussed. The module is still under development especially regarding I/O handling, and therefore is not released yet in Hackage, which is the repository of Haskell modules. The code is available on demand as of now.

This chapter is broadly divided into two sections. In Section 4.1, data structures and implementations of the algorithms in Chapter 3 are briefly discussed, where as in Section 4.3, an example of model checking a scenario is presented.

Besides the main task of model checking POL/EPL formulas, other auxiliary tasks such as an implementation of a customized observation/protocol expression evaluator, functions for DFA manipulation etc are also implemented.

This implementation is written in Haskell and can be used from a command line interface. All parts of the implementation are not discussed in detail and the complete source code is not included in this work, only relevant code snippets are presented.

In Section 4.1, a very brief overview of existing software for epistemic model checking is presented. The rationale of choosing Haskell is also illustrated in this section highlighting its advantages for dealing with data types for logical formulas.

4.1 Introduction

Since our implementation deals with three modal operators, namely the Knowledge Operator K, the Observation Update Operator $[\pi]$, and the Protocol Update Operator $\mathcal{A}_e$, of which, only K has been studied before for implementation purpose, we begin with a very brief discussion of a few other model checking tools that have incorporated the epistemic modality K. As mentioned in Chapter 3, the vast majority of existing symbolic model checking tools deal with temporal modalities

only. One of the first model checkers for temporal logic incorporating K is MCTK [SSL07], which employed the technique of translating the modality K into a boolean formula thereby facilitating symbolic model checking. Since we are not using symbolic means for this implementation, we are not going to mention further any other tool employing symbolic translation methodology. For Dynamic Epistemic Logic, in contrast, the standard explicit implementations are the two explicit model checkers by Jan van Eijck: DEMO and its successor DEMO-S5 [vE14], both written in Haskell.

The major advantage of explicit model checking tools for epistemic logics is their usability in the sense that they provide an interface to deal with the Kripke structures directly. The models can also be manipulated at a single possible world and if required, can be visualized using any graph generating tool.

DEMO further utilized the power of Haskell's type system to gain extra flexibility: possible worlds in Kripke models can be of any suitable type a, thereby carrying information in their names, eliminating the need for a separate valuation function.

4.1.1 Rationale for Choice of Haskell

The choice of Haskell is motivated due to the following considerations [Gat18],

- Since this is a functional language, its facilities for list syntax, pattern matching and point-free function composition allow to write code that resembles the original notation thereby greatly reducing the burden to devise representations for mathematical objects.

- Since Haskell is statically typed, it is guaranteed to produce errors arising out of type mismanagement in compile time. This property also helps ensuring correctness of a program. For example it is impossible to represent a formula that is not well-formed in our program.

- Since Haskell provides succinct representations for our purpose, using an object-oriented language will result in greater number of lines of code.

- Most of the existing tools for model checking Epistemic Logic are written in Haskell, thereby providing some opportunity for code re-use.

- Most importantly, Haskell is lazy, i.e. it only evaluates expressions in our program when they are needed. This provides a means to work with infinite structures such as the list of natural numbers [0..] or an infinite supply of

atomic propositional variables — as long as it is made sure that, once this is actually run, only a finite part will be used. This also helps with finite objects in the following sense. If parts of a model or structure is needed only for some computation, the rest need not be immediately computed.

It is worth mentioning that in terms of formulating a program, it often happens that differences which we did not care about when defining something, suddenly become important in order to provide a correct implementation. As an example, we usually identify a propositional variable $p \in \mathcal{V}$ with the same variable used as a formula $p \in \mathcal{L}(\mathcal{V})$, for some vocabulary $\mathcal{V}$. But to implement this in a typed language such as Haskell, it is necessary to make a distinction between these two. Similarly, distinctions need to made regarding subsets of $A \times B$ and functions of the form $A \to \mathscr{P}(B)$ in Haskell : lists of pairs [(a,b)] and unary functions to lists $a->[b]$ are different types.

Furthermore, often mathematical definitions need to be reformulated due to practical considerations. For an example, rather than using $\varphi \wedge \psi$, it is more convenient to use $\varphi \vee \psi$ whenever feasible since, evaluating the former requires consideration of both φ and ψ. While this particular example seems irrelevant and a trivial point to make, such effects propagate for complex boolean formulas and their elimination results in better performance.

In the following section, the details of the implementation are discussed. As mentioned before, only relevant code snippets are provided.

4.2 Explicit Model Checking

The module developed here is called EPLMC (Epistemic Logic Model Checker). It is capable of model checking any EPL formula, provided enough memory and time. We discuss the issues with scalability of this approach in Section 4.2.6.

This module is still highly experimental and Input/Output is still quite cumbersome. Specifically, a data format for reading in a complete description of a Epistemic Expectation Model (EEM) and Protocol Models is yet to be implemented. However input/output functionalities have been designed for Observation and Protocol Expressions.

Furthermore, no functionality of visualizing EEMs, is provided yet. Existing graph visualization tools for Haskell (such as GraphViz module for Haskell), still lack the capability to accept certain symbols that are used in regular expressions to denote operators, as labels for graph nodes. Therefore no ready-made solution is available as of now for this purpose. We leave all this for future work.

This section is further divided into the following subsections:

4.2.1 Basic Representations

In Figure 4.1 and Figure 4.2, some basic data-types are shown.

```
type Rel a b = [(a,b)]
type Erel a = [[a]]

data Agent = Agt Int
    deriving (Eq,Ord)

instance Show Agent where
    show (Agt n) = "Agent " ++ show n
```

Figure 4.1: ADT for an agent and type synonym for a partition of a set

```
data Proposition = P Int
    deriving (Eq,Ord)

instance Show Proposition where
    show (P n) = "P" ++ show n

data BForm = PrpB Proposition
           | BNeg BForm
           | BAnd BForm BForm
           | BOr BForm BForm
  deriving (Eq)
```

Figure 4.2: ADTs for propositions and Boolean formulas

4.2.2 Representing Observation and Protocol Expressions

We define Observation and Protocol Expressions in terms of a construct in Haskell, called Algebraic Data Types (ADTs). This construct provides a means to implement a new data type T from existing types by prefixing the types with corresponding tags, called value constructors. The following provides an example.

Example 4.2.1. *'data Pair = P Int Double' creates a pair of numbers, an Int and a Double together. The tag P is used (in other constructors and pattern matching)*

to combine the contained values (of type Int and Double in that order) into a single structure that can be assigned to a variable of type Pair.

In Figure 4.3, the ADT for Observation Expressions, and in Figure 4.4, the ADT for Protocol Expressions, are shown.

```
data ObExpr = ObDel
            | ObEps
            | ObLit Char
            | ObAlt ObExpr ObExpr
            | ObCat ObExpr ObExpr
            | ObStar ObExpr
  deriving (Eq,Ord)

showObExpr :: ObExpr -> String
showObExpr ObDel         = "$"
showObExpr ObEps         = "@"
showObExpr (ObLit c)     = [c]
showObExpr (ObAlt e1 e2) = "(" ++ showObExpr e1 ++  "+" ++ showObExpr e2 ++ ")"
showObExpr (ObCat e1 e2) = "(" ++ showObExpr e1 ++ "." ++ showObExpr e2 ++ ")"
showObExpr (ObStar e)    = showObExpr e ++ "*"

instance Show ObExpr where
    show = showObExpr
```

Figure 4.3: ADT for Observation Expressions

```
data PrtExpr = PtDel
             | PtEps
             | PtLit Char
             | PtAlt PrtExpr PrtExpr
             | PtTest BForm
             | PtCat PrtExpr PrtExpr
             | PtStar PrtExpr
  deriving (Eq)

showPtExpr :: PrtExpr -> String
showPtExpr PtDel         = "$"
showPtExpr PtEps         = "@"
showPtExpr (PtLit c)     = [c]
showPtExpr (PtTest f)    = "?" ++ show f
showPtExpr (PtAlt e1 e2) = "(" ++ showPtExpr e1 ++  "+" ++ showPtExpr e2 ++ ")"
showPtExpr (PtCat e1 e2) = "(" ++ showPtExpr e1 ++ "." ++ showPtExpr e2 ++ ")"
showPtExpr (PtStar e)    = showPtExpr e ++ "*"

instance Show PrtExpr where
    show = showPtExpr
```

Figure 4.4: ADT for Protocol Expressions

We only show the important functions for Observation Expressions and Protocol Expressions in Figure 4.5 and Figure 4.6.

4.2.3 Tokenization and Parsing

We use two tools, alex and happy, which are a lexical analyzer and a parser respectively, written in Haskell and available as standalone soft-wares, for tokenizing and parsing observation expressions.

```
obNullable :: ObExpr -> Bool
obNullable r = if obRedux(obAux r) == ObDel then False else True

obQ :: ObExpr -> ObExpr -> ObExpr
obQ e (ObCat l1 l2) = obRedux( obQ (obRedux (obQ e l1) ) l2)
obQ (ObDel) _      = ObDel
obQ (ObEps) l    = if (ObEps == obRedux(l)) then ObEps else ObDel
obQ (ObLit e) l = if (ObLit e /= l) then ObDel else ObEps
obQ (ObStar e) l     =  obRedux(ObCat (obRedux (obQ e l)) (obRedux (ObStar e)))
obQ (ObCat e1 e2) l = obRedux(ObAlt (obRedux(ObCat (obRedux(obQ e1 l)) e2))
                                    (obRedux(ObCat (obAux e1) (obRedux(obQ e2 l)))))
obQ (ObAlt e1 e2) l =  obRedux(ObAlt (obRedux(obQ e1 l)) (obRedux(obQ e2 l)))
```

Figure 4.5: Function ObQ for the quotient (\) operation upon Observation Expressions

```
ptConvert :: PrtExpr -> [Proposition] -> ObExpr
ptConvert PtDel ps          = ObDel
ptConvert PtEps ps          = ObEps
ptConvert (PtLit c) ps      = ObLit c
ptConvert (PtTest f) ps     = if (isValid ps f) then ObEps else ObDel
ptConvert (PtAlt e1 e2) ps  = obRedux ( ObAlt (ptConvert e1 ps) (ptConvert e2 ps))
ptConvert (PtCat e1 e2) ps  = obRedux ( ObCat (ptConvert e1 ps) (ptConvert e2 ps))
ptConvert (PtStar e) ps     = obRedux (ObStar (ptConvert e ps))
```

Figure 4.6: Function for converting Protocol Expressions to Observation Expressions

In Figure 4.7 and Figure 4.8, the alex specifications for Observation Expressions are shown. We denote δ with @ and ϵ with $. In Figure 4.9 and Figure 4.10, the

```
$alpha = [a-zA-Z]
$eol   = [\n]

tokens :-

  $eol                          ;
  $white+                       ;
  [@]                           { \s -> ObTokenEps }
  [\$]                          { \s -> ObTokenDel }
  [\+]                          { \s -> ObTokenAlt }
  [\.]                          { \s -> ObTokenCat }
  [\*]                          { \s -> ObTokenStar }
  \(                            { \s -> ObTokenLParen }
  \)                            { \s -> ObTokenRParen }
  $alpha                        { \s -> ObTokenSym (head s) }
```

Figure 4.7: Tokens for Observation Expressions

happy specifications for Protocol Expressions are shown. We do not show the alex specification for Observation Expressions and happy Specification for Protocol Expressions in this work. The purpose of these specifications is to build corresponding parsers for Observation and Protocol Expressions for I/O.

In the next section, the representation and functions of EEMs and Protocol Models are shown.

```
data ObToken = ObTokenEps
             | ObTokenDel
             | ObTokenSym Char
             | ObTokenAlt
             | ObTokenCat
             | ObTokenStar
             | ObTokenLParen
             | ObTokenRParen
             deriving (Eq,Show)

scanObTokens = alexScanTokens
```

Figure 4.8: ADT for Observation Expressions Tokens

```
%token
    ATOM  { PtTokenSym $$ }
    FORM  { PtTokenTest $$ }
    '+'   { PtTokenAlt }
    '.'   { PtTokenCat }
    '*'   { PtTokenStar }
    '@'   { PtTokenDel }
    '$'   { PtTokenEps }
    '('   { PtTokenLParen }
    ')'   { PtTokenRParen }

%left '+'
%left '.'
%right '*'
```

Figure 4.9: Happy specification for Protocol Expressions

4.2.4 Epistemic Expectation Models and Protocol Models

In Figure 4.11, the ADT of an EEM, and in Figure 4.12, that of a Protocol Model is shown. Since the accessibility relations are equivalences, they are represented

```
PrtExpr : '(' PrtExpr '+' PrtExpr ')'    { PtAlt $2 $4 }
        | PrtExpr '+' PrtExpr            { PtAlt $1 $3 }
        | '(' PrtExpr '.' PrtExpr ')'    { PtCat $2 $4 }
        | PrtExpr '.' PrtExpr            { PtCat $1 $3 }
        | '(' PrtExpr ')' '*'            { PtStar $2 }
        | PrtExpr '*'                    { PtStar $1}
        | FORM                           { PtTest $1 }
        | '(' FORM ')'                   { PtTest $2 }
        | ATOM                           { PtLit $1 }
        | '$'                            { PtDel }
        | '@'                            { PtEps }
```

Figure 4.10: Happy specification for Protocol Expressions

by partitions, indexed by each agent. Here they are given by list of tuples of the form [(agent,[[states]])], where each element in this list is a tuple of the form (agent[[states]]). For example, if for an agent 'alice', and the states s,t,u and v, it is the case that sR_{alice}u,uR_{alice}v but not sR_{alice}t, then the tuple for 'alice' is (alice,[[s,u,v],[t]]).

```
data EpExpM state = Mo
    [state]
    [Agent]
    [(state,[Proposition])]
    [(state,ObExpr)]
    [(Agent,Erel state)]
    deriving (Eq)

instance Show state => Show (EpExpM state) where
  show (Mo worlds ags val exp accs) = concat
    [ "Mo\n  "
    , show worlds, "\n  "
    , show ags, "\n  "
    , show val, "\n  "
    , show exp, "\n  "
    , show accs, "\n  "
    ]
```

Figure 4.11: ADT for EEMs

In Figure 4.13, the ADT for an EPL formula is shown. The Protocol Installation Operator is written as 'PU', the Observation Update Operator is written as 'PO' and the Knowledge operator is denoted as 'Kn'. In Figure 4.14, the most important function for our purpose, the function for model checking an EPL formula is shown. This function has been named as 'isValidAt'. It takes three inputs — an Epistemic

```
data PrtM state = Po
    [state]
    [Agent]
    [(state,PrtExpr)]
    [(Agent,Erel state)]
    deriving (Eq)

type PtdPrtM = (PrtM Int, Int)

instance Show state => Show (PrtM state) where
  show (Po objects ags prt accs) = concat
    [ "Po\n  "
    , show objects, "\n  "
    , show ags, "\n  "
    , show prt, "\n  "
    , show accs, "\n  "
    ]
```

Figure 4.12: ADT for Protocol Models

```
data EPLForm phi = Top
                 | Prp Proposition
                 | Ng (EPLForm phi)
                 | Conj [EPLForm phi]
                 | Disj [EPLForm phi]
                 | Kn Agent (EPLForm phi)
                 | PO ObExpr (EPLForm phi)
                 | PU PtdPrtM (EPLForm phi)
               deriving (Eq,Show)
```

Figure 4.13: ADT for an EPL formula

Expectation Model $\mathcal{M}$, a state s in $\mathcal{M}$, and an EPL formula φ in that order. In Figure 4.15, the function to obtain an updated EEM, $\mathcal{M}'$ from an input EEM, $\mathcal{M}$ after performing an update with a string w, is shown. In Figure 4.16 and Figure 4.17, the functions to perform protocol update are shown. We do not show the various helper functions here. The main function that performs this job is the 'prtProduct' function.

In the next subsection, code snippets for representation of DFAs and some func-

```
isTrueAt :: EpExpM Int -> Int -> EPLForm Int -> Bool
isTrueAt _ _ Top = True
isTrueAt (Mo _ _ val _ _) w (Prp p) = p `elem` apply val w
isTrueAt m w (Ng f)    = not (isTrueAt m w f)
isTrueAt m w (Conj fs) = all (isTrueAt m w) fs
isTrueAt m w (Disj fs) = any (isTrueAt m w) fs
isTrueAt m w (Kn ag f) = all (flip (isTrueAt m) f) (bl (relE ag m) w)
isTrueAt m w (PO e f)  = and [isTrueAt (updPO m s) w f | s <- strList]
  where
    expDFA  = eliminateDeads (filterMark (buildDfa (returnObX w m)))
    eDFA    =  eliminateDeads (filterMark (buildDfa e))
    pdList  = map (convertProduct eDFA) (findPartitions expDFA)
    strList = filter (/= [] ) $ map findStringDFA pdList
isTrueAt m@(Mo _ _ val _ _) w (PU (p@(Po _ _ prt _),t) f)
    = obRedux (ptConvert (apply prt t) (apply val w)) == ObDel
    || isTrueAt (prtProduct m p) (returnState m p (w,t)) f
```

Figure 4.14: Main function for Model checking

```
updPO :: (Show state, Ord state) => EpExpM state -> String -> EpExpM state
updPO m@(Mo states agents val obs rels) w =
    Mo states' agents val' obs' rels' where
    states' = [s | s <- states, (obRedux (obQ (apply obs s) (makeObs w))) /= ObDel ]
    val'    = [ (s, apply val s) | s <- states' ]
    obs'    = [(s, obRedux(obQ (apply obs s) (makeObs w)) )  | s <- states' ]
    rels'   = [ (ag, restrict states' r) | (ag,r) <- rels ]
```

Figure 4.15: Function for Update by Observation

```
convertCom :: (Ord a) => ComM a -> EpExpM Int
convertCom c@(Com worlds ags val obs accs) = (Mo worlds' ags val' obs' accs') where
    sMap    = buildPrdMap c
    worlds' = [xtractInt $ Map.lookup f sMap | f <- worlds]
    val'    = [(xtractInt $ Map.lookup (fst f) sMap, snd f) | f <- val ]
    obs'    = [(xtractInt $ Map.lookup (fst f) sMap, snd f) | f <- obs ]
    accs'   = [(fst f, nubPart (tupleListToErel (snd f) sMap)) | f <- accs]

prtProduct :: (Eq a,Show a, Ord a) => EpExpM a -> PrtM a -> EpExpM Int
prtProduct m p = convertCom $ prtProduct' m p
```

Figure 4.16: Functions for Protocol Update

```
prtProduct' :: (Eq a,Show a) => EpExpM a -> PrtM a -> ComM a
prtProduct' md@(Mo worlds ags val obs accs) pt@(Po objects pags prt paccs)
  = Com worlds' ags val' obs' accs' where
    worlds' = filter predicate [(s,t) | s <- worlds, t <- objects]
    predicate tup = obRedux (ptConvert (apply prt (snd tup)) (apply val (fst tup))) /= ObDel
    val'    = [(p,apply val (fst p))  | p <- worlds']
    obs'    = [(p, (obRedux (ptConvert  (apply prt (snd p)) (apply val (fst p))))) | p <- worlds']
    accs'   = [(ag, worldList ag worlds' accs paccs) | ag <- ags]
    worldList ag worlds' accs paccs = [(x,y) | x <- worlds', y <- worlds', func x y ag accs paccs == True]
    func x y ag accs paccs = if ((bl (apply accs ag) (fst x)) == (bl (apply accs ag) (fst y )))
                             && ((bl (apply paccs ag) (snd x)) == (bl (apply paccs ag) (snd y)))
                             then True else False
```

Figure 4.17: Functions for Protocol Update

tions defined upon them are shown.

4.2.5 DFA Construction, Representation and Functionalities

In Figure 4.19, the ADT for a DFA is shown. Due to practical considerations, DFA states are represented by lists of Integers rather than Integers. For example, the DFA in Figure 4.18 is represented by, $DFA\ [1,2][[1,2],[3]][([1,2],`b',[1,2]),([1,2],`a',[3]),([3],`b',[3])]$. In Figure 4.20, the main function that builds a DFA from an

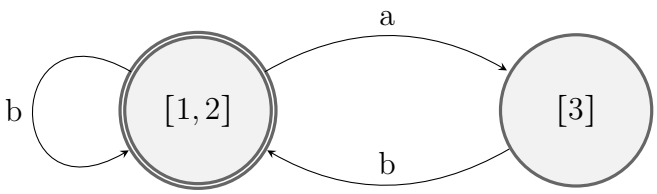


Figure 4.18: An Automaton

```
type Trans = ([Int], Char, [Int])

data ObDFA = ObD
             [Int]
             [[Int]]
             [Trans]
             [[Int]]
             [Char]
```

Figure 4.19: ADT for DFAs

observation expression is shown. This function implements the method mentioned in Section 3.3.2. For the sake of brevity, the helper functions for computation of syntax tree, nullable, firstpos, lastpos and followpos functions are not shown.

The utility functions to find the list of DFAs corresponding to pre-derivatives of an observation expression and finding a string in a DFA are shown in Figure 4.21. The main functions that perform these jobs are 'findPartitions' and 'findStringDFA' respectively.

4.2.6 Scalability and Complexity

The performance of this tool depends upon four factors: the number of states in the Epistemic Expectation Model $n_{\mathcal{M}}$, the structure of the DFA corresponding to the observation expression at the given state in the Epistemic Expectation Model,

```
buildDfa :: ObExpr -> ObDFA
buildDfa obX = ObD startState states trans finalStates symbols
  where
    auxObX      = obAugment obX
    obXTree     = buildTree auxObX
    nullable    = buildNullable obXTree
    firstPos    = buildFirst obXTree
    lastPos     = buildLast obXTree
    followPos   = buildFollow obXTree
    symbolMap   = buildSymbolMap obXTree
    symbols     = Map.keys symbolMap
    startState  = getStartState obXTree firstPos
    (states, trans) = buildStates_Trans startState symbolMap followPos symbols
    finalStates = getFinalStates states ((getNodeNumber obXTree) - 1)
```

Figure 4.20: Function for building a DFA

```
findString :: [Int] -> [[Int]] -> [Trans] -> String
findString s [] trans = []
findString s p trans = (findChar s (head p) trans) ++ (findString (head p) (tail p) trans)

findChar :: [Int] -> [Int] -> [Trans] -> String
findChar u v t = [head [c | (u,c,v) <- filter ((== v).lstTriplet) $ filter ((== u).fstTriplet) t ]]

findStringDFA :: ObDFA -> String
findStringDFA fa@(ObD start states trans finals symbols) = findString start path trans
  where
    path = extractPath $ findPath fa

findPartitions :: ObDFA -> [ObDFA]
findPartitions fa@(ObD start states trans final symbol) =
      [ObD start states trans [final'] symbol | final' <- filter (not.( `elem` final)) states ] ++ [fa]
```

Figure 4.21: DFA utility functions

the number of states in the Protocol Model $n_{\mathcal{A}}$ and the length l_p, of the protocol expression at the designated state in the protocol model.

Of these four, discerning the effect of the number of states in the Epistemic Expectation Model, that of the Protocol Model and the length of the protocol expression, for a formula that does not contain the Observation Update Operator but contains the Protocol Update operator is straightforward — it is bounded by $\mathcal{O}(n_{\mathcal{M}} n_{\mathcal{A}} l_p)$.

But determining the effect of the structure of the DFA is not straightforward. We leave this for future work which would help us to obtain a proper performance benchmarking.

4.3 An Example of Model Checking

In this section we consider the **The Valentine's Day problem** mentioned in Section 2.3.1. To recall, the problem is as following: *Three friends Carl, Bob and Alice are sitting in a pub. Carl says to Bob, who is his childhood friend, "On Valentine's day I went to this pub with Mike and Sara. It was a crazy night!". This immediately catches the attention of Alice, who is secretly in love with Mike. She asks: 'What happened?' Carl winks to Bob, who know each other like the back of their hands, and replies, "Nothing". Bob immediately realizes that indeed nothing had occurred, whereas Alice, who doesn't know what that wink stands for, becomes unsure about the events of that night.*

The initial model is called $\mathcal{M}$ and is depicted in Figure 4.22 and the protocol model, $\mathcal{A}$ is depicted in Figure 4.23. In Figure 4.24 and Figure 4.25, the way

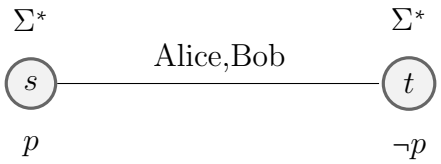


Figure 4.22: Initial Epistemic Expectation Model $\mathcal{M}$

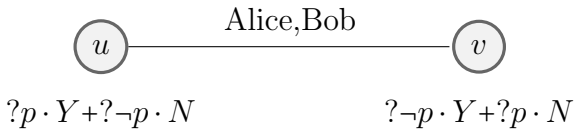


Figure 4.23: Protocol Model

these models are provided as inputs in an **EPLMC** session are shown — the initial model, $\mathcal{M}$ is denoted by the variable 'initM', and the protocol model $\mathcal{A}$ is denoted by the variable 'ptcl'.

As mentioned before, the accessibility relations are represented with partitions — lists of lists in Haskell. We denote Alice as Agent 1 and Bob as Agent 2, the worlds s and t are given with integers 1 and 2 and the states u and v of the protocol model are given by the integers 3 and 4. It is possible in EPLMC to denote these states with their own custom data types, rather than with integers, but for the sake of simplicity, we use integers. The actual state in $\mathcal{M}$ is t.

In Figure 4.26, the model $\mathcal{M} \times \mathcal{A}$ is depicted. In Figure 4.27, the result of performing this update, $\mathcal{M} \times \mathcal{A}$ in EPLMC is shown.

Now we show in Figure 4.28, the resultant model, $(\mathcal{M} \times \mathcal{A})\,|_N$ after updating the model $\mathcal{M} \times \mathcal{A}$ with the string "N". In Figure 4.29, representation of this model

```
*EPLMC> :l EPLMC.hs
[ 1 of 12] Compiling BoolForm          ( BoolForm.hs, interpreted )
[ 2 of 12] Compiling BoolToken         ( BoolToken.hs, interpreted )
[ 3 of 12] Compiling BFormParser       ( BFormParser.hs, interpreted )
[ 4 of 12] Compiling ObExpr            ( ObExpr.hs, interpreted )
[ 5 of 12] Compiling Ob2DFA            ( Ob2DFA.hs, interpreted )
[ 6 of 12] Compiling ObToken           ( ObToken.hs, interpreted )
[ 7 of 12] Compiling ObParser          ( ObParser.hs, interpreted )
[ 8 of 12] Compiling PrtExpr           ( PrtExpr.hs, interpreted )
[ 9 of 12] Compiling PrtToken          ( PrtToken.hs, interpreted )
[10 of 12] Compiling PrtParser         ( PrtParser.hs, interpreted )
[11 of 12] Compiling Util              ( Util.hs, interpreted )
[12 of 12] Compiling EPLMC             ( EPLMC.hs, interpreted )
Ok, 12 modules loaded.
*EPLMC> (y,n) = (ObLit 'y', ObLit 'n')
*EPLMC> (s,t,u,v) = (1,2,3,4)
*EPLMC> sigmaS = ObStar (ObAlt y n)
*EPLMC> (p,np) = (P 1,P 2)
*EPLMC> (pb,npb) = (PrpB p,PrpB np)
*EPLMC> pe1 = PtAlt (PtCat (PtTest pb) (PtLit 'y')) (PtCat (PtTest npb) (PtLit 'n'))
*EPLMC> pe2 = PtAlt (PtCat (PtTest npb) (PtLit 'y')) (PtCat (PtTest pb) (PtLit 'n'))
*EPLMC> (alice,bob) = (Agt 1,Agt 2)
*EPLMC>
```

Figure 4.24: Setting up the environment for an EPLMC session — input of the states, agents, propositions, Observation Expressions and Protocol Expressions

```
*EPLMC> ptcl
Po
  [3,4]
  [Agent 1,Agent 2]
  [(3,((?P1.y)+(?P2.n))),(4,((?P2.y)+(?P1.n)))]
  [(Agent 1,[[3,4]]),(Agent 2,[[3],[4]])]

*EPLMC> initM
Mo
  [1,2]
  [Agent 1,Agent 2]
  [(1,[P1]),(2,[P2])]
  [(1,(y+n)*),(2,(y+n)*)]
  [(Agent 1,[[1,2]]),(Agent 2,[[1,2]])]

*EPLMC> installedM = prtProduct initM prtcl
```

Figure 4.25: Internal Representation of $\mathcal{M}$ — denoted by the variable 'initM' and $\mathcal{A}$ — denoted by 'ptcl'

$(\mathcal{M}\times\mathcal{A})\,|_N$ as computed in EPLMC is shown. This is denoted by 'observeUpdateM' in this session. The symbol '@' is used to denote ϵ.

Now in Figure 4.30, an example of providing a formula as input and checking its validity in an EPLMC session is shown — the variable 'phi' denotes the formula $\varphi = [!\mathcal{A}_u][N](K_{Bob}\neg p \wedge \neg K_{Alice}\neg p)$. It is seen from Figure 4.30, that $\mathcal{M}, t \vDash \varphi$ since the function for model checking in EPLMC, 'isTrueAt', when provided with input initM, t and phi, returns True. From Figure 4.29 and Figure 4.28, it follows that

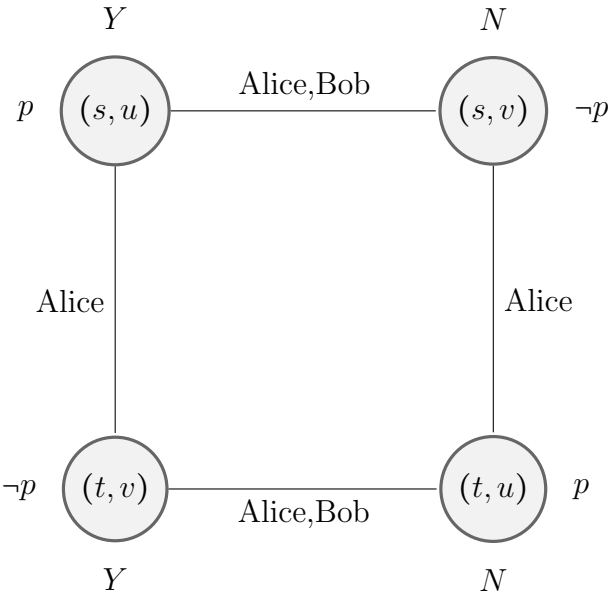


Figure 4.26: Updated Expectation Model

```
*EPLMC> installedM
Mo
  [1,2,3,4]
  [Agent 1,Agent 2]
  [(1,[P1]),(2,[P1]),(3,[P2]),(4,[P2])]
  [(1,y),(2,n),(3,n),(4,y)]
  [(Agent 1,[[1,2,3,4]]),(Agent 2,[[1,3],[2,4]])]

*EPLMC>
```

Figure 4.27: The Updated Model, 'initM' × 'ptcl'

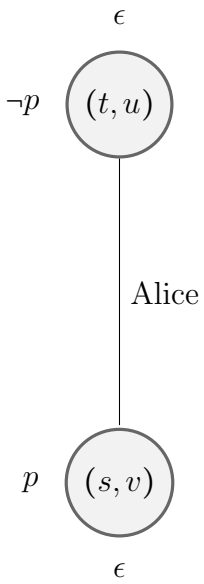


Figure 4.28: Expectation Model $(\mathcal{M} \times \mathcal{A})\,|_N$

indeed $(\mathcal{M} \times \mathcal{A})\,|_N \vDash K_{Bob}\neg p \wedge \neg K_{Alice}\neg p$, as at state (t, u), $K_{Bob}\neg p$, and $\neg K_{Alice}\neg p$. This is because Alice can not distinguish between the two states as there remains an

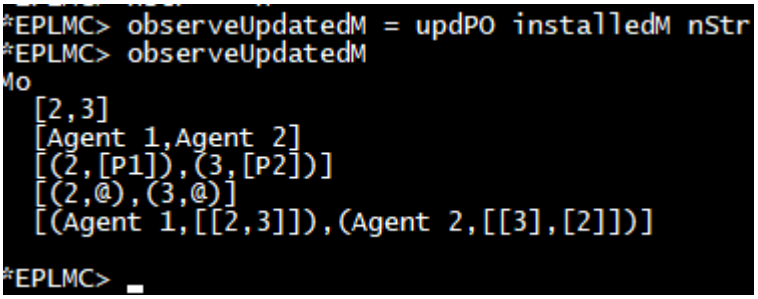


Figure 4.29: Expectation Model after observation of "N"

edge labeled Alice between the two worlds (t, u) and (s, v). Similarly in Figure 4.29, for Alice (Agent 1), the accessibility relation (in the form of a list of lists of states) is [[1,2]] (both states are in the same partition), and thus for her the two states are indistinguishable.

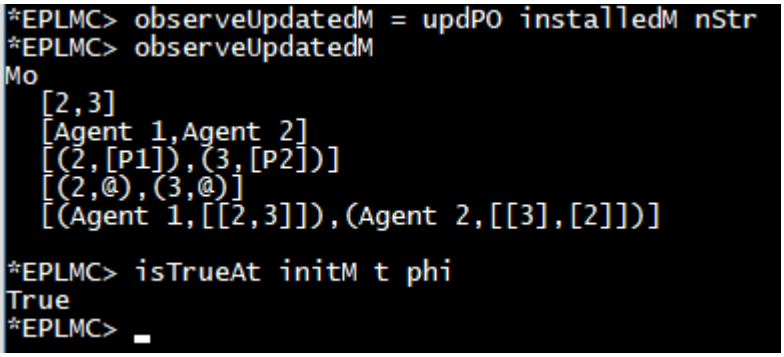


Figure 4.30: Model Checking the formula φ in EPLMC with respect to $(\mathcal{M}, t)$

Chapter 5

Conclusions

In this work the main objective was to design and implement a framework for model checking logics of protocols, namely POL and EPL. In Section 6.1, the main ingredients of this work are summarized, and in Section 6.2, the scope and direction of future work are presented.

5.1 Summary

We started with a preliminary discussion of the Model Checking Problem in Epistemic Logic and presented an overview of the Epistemic Logic paradigm with an emphasis upon the framework of Dynamic Epistemic Logic.

In the next chapter, an introduction to the Logic of Public Observations and Epistemic Protocols was presented. The next two chapters revolved around the following two goals, 1) formulation of an algorithmic framework for model checking Logics of Public Observations and Epistemic Protocols, 2) design and implementation of a model checking tool using these approaches. To achieve these goals,

- A constructive proof of decidability of POL is attained.
- An algorithm framework for POL and EPL is formulated.
- A model checking tool written in Haskell called EPLMC for this algorithmic framework is developed.

There remains a plethora of scope for further development which we discuss in Section 6.2

5.2 Scope for Future Work

There still remains much room for improvement in EPLMC. As of now, it is designed as a module. Although it is capable of performing the required model checking task, it can not be invoked on its own. Functionalities for efficient handling of I/O need to be added to build a full fledged stand-alone software. For this

purpose, a description language for input of Epistemic Expectation Models and Protocol Models, and a mini-compiler for that language need to be implemented. Another line of investigation would be to investigate optimizations for further performance improvement.

In yet another direction, symbolic encoding of Epistemic Expectation Models remains an open problem. This is useful in order to mitigate state explosion problem. To this end, various approaches based upon BDDs (Binary Decision Diagram) and ASP (Answer Set Programming) may be adopted. We plan to pursue these approaches further to formulate a symbolic framework and a corresponding software.

Chapter 6

Conclusions

In this work the main objective was to design and implement a framework for model checking logics of protocols, namely POL and EPL. In Section 6.1, the main ingredients of this work are summarized, and in Section 6.2, the scope and direction of future work are presented.

6.1 Summary

We started with a preliminary discussion of the Model Checking Problem in Epistemic Logic and presented an overview of the Epistemic Logic paradigm with an emphasis upon the framework of Dynamic Epistemic Logic.

In the next chapter, an introduction to the Logic of Public Observations and Epistemic Protocols was presented. The next two chapters revolved around the following two goals, 1) formulation of an algorithmic framework for model checking Logics of Public Observations and Epistemic Protocols, 2) design and implementation of a model checking tool using these approaches. To achieve these goals,

- A constructive proof of decidability of POL is attained.
- An algorithm framework for POL and EPL is formulated.
- A model checking tool written in Haskell called EPLMC for this algorithmic framework is developed.

There remains a plethora of scope for further development which we discuss in Section 6.2

6.2 Scope for Future Work

There still remains much room for improvement in EPLMC. As of now, it is designed as a module. Although it is capable of performing the required model checking task, it can not be invoked on its own. Functionalities for efficient handling of I/O need to be added to build a full fledged stand-alone software. For this

purpose, a description language for input of Epistemic Expectation Models and Protocol Models, and a mini-compiler for that language need to be implemented. Another line of investigation would be to investigate optimizations for further performance improvement.

In yet another direction, symbolic encoding of Epistemic Expectation Models remains an open problem. This is useful in order to mitigate state explosion problem. To this end, various approaches based upon BDDs (Binary Decision Diagram) and ASP (Answer Set Programming) may be adopted. We plan to pursue these approaches further to formulate a symbolic framework and a corresponding software.

Bibliography

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006.

[Azi10] Haris Aziz. Multiagent systems: Algorithmic, game-theoretic, and logical foundations by y. shoham and k. leyton-brown cambridge university press, 2008. *SIGACT News*, 41(1):34–37, March 2010.

[BCM+92] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 1020 states and beyond. *Information and Computation*, 98(2):142 – 170, 1992.

[BMS98] Alexandru Baltag, Lawrence S. Moss, and Slawomir Solecki. The logic of public announcements, common knowledge, and private suspicions. In *Proceedings of the 7th Conference on Theoretical Aspects of Rationality and Knowledge*, TARK '98, page 43–56, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[Bry86] Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[CW96] Edmund Clarke and Jeannette Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28, 12 1996.

[DvdHK07] Hans van Ditmarsch, Wiebe van der Hoek, and Barteld Kooi. *Dynamic Epistemic Logic*. Springer Publishing Company, Incorporated, 1st edition, 2007.

[Eme08] E. Allen Emerson. The beginning of model checking: A personal perspective, 2008.

[FHMV95] R Fagin, JY Halpern, Y Moses, and MY Vardi. Reasoning about knowledge mit press. *Cambridge, MA, London, England*, 1995.

[Gat18] Malvin Gattinger. *New Directions in Model Checking Dynamic Epistemic Logic*. PhD thesis, University of Amsterdam, 2018. Available at https://malv.in/phdthesis.

[HMU07] J.E. Hopcroft, R. Motwani, and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Pearson/Addison Wesley, 2007.

[Pla07] Jan Plaza. Logics of public communications. *Synthese*, 158:165–179, 08 2007.

[Pnu77] A. Pnueli. The temporal logic of programs. In *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, pages 46–57, Los Alamitos, CA, USA, oct 1977. IEEE Computer Society.

[PR03] Rohit Parikh and Ramaswamy Ramanujam. A knowledge based semantics of messages. *Journal of Logic, Language and Information*, 12(4):453–467, 2003.

[SSL07] Kaile Su, Abdul Sattar, and Xiangyu Luo. Model checking temporal logics of knowledge via obdds. *The Computer Journal*, 50(4):403–420, 2007.

[TB19] Nicolas Troquard and Philippe Balbiani. *"Propositional Dynamic Logic", The Stanford Encyclopedia of Philosophy, Edward N. Zalta (ed.)*, March 2019. Available at https://plato.stanford.edu/archives/spr2019/entries/logic-dynamic.

[vDGVW14] Hans van Ditmarsch, Sujata Ghosh, Rineke Verbrugge, and Yanjing Wang. Hidden protocols: Modifying our expectations in an evolving world. *Artificial Intelligence*, 208:18 – 40, 2014.

[vE14] Jan van Eijck. *"DEMO-S5"*, 2014. Available at https://homepages.cwi.nl/~jve/software/demo_s5.

[Wan11] Yanjing Wang. Reasoning about protocol change and knowledge. volume 6521, pages 189–203, 01 2011.