Protocol Synthesis in the Dynamic Gossip Problem

Leo Poulson

May 10, 2019

Contents

1 Introduction **3**
- 1.1 Existing Software . . . 4
- 1.2 Contributions . . . 5
- 1.3 Road Map . . . 5

2 Background **6**
- 2.1 Dynamic Epistemic Logic . . . 6
  - 2.1.1 Epistemic Logic . . . 6
  - 2.1.2 Event Models . . . 8
- 2.2 The Gossip Problem . . . 9
  - 2.2.1 Dynamic Gossip . . . 10
  - 2.2.2 Protocols . . . 10
  - 2.2.3 Formalisation . . . 10
- 2.3 Planning . . . 12
  - 2.3.1 Epistemic Planning . . . 13
- 2.4 Existing Tools . . . 13
  - 2.4.1 DEMO-S5 . . . 14
  - 2.4.2 Gossip . . . 14
  - 2.4.3 SMCDEL . . . 14
- 2.5 Uniform Strategies . . . 15
  - 2.5.1 Game Arenas . . . 15
  - 2.5.2 Transducers . . . 16
  - 2.5.3 The Powerset Arena . . . 16
  - 2.5.4 Lifting Transducers . . . 17

3 Algorithm **18**
- 3.1 $\mathcal{ME}^*$ . . . 20
- 3.2 The Powerset Construction . . . 21
  - 3.2.1 Event Transducers . . . 21
  - 3.2.2 Transducer Composition . . . 21
  - 3.2.3 The Construction . . . 22
- 3.3 Combining Automata . . . 24
- 3.4 Searching . . . 25

4 Implementation **26**
- 4.1 `Model.hs` . . . 26
- 4.2 `FSM.hs` and `FST.hs` . . . 28
- 4.3 `ME.hs` . . . 28
- 4.4 `Powerset.hs` . . . 29

5 Evaluation **31**
5.1 Plan for Testing . . . 31
5.2 Testing Frameworks . . . 32
5.3 Correctness Results . . . 32
5.4 Profiling . . . 33
5.4.1 Profiling Framework . . . 33
5.4.2 Profiling Results . . . 34
5.5 Results Analysis . . . 36
5.5.1 `Models` . . . 36
5.5.2 Search Strategies . . . 37
5.5.3 `BFSM` . . . 37
5.5.4 Monotonicity . . . 37
5.5.5 ANY and related protocols . . . 38

6 Conclusion **39**
6.1 Future Work . . . 39

Chapter 1

Introduction

Often we come across situations, processes or phenomena in the real world that we want to reason about computationally. These real-world systems tend to contain lots of information that are useful to us - the state of knowledge of a certain agent, the wind speed on a certain day - and lots of information that is less useful - what an agent in question had for breakfast this morning, which day of the week it is. We use a *model* to abstract away the unimportant information from a system and simplify the real world.

This use of mathematical models for real-world systems gives rise to the task of *model checking*. This is the process of taking some model $\mathcal{M}$ and some formula ϕ and finding if the model $\mathcal{M}$ satisfies the formula ϕ. **TODO add to this?**

Planning is the task of computing a set of actions to take some state within a model to a successful state. In such a model we have a set of *actors*, who will perform these actions that we search for. Put forward in Ghallab, Nau, and Traverso 2004, planning covers a wide range of applications; from making a robot enter a room to synchronising multiple computers working on the same document. In this project we concern ourselves with a special case of planning; namely *epistemic planning*. In this we begin with a state embodying the knowledge states for a set of agents. We then want to compute a set of actions the agents may perform in order to take the knowledge state of the agents to a successful one.

Epistemic planning is a relatively young area of research, first put forward in Bolander and Birkegaard 2011. It is a specialisation of general planning, concerning itself with the *knowledge* of agents in a model. It rests upon work done in Dynamic Epistemic Logic, which is a logic used to reason about knowledge and how it is changed by actions. Despite epistemic planning being proved to be undecidable in a lot of cases [1], it is decidable for models where the preconditions and consequences of events are *propositional*[2] (Yu, Wen, and Liu 2013). It is this fragment of the planning problem that we work on in this project, due to its decidability property. This result means that interest in the area has been maintained since its birth (Aucher, Maubert, and Pinchinat 2014), leading to the work that this project uses.

The *gossip problem* is a problem regarding peer-to-peer information sharing. A set of agents start out with some *secret information*, and their goal is to transfer this information across the network, such that every other agent finds out their secret. We call an agent who knows the secret of every other agent an *expert*; we want to reach a state where every agent is an expert. When two agents communicate they tell each other all of the secrets they know; hence the slightly frivolous but very apt title of the *gossip* problem.

The gossip problem was first put forward in Tijdeman 1971, and was such that each agent began with the phone number of every other agent. We can hence visualise the set of agents as nodes and the fact of agent a knowing the phone number of b as a directed edge, and so such a formulation would be a complete graph as in Figure 1.1a.

Recently a variation of this problem has been studied, entitled the *dynamic* gossip problem (van Ditmarsch et al. 2018, van Ditmarsch et al. 2016). This is a popular research topic, with applications in the study of epidemics and information discovery (Haeupler et al. 2016), among others. In these scenarios our network may model some peer to peer computer network where our agents are computers, and the goal is to find the IP addresses of all of the other computers in the network; or a social network, where our nodes are people who want to connect with

[1] It is undecidable for models where the number of agents is greater than 1 (Aucher and Bolander 2013), and also in single-agent models where the accessibility relation (the relation between indistinguishable states) is not an equivalence relation.

[2] This means formulae that do not include the modal knowledge operator K which we come to define later.

(a) *An example of a gossip graph in the classical formulation.*

(b) *An example of an incomplete gossip graph.*

Figure 1.1: *Some exemplar gossip graphs.*

every other user in their circle of friends.

In this, we begin with each agent knowing some subset of the phone numbers of every other agent. Hence a starting configuration of the dynamic gossip problem may look like Figure 1.1b. When two agents converse on the phone, they also exchange all of the phone numbers that they know as well as all of the secrets. As such, the graph's edges increase in number as phone calls occur.

There are many other variants of the gossip problem studied in the literature, and many potential decisions to make when defining the way the agents in the gossip behave. We list the decisions taken in this thesis here;

- We choose to use the *synchronous* gossip problem. The asynchronous gossip problem lets two agents call each other at the same time, and has no way of controlling when agents can communicate each other. In contrast, the synchronous version assumes the control of some global clock that lets agents know when they may call each other, and when a call occurs. This has implications that are later discussed in Section 2.2.3.
- We assume the existence of some global, omniscient *observer* who decides when the successful formula is satisfied. In the literature the gossip problem is studied in its purely distributed form - in a practical distributed system we cannot assume the existence of such a system to do this. It goes without saying that systems of this kind are more difficult to reason about and work with. Given the difficulty of the task being undertaken in this project we chose to make it easier for ourselves here.
- When agents communicate, they tell each other *everything they know*. Situations in which agents can communicate little information are studied[3], however for the sake of ease and simplicity we choose to work with the fragment in which agents share all of their knowledge when communicating.

Recall that earlier we said that the epistemic planning problem is decidable for epistemic models where the preconditions and consequences of events are *propositional*. Gossip models, like those in Figure 1.1b, are an example of these[4]. Due to this, we are capable of performing planning on gossip models. In this thesis we put forward a process with which this can be performed, and an example implementation of this.

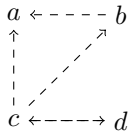


Figure 1.2

Then given an input pair $\mathcal{M}$, which is a gossip graph, and some formula ϕ we are given the sequence of calls $c_1, c_2, \ldots, c_n$ whose occurence on $\mathcal{M}$ will reach a state in which ϕ is modelled. For example, consider the graph in Figure 1.2, with the winning formula $K_a \forall_{i \in Ag} E_i$[5]. Then the returned string of calls will be ba, ca, ad, ab, ac. Later in this thesis we go through the process with which this call sequence is produced, and explain the decisions taken.

1.1 Existing Software

As research work is done into a topic, it is only natural that software is developed to support it. The three related pieces of software are **DEMO-S5 SMCDEL** and **GithubGossip** These three tools are all *model checkers* for dynamic

[3]With applications in practical networking; we can imagine a network where communication over channels is expensive, and as such agents want to minimise the amount of information transferred.

[4]In Section 2.2.3 we show this, by giving a way to convert between the two algorithmically.

[5]In this thesis we abbreviate the statement "agent i is an expert" to be the proposition E_i. We also abbreviate the statement "all agents are experts" to the proposition E, although $\forall_{i \in Ag} E_i$ is semantically equivalent.

epistemic logic. A model checker is a program that, given a model $\mathcal{M}$ and a formula ϕ, tells the user whether or not $\mathcal{M} \models \phi$. Given the setting, they specifically answer the question "given an initial state and a set of events, does the state reached by the occurrence of these events at the initial state satisfy some property?".

We can see model checking as the *dual* of planning; in planning we want to find such a sequence of events, whilst model checking verifies that these events do indeed bring the system to a successful state. **GithubGossip** is also capable of planning, however in a naïve way; it enumerates the set of all possible calls and then checks if any of these are successful. As we see later, this is a very inefficient method for planning.

Despite this, there does not exist any software to solve the epistemic planning problem as envisioned in the literature; indeed, the only software to solve any epistemic planning problem is restricted to just gossip models.

1.2 Contributions

At a high level, our aim in this project is to put forward a process to solve the planning problem for epistemic models and propositional event models. This is motivated by the Dynamic Gossip problem; despite all of the work done in the area of Gossip little has been done on the topic of *planning* for the Gossip problem. Dynamic Gossip can be modelled perfectly by epistemic models and propositional event models, and hence is a perfect case study for the tool we will develop.

We have four major contributions in this thesis, as follows:

- The first contribution is the background chapter. The literature on the topic spans from epistemic logic to game theory, and is far outside the reach of the standard computer science curriculum, We present the content in a simple, straightforward manner, such that an undergraduate reader may understand all of the concepts involved.
- Our second contribution is the derivation of an algorithm to solve the epistemic planning problem, as a specialisation of the sketch given in Aucher, Maubert, and Pinchinat 2014. Through a scrutinization of this theory, which involves the very general theory of two-person games, we extract a conceptually clean procedure that can be understood in terms of simple automata theory.
- As a testament to the functionality of the algorithm designed, we provide an implementation of it in Haskell. This is the first implementation of a planning algorithm for general epistemic models.
- Finally, we give an empirical evaluation of the software developed and the process designed. In this evaluation we inspect the correctness of the system, and perform a study into the program's efficiency in terms of time and space. As part of this, we implement a framework with which we automatically perform these tests. We also build a framework to aid with profiling the code.

1.3 Road Map

In Section 2, we introduce, formalise and explain the concepts and tools needed to understand the work in the rest of the thesis. This constitutes the first of our challenges; collecting and displaying all of the surrounding literature in a pleasant, simple manner is a goal of this thesis and this section achieves this.

In Section 3 we put forward the algorithm we designed in order to solve the epistemic planning problem. This uses a simplified version of the process put forward in Section **??**. We also give an exemplar execution of the algorithm, in order to show how it functions.

Section 4 discusses the Haskell implementation of the algorithm designed in Section 3. We discuss the design decisions taken throughout the implementation, including motivation for the use of Haskell to solve this problem.

In Section 5 we evaluate the algorithm and its implementation. To do this we use **GithubGossip** as both a model checker, to validate our results returned from our planning tool, and also as a tool capable of planning, in order to analyse the space and time efficiency of our tool.

In Section 6 we conclude the thesis, and lay out some possible future work that could be done in the area.

Chapter 2

Background

2.1 Dynamic Epistemic Logic

2.1.1 Epistemic Logic

Epistemic Logic is the logical language that we use to reason about knowledge. It is a modal logic; this is a class of logics that supplement the language of propositional logic[1] with some operator $\Box$[2]. Modal logic can be used to model the passage of time, knowledge, obligation or any other modality. For example, $\Box\phi$ in temporal logic can be read as "at all points in the future, ϕ holds"; in deontic (obligation) logic, we can read $\Box\phi$ as "it is morally necessary that ϕ holds". We see that the addition of the $\Box$ operator provides us with much more expressivity than the standard language of propositional logic.

In this thesis, we concern ourselves with *epistemic* logic; here we interpret $\Box\phi$ as "it is known that ϕ is true". However, we change the operator slightly; we index it with the name of an agent, and we write it as $K_i\phi$[3] Hence, we interpret $K_i\phi$ as "Agent i knows that ϕ is true.".

The essential reference on epistemic logic is **ReasoningAboutKnowledge** and it is from here that most of the information in this section comes.

If we have some set of agents A and some set of propositions Λ, then we define the language of epistemic logic over this set of propositions, $\mathcal{L}(\Lambda)$, with the following BNF;

$$\phi ::= \top \mid p \mid \neg\phi \mid \phi \wedge \phi \mid K_i\phi$$

where $p \in \Lambda$ and $i \in A$. We can also define the dual to K, in the classic way as $\widehat{K}_i\phi ::= \neg K_i \neg\phi$. We read this as "agent i considers it possible that ϕ is true". We can give our epistemic logic a semantics through use of *Kripke models*.

We refer to $\mathcal{L}(\Lambda)$ without the knowledge operator K_i as $\mathcal{L}_P(\Lambda)$. This is just the language of propositional logic, as mentioned before.

A Kripke model $\mathcal{M}$ over a set of agents I and a set of propositions Λ is a triple (W, R, V), where W is a set of worlds, R is a set of binary relations over W indexed by an agent such that $R_i \subseteq W \times W$, and $V : W \to \mathcal{P}(\Lambda)$ is a valuation function that associates to every world in W some set of propositions that are true at it.

For epistemic logic, we think of R_i as being the set of pairs of worlds that an agent i cannot distinguish between, and thus considers *possible*. This notion of epistemic possibility is essential, and is key to understanding the rest of the thesis. As an example, consider waking up in the morning with the curtains drawn. You cannot distinguish between the world in which it's sunny outside and the world in which it's cloudy outside; if it were sunny outside you could not tell, and likewise if it were cloudy outside. We label these two worlds ☼and ☁, and at this state (☼, ☁) $\in R$ [4]. Hence one considers both worlds *possible*.

However when you get out of bed, open the curtains and see that it is indeed another cloudy day in Bristol you can now distinguish between the current, cloudy world and a world in which it is sunny - formally, (☼, ☁) $\notin R$. After one performs this action one no longer considers it possible that it is sunny.

[1] By which we mean the language defined by the grammar $\phi ::= \top \mid p \mid \neg\phi \mid \phi \wedge \phi$, as well as the abbreviations $\bot, \vee, \to$ as is standard.
[2] In modal logic, $\Box$ usually means "necessarily" and $\Diamond$ means "possibly".
[3] Where the K stands for **K**nows.
[4] Respectively (☁, ☼) $\in R$; we later clarify that our relations are all symmetric.

In our semantics, we use this relation to define knowledge in terms of possibility; agent i knows that something is true if it is true at all of the worlds that i cannot distinguish between.

When giving a semantics to a formula on a Kripke model, we need to use a *pointed Kripke model*. This is just a pair $(\mathcal{M}, w)$ where w is a world of $\mathcal{M}$. Then we read $(\mathcal{M}, w) \models \phi$ as "w satisfies ϕ". We define the evaluation as follows:

$$
\begin{aligned}
&(\mathcal{M}, w) \models \top \\
&(\mathcal{M}, w) \models p \text{ iff } p \in V(w) \\
&(\mathcal{M}, w) \models \neg\phi \text{ iff } (\mathcal{M}, w) \not\models \phi \\
&(\mathcal{M}, w) \models \phi \wedge \psi \text{ iff } (\mathcal{M}, w) \models \phi \text{ and } (\mathcal{M}, w) \models \psi \\
&(\mathcal{M}, w) \models K_i\phi \text{ iff for all } v \text{ such that } (w, v) \in R_i, (\mathcal{M}, v) \models \phi
\end{aligned}
$$

We now shed some light on the semantics of $K_i\phi$. Recall that agent i *knows* a proposition to be true if it holds at all worlds i cannot distinguish from its current one. The semantics expresses this; we see that we go to each world v such that $(w, v) \in R_i$ and check that ϕ holds there. This matches up with our specification, where a formula $K_i\phi$ holds if at all indistinguishable worlds from the current one ϕ holds.

The knowledge operator K is given several properties. These are done to make it better reflect the properties of human knowledge in the real world - thus bringing the model closer to reality - and also in order to make it more computationally tractable[5]. We make all of our relations R_i equivalence relations. This means three things;

- R_i is *reflexive*; for all $w \in W$, $(w, w) \in R_i$.
- R_i is *symmetric*; for all $w, v \in W, (w, v) \in R_i$ iff $(v, w) \in R_i$.
- R_i is *transitive*; for all $w, v, u \in W$ if $(w, v) \in R_i$ and $(v, u) \in R_i$, then $(w, u) \in R_i$.

This is done in order to convey that agent i considers world v possible from world w if in both w and v agent i has the same information; that is, they are indistinguishable to the agent.

It is identical to say that our relations R_i are equivalence relations or to say that our model is an S5 model. This is defined as a model in which the modal operator K obeys the following axioms:

- K: $K(\phi \to \psi) \to (K\phi \to K\psi)$
- T: $K\phi \to \phi$
- 5: $\widehat{K}\phi \to K\widehat{K}\phi$

These axioms hold independent of which agent's knowledge we are reasoning about; they are simply a property of knowledge in the model.

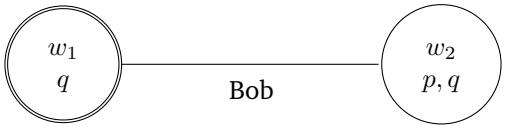


Figure 2.1

In Figure 2.1, we can see an exemplar Kripke Model. In this epistemic model[6], we have two agents, Alice and Bob, and two worlds, w_1 and w_2. Alice can distinguish between w_1 and w_2, and as such has no lines on the diagram. The implications of this with respect to R_{Alice} are that $(w_1, w_2) \notin R_{Alice}$. However Bob cannot, hence the line connecting w_1 and w_2. At w_1 the proposition q is true, and at w_2 propositions p and q are true. Our "actual" world is w_1, and as such it is double ringed. This means that any formula ϕ will be evaluated as $(\mathcal{M}, w_1) \models \phi$.

To show some examples, let's evaluate some formulae on this model.

- $(\mathcal{M}, w_1) \models q$ as at w_1, q is true.

[5] Recall that the epistemic planning problem is undecidable for single-agent models with an accessibility relation which is not an equivalence relation.

[6] In this thesis we use the names "epistemic model" and "Kripke model" interchangeably; they mean exactly the same things.

- $(\mathcal{M}, w_1) \models K_{\text{Alice}} q$. To check that this is true, let's consider all of the worlds Alice considers possible from w_1. This entails computing the set of worlds $\{v \mid (w_1, v) \in R_{Alice}\}$. The only world is w_1 itself, and we know from above that $(\mathcal{M}, w_1) \models q$. Hence $K_{\text{Alice}} q$ holds here.
- $(\mathcal{M}, w_1) \models \neg K_{\text{Bob}} (p \wedge q)$. We consider all the worlds that Bob considers possible from w_1. This is both w_1 and w_2. But we see that $(\mathcal{M}, w_1) \not\models (p \wedge q)$, since $(\mathcal{M}, w_1) \not\models p$.

2.1.2 Event Models

Public announcement logic (PAL), first described in **PAL** was the first development in epistemic logic to support *information change*. PAL supports truthful, public communications, and supplements the language of epistemic logic with an operator $[!\psi]$. This operator takes us to a new model consisting of only worlds in which ψ holds. For example, consider a group of agents playing some card game. No agent knows what the others hold in their hands; hence the set of worlds in the model is vast, as the agents consider it possible that any player could have any card. Then agent a publicly announces that he holds the 3 of spades; hence the worlds of the model are restricted to those on which the proposition "agent a holds the 3 of spades" is true.

The problem with PAL is clear; we may only model truthful, public communications. What if agent a were lying, or agent a was able to communicate with agent b secretively, without the other agents realising? Clearly PAL is insufficient to model all of the complicated types of communication that may occur. It also only is capable of modelling communications; often, the occurence of some *physical* action can change the knowledge states of agents involved in a system[7].

To model these more complicated events, we use *event models*. These treat events in a very similar way to how Kripke models treat worlds; we think of a set of possible events that can occur, and encode the pairs events that an agent can and cannot tell apart. Just like worlds, it is possible for events to occur that an agent cannot tell apart; for example, consider a coin being tossed, and the result being hidden from an agent a. Then agent a cannot distinguish between the event in which the coin landed face up, and the event in which it landed face down. As another example, consider a group of 4 people who can all call each other on the phone. Agent a knows that a call occurs between the other agents, but does not know who between; hence a cannot distinguish between a call from b to c and c to d, and so on.

We give the modern definition of event models, as in **MalvinThesis** and Aucher, Maubert, and Pinchinat 2014. An event model $\mathcal{E}$ is a tuple $(E, P, \mathsf{pre}, \mathsf{post})$. E is a finite set of events; P is a set of relations P_i for each $i \in Ag$, such that $P_i \subseteq E \times E$. As before, we make all relations P equivalence relations. $\mathsf{pre} : E \to \mathcal{L}(\Lambda)$ is the *precondition function*; given an event $e \in E$, it returns to us a formula that must be true in order for e to occur. $\mathsf{post} : E \times \Lambda \to \mathcal{L}(\Lambda)$ is the *postcondition function*; given an event $e \in E$ and a proposition $p \in \Lambda$, it returns to us some formula that had to be true at the prior state s in order for p to be true after event e occurs at s. We will later give examples for these two functions.

Updating a Kripke model $\mathcal{M}$ with an event model $\mathcal{E}$ is gives us another Kripke model $\mathcal{M} \times \mathcal{E} ::= (W', R', V')$, where:

$$
\begin{aligned}
W' &::= \{(w, e) \in W \times E \mid (\mathcal{M}, w) \models \mathsf{pre}(e)\} \\
R'_i &::= \{((w, e), (v, f)) \mid (w, v) \in R_i, (e, f) \in P_i, (w, e), (v, f) \in W'\} \\
V'(w, e) &::= \{p \in \Lambda \mid (\mathcal{M}, w) \models \mathsf{post}(e, p)\}
\end{aligned}
$$

We can see that the precondition function is used to ask the question "is it possible for the event in question to occur at the given world", whilst the postcondition function lets us compute the *consequences* of a given event occuring at a world. They are best described in **Prisoners** for an event e, if $\mathsf{pre}(\mathsf{e}) = \phi$ and $\mathsf{post}(\mathsf{e}, p_1) = \psi_1$, we say "if ϕ holds, then after e occurs $p_1 := \psi_1$".

We should give some insight on the definition of W' and R'. We see that in our updated set of worlds, the worlds become pairs of elements in W and events from E. This will telescope infinitely; for instance worlds in $\mathcal{M} \times \mathcal{E} \times \ldots \times \mathcal{E}$ will be of the form $(\ldots (w, e) \ldots, e)$, where $w \in W$ and $e \in E$. Hence the worlds are traces of events that happened. Similarly, R' relates pairs of worlds from W', which are again pairs of worlds and events. Here, the worlds remain indistinguishable only if they were indistinguishable before and both events are indistinguishable to i.

[7] An example of this is given in Figure 2.3

We see that it is the postcondition function that allows for *factual change*; that is, the update of what is true at a state given the occurence of some event. We can see how this gives us a much more expressive model than simple PAL; with carefully selected pre and post functions, we can express a very wide range of communications and events.

As before, we now show some example event models, and show the effects of updating an epistemic model with an event model.

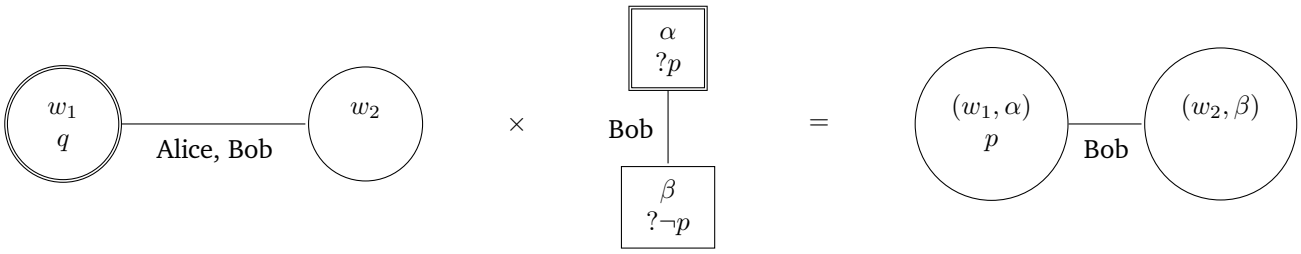


Figure 2.2

In this example from the literature (**MalvinThesis**), consider two agents, again Alice and Bob. They are waiting in the same room to receive a letter about Alice's entrance onto a PhD program. Suddenly the postman arrives, and Bob sees Alice pick up and open a letter with the University's logo on the front. Alice reads the letter and learns that she gets the position, but she does not tell Bob the result. Bob sees that Alice now knows whether she got it, but he does not know if she did get it or not.

Here, p stands for the proposition "Alice gets the position". This initial situation is represented by model $\mathcal{M}$. In this initial situation, both Alice and Bob cannot distinguish between the worlds where p is true and where p is not true. Then the event model $\mathcal{E}$ represents the event of Alice reading the letter from the unviersity. α represents the event of Alice getting the position and β that she doesn't - hence the preconditions are p and $\neg p$ respectively; that is, $\mathsf{pre}(\alpha) = p$ and $\mathsf{pre}(\beta) = \neg p$. Bob cannot distinguish between either of these events, but Alice can, as she reads the letter. Then finally the epistemic model $\mathcal{M} \times \mathcal{E}$ represents the situation in $\mathcal{M}$ after the event $\mathcal{E}$ has occurred. We can see the epistemic model $\mathcal{M}$, event model $\mathcal{E}$ and updated model $\mathcal{M} \times \mathcal{E}$ in Figure 2.2.

As a second example, consider that Alice and Bob are in a room. There's a coin on the table with the heads face up - we set this to be represented by the proposition p. Alice and Bob can both see the coin face up. This situation is modelled by epistemic model $\mathcal{M}$. Then Bob picks up the coin and flips it in the air. Bob sees the result of this coin flip, but does not show Alice. We can see the event model and epistemic model, and their update in Figure 2.3.

Our epistemic model has just one state - that is, the state where the coin is on the table face up. Both Alice and Bob can see the coin, and as such they both know that p holds. Then our event model has two events; α, in which the coin lands face-up, and β, in which the coin lands face-down. Bob can distinguish between these two events as he can see the result of the coin toss, but Alice cannot, as she does not see it. In the former p is set to true, and in the latter p is set to false. Hence in the epistemic model $(\mathcal{M} \times \mathcal{E})$ we have two states; that in which the coin landed face-up and that in which the coin landed face-down.

2.2 The Gossip Problem

Gossip is a procedure for spreading secrets around a group of agents, where the agents are commonly displayed as nodes in a graph and the ability of one agent to contact another displayed as an edge between two nodes. Gossip was first put forward in Tijdeman 1971, where the network is a complete graph; all agents can contact one another. One question here was to find how many calls are needed for every agent to learn the secret of every other agent. We will henceforth describe an agent knowing the secret of every other agent as this agent being an *expert*. It was quickly proved (**TelephoneDisease GandT**) that this number, for a network where the number of agents n is greater than 4, is $2n - 4$.

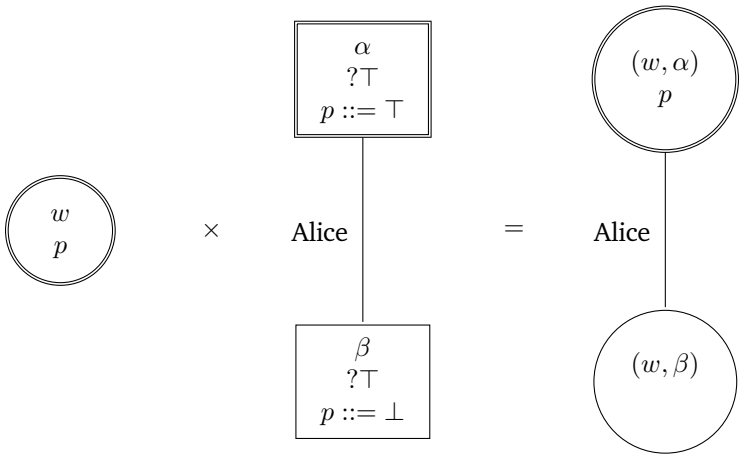


Figure 2.3: *An event model and epistemic model for the coin toss example, and the updated event model.*

2.2.1 Dynamic Gossip

Dynamic Gossip is a variant of the classical gossip problem, in which we start off with an incomplete graph. This represents the fact that the agents only have the phone numbers of some of the other agents. In this case, when two agents talk on the phone, they also exchange all of the phone numbers that they know, as well as the secrets that they know.

2.2.2 Protocols

The gossip problem, as we have mentioned so far, can rely on some central scheduler to tell the agents what call to make, and lets them know when to stop. However in distributed computing, methods that do not require this central authority are desirable. In such a situation, the agents need to have some form of rules to follow to decide how to behave and who to call next. This is the motivation for a *gossip protocol*, which are short conditions that must be fulfilled in order for an agent to make a specific call. These protocols were first proposed in **EPfDG KnowledgeandGossip** and some exemplar ones are:

- ANY - If x knows the phone number of y, x may call y.
- LNS - If x knows the phone number of y and x does not know the secret of y, then x may call y.
- PIG - If x knows the phone number of y and x considers it possible that either x or y will learn a new secret as a result of the call, then x may call y.

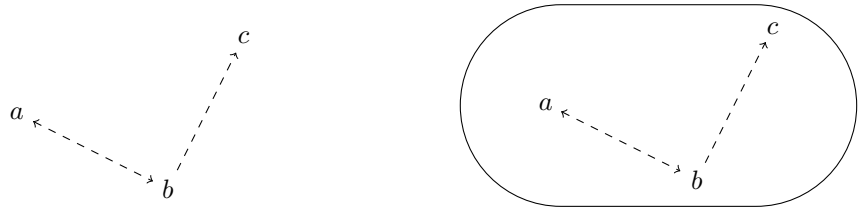
In this thesis, we do not study the distributed gossip problem; rather, a version with a central authority that surveys the network topology and then decides which agent is allowed to make a call, and also who they will call. However, these protocols do still have use for us. ANY allows for infinite call sequences - for example, one where agent a just repeatedly calls b, whereas all of the call sequences induced by LNS are of finite length. In the long run this does not really matter; both ANY and LNS have runtime expected execution length in $O(n \log n)$ (van Ditmarsch et al. 2018), yet in our implementation tests performed with LNS took considerably less time than ANY. For this pragmatic reason LNS is often used in this thesis. PIG is an example of an *epistemic* protocol; these are protocols that concern the knowledge of the agents. These are much more difficult to reason about in our implementation, for reasons that we come onto later.

It should also be noted that there are certain classes of graphs for which LNS cannot induce a successful call sequence, yet ANY can (van Ditmarsch et al. 2018). However, this thesis does not investigate this topic, and as such this will no longer be mentioned.

2.2.3 Formalisation

We now go on to formalise some of the ideas mentioned so far in this section. The definitions of gossip graphs are classic and can be found in all of the related literature (e.g. van Ditmarsch et al. 2018, **MalvinThesis**).

We formally denote a gossip graph $\mathcal{G}$ with a triple (A, N, S). A is a finite set of agents in the graph; $N \subseteq A \times A$ is a set of ordered pairs of agents such that $(u, v) \in N$ iff u knows the phone number of v. $S \subseteq A \times A$ is a set of ordered pairs of agents such that $(u, v) \in S$ iff u knows the *secret* of v. We denote the fact that $(u, v) \in N$ with the proposition Nuv; respectively we denote $(u, v) \in S$ with the proposition Suv. We also have that for all gossip graphs, for any agent a, $(a, a) \in N$ and $(a, a) \in S$, expressing that any agent always knows their own phone number and secret.



(a) *An exemplar gossip graph.* **(b)** *A Kripke model $\mathcal{G}$ for a Gossip State*

Figure 2.4

We can use the notation above to formally describe a gossip graph. For instance, the gossip graph that we can see in Figure 2.4a can be expressed as the triple $\mathcal{G} = (\{a, b, c\}, \{(a, a), (a, b), (b, a), (b, b), (b, c), (c, c)\}, \{(a, a), (b, b), (c, c)\})$. However given that we have the property described above, we can remove the facts of agents knowing their own secret and phone number and represent this as the more terse $\mathcal{G} = (\{a, b, c\}, \{(a, b), (b, a), (b, c)\}, \{\})$.

One might ask how gossip fits into the picture we drew above of Kripke models and event models. We give a way to generically define Kripke models and event models for an arbitrary gossip graph $\mathcal{G} = (A, N, S)$. A Kripke model for a graph $\mathcal{G}$ uses the set of propositions $\{N_{ij} \mid i \in A, j \in A\} \cup \{S_{ij} \mid i \in A, j \in A\}$. We will hereafter refer to this set of propositions as $\Lambda_{\mathcal{G}}$. The Kripke model for a graph $\mathcal{G}$ has just one world, q, which is the world that models the graph. We define $\mathcal{M}_{\mathcal{G}} = (W, R, V)$ where $W = \{q\}$, $R_i = \{(q, q)\}$ for every agent $i \in A$, and $V(q) = \{N_{ij} \mid (i, j) \in N\} \cup \{S_{ij} \mid (i, j) \in S\}$. An example of this is in Figure 2.4; we see that the Kripke model for the graph is not particularly complicated.

An event model can also be produced in a similar way. We make an event model $\mathcal{E} = (E, P, \mathsf{pre}, \mathsf{post})$, where

- $E = \{ij \mid i \in A, j \in A, i \neq j\}$. Our set of events is the calls between agents, however an agent cannot call themself. A call is represented by a pair of agents; ij represents a call from i to j.
- $P_i = \{(jk, j'k') \mid j, k, j', k' \in A, i \notin \{j, k, j', k'\}\} \cup \{(ij, ij) \mid j \in A\} \cup \{(ji, ji) \mid j \in A\}$, for every $i \in A$. We can read this as saying agent i cannot distinguish between a call that it is not a part of; hence the former part of the definition. The later two sets say that agent i does not confuse a call that it's a part of with any other call. This means that a cannot distinguish between the world reached by the call bc happening and the world reached by the call cd happening; however a can tell the difference between worlds reached by calls it is a member of.
- $\mathsf{pre}(ij) = N_{ij}$. By this we mean that the precondition for a call from agent i to j is that i knows the phone number of j.
  It is here that we can add in protocols from Section 2.2.2; for instance, in an event model where we follow the protocol LNS, we add the condition $\neg S_{ij}$ to the precondition; hence $\mathsf{pre}_{\mathsf{LNS}}(ij) = N_{ij} \wedge \neg S_{ij}$.
- $\mathsf{post}(ij, N_{nm}) = \begin{cases} N_{im} \vee N_{jm} & \text{if } n = i \text{ or } n = j \\ N_{nm} & \text{otherwise} \end{cases}$
  $\mathsf{post}(ij, S_{nm}) = \begin{cases} S_{im} \vee S_{jm} & \text{if } n = i \text{ or } n = j \\ S_{nm} & \text{otherwise} \end{cases}$
  This states that if a call occurs, then for an agent n to know the number of an agent m afterwards, either n was a part of the call and n either knew the number of m before or learned it from speaking to the other agent in the call, or n was not a member of the call and knew the number of m anyway. The same goes for secrets.

In Figure 2.5, we give the event model for Figure 2.4a. See that this matches up with our earlier description; every agent cannot distinguish between the calls they are not involved with.

Recall the discussion in Section 1, where we say that we choose to use the synchronous variant of the gossip problem. In this we assume the existence of a central clock, which synchronises the calls between agents. The

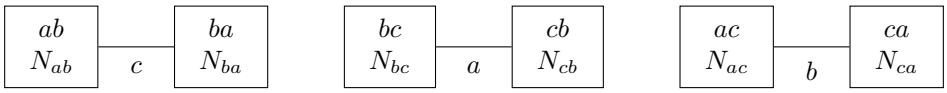


Figure 2.5: *An Event model $\mathcal{E}_\mathcal{G}$ for the calls associated with Figure 2.4a.*

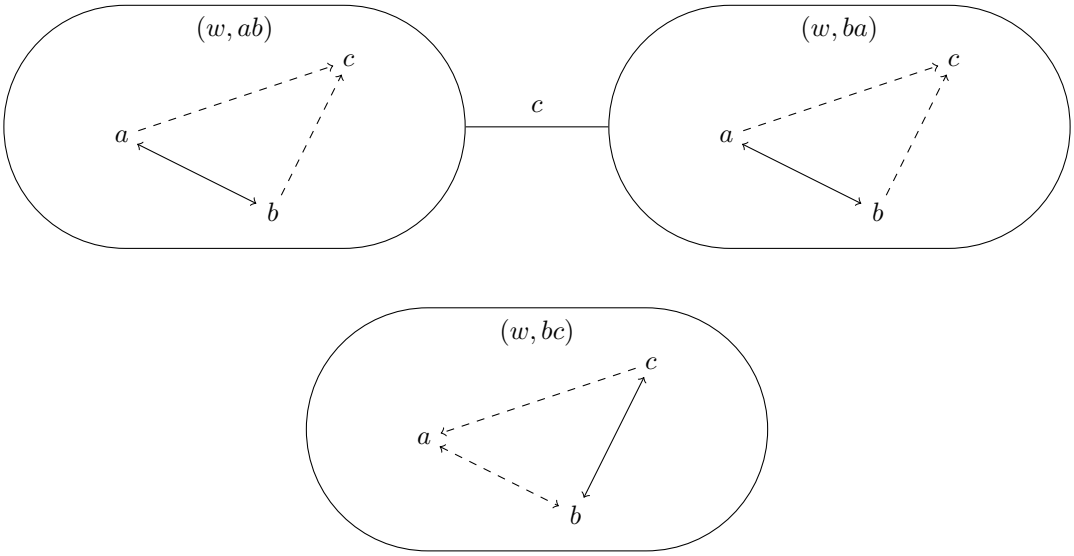


Figure 2.6: *The Kripke model $\mathcal{G} \times \mathcal{E}_\mathcal{G}$.*

implications of this are that when a call occurs, all of the other agents are aware that a call occurs; but only the agents involved in the call are aware of who is involved in the call. Hence when a call occurs between agents a and b, all agents know that *some* call occued. a and b know exactly which call occured; however the other agents have no idea which call happened. They *consider it possible* that any call between any pair of agents (excluding themselves) could have happened.

In Figure 2.6 we can see the result of updating the Kripke model $\mathcal{G}$ with the event model $\mathcal{E}_\mathcal{G}$. We see that the calls cb, ac, ca do not have corresponding worlds; their preconditions were not satisfied by any of the worlds in $\mathcal{G}$. Also note that the worlds (w, ab) and (w, ba) are indistinguishable to c; this is because c cannot distinguish between calls ab and ba as c is not included in it. Also note that the two worlds (w, ab) and (w, ba) have the exact same graph representing them[8]; in the future we will not make this distinction.

In the implementation a world is represented *just by the propositions that are true at it*. This makes use of the bijection from $\mathcal{P}(\Lambda_\mathcal{G})$ to the set of gossip graphs. By this we mean that every gossip graph (e.g. Figure 2.4a) is represented by exactly one set of propositions $S \subseteq \Lambda_\mathcal{G}$, and vice versa. This gives us the useful property of being able to encode these gossip graphs with sets of propositions in the implementation, which makes gossip graphs much easier to deal with and reason about.

2.3 Planning

Automated planning is the process of computing which set of events must occur to take a system from some given initial state to some successful state. Such a system is defined as a triple $\Sigma = (S, A, \gamma)$, where:

- S is some set of states;
- A is some set of actions;
- $\gamma : S \times A \hookrightarrow S$ is a state-transition function. It is partial; for $s \in S, a \in A$, either $\gamma(s, a) \in S$ or it is undefined.

Then an instance of the planning problem is a triple (Σ, s_i, S_g), where;

[8] We revisit this idea of gossip graphs with three agents being "boring" again later. This "boring"-ness comes from the fact that calls ab and ba are essentially the same thing.

- Σ is a planning system;
- $s_i \in S$ is some initial state;
- $S_g \subseteq S$ is some set of goal states.

A *solution* to the planning problem is some ordered set of actions $a_0, a_1, \ldots, a_n$ such that

$$\gamma(\gamma(\ldots\gamma(s_0, a_0)\ldots, a_{n-1}), a_n) \in S_g.$$

We call a planning problem *multi-agent* if the number of agents in the system is greater than 1, and *single-agent* if the number of agents is equal to 1.

2.3.1 Epistemic Planning

The dynamic epistemic logic community has recently been investigating a special case of planning, namely *epistemic* planning (Bolander and Birkegaard 2011, Aucher and Bolander 2013). This is planning where we may have some initial epistemic state (e.g. agent a knows ϕ), some actions that update these epistemic states, and some accepting epistemic state (e.g. agent b knows ψ). It is not hard to see how this can be applied to the gossip problem.

We give a formal definition of the epistemic planning problem that slightly differs from convention, however this definition and that from the literature are equivalent. The epistemic planning problem is as follows; given a pointed epistemic model $(\mathcal{M}, w)$, an event model $\mathcal{E}$, and some goal formula ϕ, find some string of events $e_1, e_2, \ldots, e_n$ such that $(\mathcal{M}, w) \otimes (\mathcal{E}, e_1) \otimes \ldots \otimes (\mathcal{E}, e_n) \models \phi$. The *propositional epistemic planning problem* is the restriction of the epistemic planning problem to event models with pre- and post-condition functions whose codomains are in $\mathcal{L}_P(\Lambda)$ that is, they do not contain the modal operator K. These event models are what we call propositional event models.

In Bolander and Birkegaard 2011 it is shown that the multi-agent epistemic planning problem is undecidable for a general epistemic models and event models. However, placing certain restrictions on the system used yields decidability (Yu, Wen, and Liu 2013, Aucher, Maubert, and Pinchinat 2014). One particularly important result is that the *propositional epistemic planning problem is decidable*; it is this important result that this thesis relies on.

In this thesis, we focus on solving the epistemic planning problem by means of the automata constructions described in Aucher, Maubert, and Pinchinat 2014. These methods, and the resulting piece of software associated with this thesis, are unique in that they will work for any epistemic model and (propositional) event model. This unfortunately means that we cannot plan for situations like muddy children or drinking logicians (**DEMO-S5**) as they require epistemic pre- or post-conditions.

The question arises of the significance of the gossip problem to the epistemic planning problem. There are several reasons for the use of the gossip problem as the lens through which we study epistemic planning;

- It is a widely-studied problem with lots of applications already listed in this thesis. The consequences of this are that we have a wealth of literature to be used. We give examples of this in Section 2.4. Its applicability sets it apart from many other epistemic problems that are studied; these mainly are sorts of logic puzzles. Examples of these can be seen in **DEMO-S5**
- It encapsulates very easily all of the aspects of epistemic planning that we're interested in - namely the indistinguishability of events from one another, leading to epistemic possibility from the point of view of agents - whilst still being a simple enough problem to understand intuitively.

2.4 Existing Tools

We now give an overview of existing software in the realm of model checking or planning for epistemic systems.

Recall that model checking for an epistemic model is the question: given an epistemic model $\mathcal{M}$ and a formula $\phi \in \mathcal{L}(\Lambda)$ for some set of propositions Λ, decide if $\mathcal{M} \models \phi$. Dually, planning for an epistemic model is the question: given an epistemic model $\mathcal{M}$, an event model $\mathcal{E}$ and a formula $\phi \in \mathcal{L}(\Lambda)$ for some set of propositions Λ, compute a sequence of events $e_1, e_2, \ldots, e_n$, where each event in the sequence is an event $\mathcal{E}$, whose occurence takes the model $\mathcal{M}$ to a state in which the formula ϕ is satisfied.

2.4.1 DEMO-S5

DEMO-S5 (**DEMO-S5**) is a model checker for epistemic models, limited to S5 models. It is also limited in that it only supports events that are either public announcements (e.g. an agent telling every other agent some statement) or publicly observable factual change (e.g. the flip of a coin, where the result is seen by every agent).

This limitation removes support for indistinguishable events; the occurence of each event is seen by each agent and is perceived in the same way. This means that the occurence of an event never increases the amount of worlds considered possible by an agent; conversely, the number of worlds considered possible only stays the same or decreases when some event occurs. This can be seen in the "Drinking Logicians" section of **DEMO-S5**

It performs the task of model checking by taking an epistemic model and a formula representing either the announcement or the fact that changed, and updating the model with the event represented by the formula. The definition of the update used is very similar to the update of an epistemic model with an event model as defined in Section 2.1.2. The given events are applied, and once they have all been applied we check if we're in a successful state or not.

Unfortunately we cannot use this to model check a gossip problem, as events in the gossip problem are neither of these; they are private one-to-one communications. Furthermore, the software is incapable of planning; simply model checking. Despite this, the software was incredibly useful in our project in aiding the understanding of epistemic and event models. Indeed, the formalisation of epistemic and event models demonstrated here was used in the code associated with this thesis.

2.4.2 Gossip

GithubGossip is another model checker, however for strictly the gossip problem. This program has a significant advantage over the other two pieces of software; namely that it is capable of planning. It does this by generating all of the possible sequences of calls starting from the graph and then checking which of these are successful (i.e., after execution of all of the calls the graph is in some successful state). It also has capabilities for studying protocols deeply, however these are not relevant to our thesis.

Although the methods used within this software are very different to the ones used in our implementation, this system proved to be very useful for testing purposes, due to its simplicity of use. The results returned by our software during testing were verified using this system. Further elaboration on this will be given in the evaluation section of this thesis.

One limitation of this system is that is may only handle gossip states, and not generic epistemic models. Also limiting is that it enumerates all possible call sequences from the initial state; we are only interested in successful ones. This means that we have the possibility of lots of wasted computation if we need to validate lots of unsuccessful paths before we get to a successful one.

However, the biggest problem is its handling of protocols like ANY, which allow for infinite call sequences. The way that the call sequences are generated is depth-first, and the system will only check the sequences generated once they have all been produced. This means that when planning with the ANY protocol it eternally generates these call sequences, and never reaches the point at which it checks the successful call sequences. This is clearly a big limitation and in our system we aim to avoid this.

2.4.3 SMCDEL

SMCDEL is the most sophisticated of the three pieces of software. A technical report is given in **SMCDEL** whilst a lot of the underlying theory is in **MalvinThesis** The main difference is that it uses *symbolic model checking*, a method put forward in **SymbolicModelChecking** This gives a much more efficient implementation **why**?, as can be seen in the Chapter 4 of **MalvinThesis**

This software is capable of efficiently model checking all classes of epistemic models. It could be capable of planning, however there would be significant work to do in order to make this happen. Furthermore, due to a reasonably unpleasant interface we choose not to use this software for benchmarking and instead use the simpler **GithubGossip**

2.5 Uniform Strategies

Recall that in the introduction we outlined that one of our contributions was a tool to perform epistemic planning on epistemic models and propositional event models. So far in this chapter we have introduced the language we use to reason about knowledge and structures with which we can model knowledge and its dynamics. We then discussed the dynamic gossip problem, and how it can be modelled with the techniques introduced in Section 2.1, and we then went on to introduce the planning problem.

This background constitutes nearly everything we need to know to understand the problem that we are solving and the process we use to do so. Section 2.5 introduces *uniform strategies*, to finish off the background chapter. Uniform strategies are the most abstract and difficult part of this background chapter, however they end up also being the most useful tool in our toolbox.

Recall that in order to verify the truth or falsity of a formula of the form $K_i\phi$, we need to visit all the worlds that agent i cannot distinguish from their current world and see whether or not ϕ holds there. Doing this quickly and easily is of great importance to our system. However, we see that in Section 3.1, this is not extremely simple to do; whilst it is straightforward to enumerate the set of worlds that can be reached through application of an event model to a Kripke model, it is not so straightforward to compute the worlds that an agent may consider possible from some world. We would need to consider the call sequence we made and compute the call sequences that are indistinguishable to the current one. This is a slow, costly process. What we want is to be able to evaluate epistemic formulae *positionally*; that is, by only looking at the world in question.

In **UniformStrategies** - the reference for uniform strategies - a process is detailed that helps us towards this goal of positional evaluation of a formula of form $K_i\phi$. This process takes as input an automaton $\mathcal{A}$, whose successful states are those in which the formula ϕ is true, and a transducer $\mathcal{T}_i$[9], which relates pairs of events that are indistinguishable to agent i. As output, we are returned an automata $\widehat{\mathcal{A}}$ whose transitions follow those in $\mathcal{A}$[10], but whose states contain in them all of the worlds indistinguishable to agent i. This lets us positionally verify formulae of the shape $K_i\phi$.

We first introduce this process - named the *powerset construction* - in the context of game arenas. These are automata-like structures, arising from game theory, that model the playing of a two-player game. They are used in **UniformStrategies** to investigate uniformity properties of sequences of plays through graphs. In this, a modality R is introduced, meaning "for all related plays". We can very easily interpret this modality as our K modality.

We give an overview of game arenas, an introduction to transducers and a definition of the powerset construction. The algorithm that we later give for epistemic planning relies heavily on this process, however uses a simplified version. In order to illustrate that our algorithm is a special case of this process, we introduce the powerset construction in its natural game theory setting.

Important to note is that this section is *not* essential to understanding the rest of this thesis. The text above details the motivation for this technique; however the reader will be able to continue onto the next chapter without reading the rest of this section and still understand the process very easily. The reader may indeed get more out of this section returning to it *after* reading the algorithm detailed in the next chapter. The *only* part that the reader should read is Section 2.5.2, which introduces our notation used to reason about finite state transducers.

2.5.1 Game Arenas

A game with *imperfect information* is a game in which the players have some uncertainty concerning the current configuration of the game (for example Poker; a player does not know which cards are in the other players' hands). A player in such a game cannot plan to play differently in a situation she cannot distinguish; hence we have to define her strategy *uniformly* across situations she cannot distinguish.

We consider two-player turn-based games played on some graph, where the vertices are labelled with propositions. These propositions hold some useful information about the world; it could be what people know about it, or some fact about the environment the player is in and so on. This set of propositions will be referred to as AP.

Then a game arena is a structure $\mathcal{G} = (V, E, v_I, l)$ where $V = V_1 \uplus V_2$ [11] is a set of positions split between those of Player 1 (V_1) and those of Player 2 (V_2). $E \subseteq (V_1 \times V_2) \cup (V_2 \times V_1)$ is the set of edges, either from a

[9] We give a recap of transducers in Section 2.5.2.

[10] That is, a word accepted by $\mathcal{A}$ will also be accepted by $\widehat{\mathcal{A}}$. The difference between the two is that the states in $\widehat{\mathcal{A}}$ contain much more information than those in $\mathcal{A}$.

[11] We use $\uplus$ here to indicate that the two sets are *disjoint*.

position in V_1 to V_2 or vice versa. $v_I \in V$ is the initial position and $l : V \to \mathcal{P}(AP)$ is a valuation function, which maps each position to the finite set of propositions that are true at it.

It may be easy to see the relationship between game arenas and finite state automata. We can see V as a set of states in an automata. E encodes the transitions between states, and as such we can see it as representing the transition function δ. v_I is the initial state of the automata, and l lets us valuate the automata states.

2.5.2 Transducers

We give a quick recap of transducers, as they are used heavily in the next two sections. A transducer is like a nondeterministic finite automaton with two tapes; an *input* tape and an *output* tape. The transducer reads an input finite word on its input and writes a finite word on its output tape.

A transducer is a 6-tuple $T = (Q, \Sigma, \Gamma, I, F, \Delta)$, where Q is a finite set of states, Σ is the input alphabet, Γ is the output alphabet, $I \subseteq Q$ is a finite set of initial states, $F \subseteq Q$ is a finite set of final states, and $\Delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times (\Gamma \cup \{\epsilon\}) \times Q$ is the transition function.

$(q, a, b, q') \in \Delta$ means that the transducer can move from state q to state q' by reading a from the input tape and writing b to the output tape. We will use this notation throughout, however this is equivalent to saying that $\Delta(q, a) = (b, q')$. Note that transducers are, in this thesis, nondeterministic; hence if $\Delta = \{(q, e, f, p), (q, e, g, r)\}$ then $\Delta(q, e) = \{(f, p), (g, r)\}$. Hence a transducer defines a binary relation $R \subseteq \Sigma \times \Gamma$, where the pairs in the relation are pairs (a, b) such that $(q, a, b, q') \in \Delta$ for some $q, q' \in Q$. Relations recognised by a transducer are called *regular* or *rational* relations.

Then we can define the extended transition relation Δ^* as the smallest relation satisfying:

- for all $q \in Q$, $(q, \epsilon, \epsilon, q) \in \Delta^*$
- if $(q, w, w', q') \in \Delta^*$ and $(q', a, b, q'') \in \Delta$, then $(q, w.a, w'.b, q'') \in \Delta^*$

We can see Δ^* as the transitive closure of Δ. We abbreviate $(q, w, w', q') \in \Delta^*$ to $q \rightsquigarrow_{w|w'} q'$ in the rest of this section. The extended transition relation lets us relate *words* of characters in Σ with words of characters in Γ.

Finally, we say that the relation recognised by some transducer T is

$$[T] ::= \left\{(w, w') \mid w \in \Sigma^*, w' \in \Gamma^*, \exists q_F \in F, \exists q_i \in I, q_i \rightsquigarrow_{w|w'} q_F\right\}$$

2.5.3 The Powerset Arena

In games with imperfect information, the *information set* of a player after a certain play is the set of positions she may possibly be in, consistent with what she has observed[12]. We can define a similar notion in our setting, and we show that this is sufficient to build a powerset arena in which formulas of the form $K_a\phi$, where ϕ is propositional, can be evaluated positionally.

For an arena $\mathcal{G} = (V, E, v_I, l)$ and a transducer $T = (Q, \Sigma, \Gamma, I, F, \Delta)$ such that $[T] \subseteq Plays_* \times Plays_*$[13], we construct this automata $\widehat{\mathcal{G}}$ in which we can evaluate formulas of the form $K_i\phi$ *positionally*. We do this by bringing extra information into the states of $\widehat{\mathcal{G}}$, including the set of other possible states we could be in.

Our new information set after a move occurs cannot be computed knowing only the previous information set and the new position; we need to simulate the nondeterministic execution of T, taking as input the sequence of positions played and writing as output the related plays. Precisely, we need two things; the set of states the transducer may be in after reading the sequence of positions played so far, and for each of these states the set of possible last positions written on the output tape. We need only remember the last letter and not the whole tape because the information set we aim to compute is just the set of the last positions of related plays.

Then our positions are of the form $(v, S, Last)$, where $v \in V$, $S \subseteq Q$, where Q is the set of states in the transducer and S is the set of possible current states of T, and $Last : S \to \mathcal{P}(V)$[14] associates to a state $q \in S$ the set of the possible last positions on the output tape fo T if the current state is q. The transitions in this arena follow those in $\mathcal{G}$, except we now maintain the additional information about the configuration of the transducer.

[12] For instance in a game of poker with one deck of cards, a player may have no idea what the other players have in their hands except that they know they do not have the cards the player has in their own hands, nor the cards face-up on the table.

[13] $Plays_*$ is the set of *finite* plays of the game in the game arena. A play is a string of events, moving from items in V_1 to V_2 to V_1 and so on.

[14] It may be more helpful to think of this as something a bit more tangible than a function, e.g. a list of tuples $(S, [P])$

To construct the initial position $\widehat{v_I} = (v_I, S_I, Last_I) \in \widehat{\mathcal{G}}$, we need to simulate the execution of T starting from its initial state and reading v_I. To do this, we introduce an artificial position $\widehat{v_{-1}} = (v_{-1}, S_{-1}, Last_{-1})$, where $v_{-1} \notin V$ is some fresh position, $S_{-1} = \{q_i\}$ where q_i is the initial state of the transducer because before starting the transducer is in its initial state, and $Last_{-1}(q_i) = \emptyset$ because we have nothing written on the output tape.

We now come onto defining $\widehat{\mathcal{G}}$. Let $\mathcal{G} = (V, E, v_I, l)$ be an arena and $T = (Q, V, q_i, Q_F, \Delta)$ be an FST. Then we define the arena $\widehat{\mathcal{G}} = (\widehat{V}, \widehat{E}, \widehat{v_I}, \widehat{l})$ as:

- $\widehat{V} = V \times \mathcal{P}(Q) \times (Q \to \mathcal{P}(V))$
- $(u, S, Last)\widehat{\to}(v, S', Last')$ if
  - $u = v_{-1}$ and $v = v_I$, or $u \to v$,
  - $S' = \{q' \mid \exists q \in S, \exists \lambda' \in V^*, q \rightsquigarrow_{v|\lambda'} q'\}$ and
  - $Last'(q') = \{v' \mid \exists q \in S, \exists \lambda' \in V^*, q \rightsquigarrow_{v|\lambda' \circ v'} q',$ or $q \rightsquigarrow_{v|\epsilon} q'$ and $v' \in Last(q)\}$
- $\widehat{v_I}$ is the only $\widehat{v} \in \widehat{V}$ such that $\widehat{v_{-1}}\widehat{\to}\widehat{v}$.
- $\widehat{l}(\widehat{v}) = l(v)$ if $\widehat{v} = (v, S, Last)$.

The definition of transitons is quite complicated, and as such we give a high-level explanation of it. Regarding the definition of the transitions, the first point means that our transitions in $\widehat{E}$ follow those in E, except for the transition leaving $\widehat{v_{-1}}$, which we use to define $\widehat{v_I}$.

The second point expresses that when we move from u to v in $\mathcal{G}$, we give v as input to the transducer. Then the set of states that we can be in, that is, S', is the set of states that can be reached from some previous possible state in $q \in S$ by reading v on the input tape and outputting some sequence λ'.

Finally, the third point expresses that if some position v' is at the end of the output tape after the transducer reads v and reaches q, then this is either because while reading v the last letter it wrote is v', or it wrote nothing and v' was already at the end of the output tape before reading v.

Finally, we say that $\widehat{v_I}$ is the only successor of $\widehat{v}_{-1}$, and that the valuation of a position in the powerset arena is the valuation of the underlying position in the original arena.

2.5.4 Lifting Transducers

We can keep on repeating this process, creating a power-powerset arena $\widehat{\widehat{G}}$. However we first need to lift our transducer T where $[T] \subseteq Plays_* \times Plays_*$ to a transducer $\widehat{T}$ where $[\widehat{T}] \subseteq \widehat{Plays_*} \times \widehat{Plays_*}$.

We write $T\downarrow$ for the transducer that computers the bijective function f that maps a play $\widehat{p} \in \widehat{Plays_*}$ to the underlying play $p \in Plays_*$, and $T\uparrow$ for the deterministic transducer that computes f^{-1}. Both can be easily constructed from our powerset arena $\widehat{G}$.

Then the lift of a transducer T is $\widehat{T} = T\downarrow \circ T \circ T\uparrow$, where $\circ$ is the transducer composition operator. This definition communicates that our lifted transducer simply computes the corresponding *underlying* play, performs the operation of the underlying transducer, and then computes the corresponding lifted play.

Once we have the transducer $\widehat{T}$, we can repeat the process in Section 2.5.3 with $\widehat{G}$ to get $\widehat{\widehat{G}}$, and so on ad infinitum. $\widehat{\widehat{G}}$ would allow us to positionally check formulae of the form $K_i K_j \phi$.

Chapter 3

Algorithm

In this section, we formalise the algorithm that we use to solve the propostional epistemic planning problem. This is a major contribution of this thesis; there is no existing, clear implementation of this in the literature.

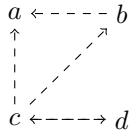


Figure 3.1

We first give a rough explanation of the process, before getting into the details of the process later in this chapter. As part of this, we revisit the example given in Section 1, to shed some light on how the program produces the given call sequence. The gossip graph used is given in Figure 3.1, and the successful formula in this case is $\phi = \forall_{i \in Ag} E_i$, where Ag is the set of agents in the gossip graph; in the case of Figure 3.1 this is the set $\{a, b, c, d\}$.

Step one; we receive as input an epistemic model $\mathcal{M}$, a propositional event model $\mathcal{E}$ and some formula $\phi \in \mathcal{L}(\Lambda)$, for some set of propositions Λ. We first construct an automata called $\mathcal{ME}^*$, by simulating the repeated application of $\mathcal{E}$ onto $\mathcal{M}$. $\mathcal{ME}^*$ is a representation of all worlds that we can access from the application of any sequence of events in $\mathcal{E}$. This lets us easily traverse the set of possible states we can reach from our initial model.

In the case of planning for the Dynamic Gossip problem, recall that we can generate an event model $\mathcal{E}$ automatically given an initial gossip graph - like the one in Figure 3.1. The gossip graph forms the model $\mathcal{M}$; hence given just the initial gossip graph we can produce the automata $\mathcal{ME}^*$.

Here, recall the definition of a planning problem from Section 2.3. We require a planning system with a set S of states, a set A of actions and a partial transition function $\gamma : S \times A \hookrightarrow S$. $\mathcal{ME}^*$ provides us with all of these; our set of states S is the state space Q, the set of actions A is the alphabet Σ and the transition function γ is exactly the transition function δ of $\mathcal{ME}^*$. Furthermore, we can define set of goal states to be the set of states that model the winning condition, and let the initial state be the initial state of $\mathcal{ME}^*$. $\mathcal{ME}^*$ when $\mathcal{M}$ is Figure 3.1 and $\mathcal{E}$ is the corresponding event model is modelled in Figure 3.2.

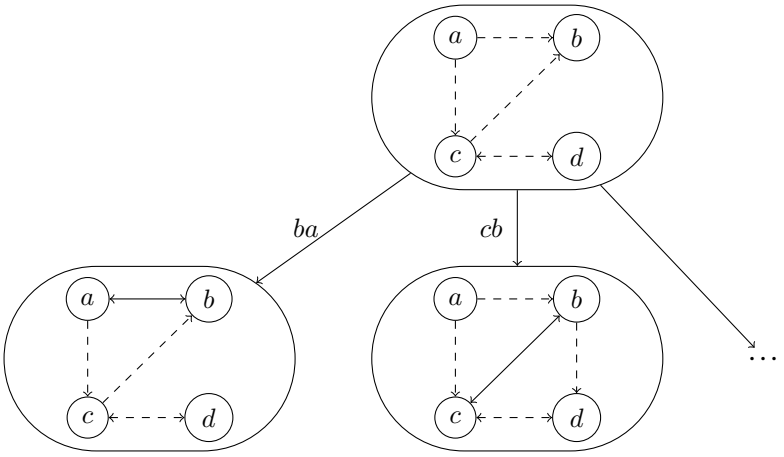


Figure 3.2: *The gossip graph in Figure 3.1, updated with $\mathcal{E}$ once. We omit some transitions for simplicity.*

Step two; we now consider the success formula ϕ. If it does not contain a knowledge modality - that is,

$\phi \in \mathcal{L}_P(\Lambda)$ - then we proceed to step three. However if it does (like in our example), our process changes. Recall that to find the truth value of a formula $K_i\phi$ evaluated at some state we need to visit all of the states indistinguishable from the current state to agent i and check if ϕ holds at *all* these states. In $\mathcal{ME}^*$ we have no way of knowing which states are indistinguishable from the state in question; hence it is very difficult to check the truth-value of such formulae.

To this end, we perform the process detailed in Section 3.2. In this process we build a transducer $\mathcal{T}_i$ which relates pairs of events that are indistinguishable to one another. We then step through $\mathcal{ME}^*$ again, building another automata $\widehat{\mathcal{ME}^*}$ where the states are instead *sets of states* in $\mathcal{ME}^*$. This is done by following the transitions in $\mathcal{ME}^*$ and, whenever we make a transition through the occurence of an event, computing the set of events that are indistinguishable to agent i and applying this to all of the states in the set of states that we could be in. This means that at each state in this powerset automata, we have the set of states that i considers possible, and hence we can quickly check formulae of form $K_i\phi$. This yields *positional evaluation* of epistemic formulae.

We can see the process of performing this procedure on the automata in Figure 3.2 in Figure 3.3. We can see that the calls cb, cd and dc all take us to the same state. This is because agent a cannot distinguish between these three calls; hence the state the occurence of any of these calls takes us to is the same. The state itself contains two states in $\mathcal{ME}^*$. The former is resulting state from applying cb to the initial state; the latter is the resulting state from applying cd or dc to the initial state[1]. Calls ba and ca take us to unique states. This is because agent i can distinguish between calls that they are a part of, and as such is completely sure of which state they are in after the call occurs.

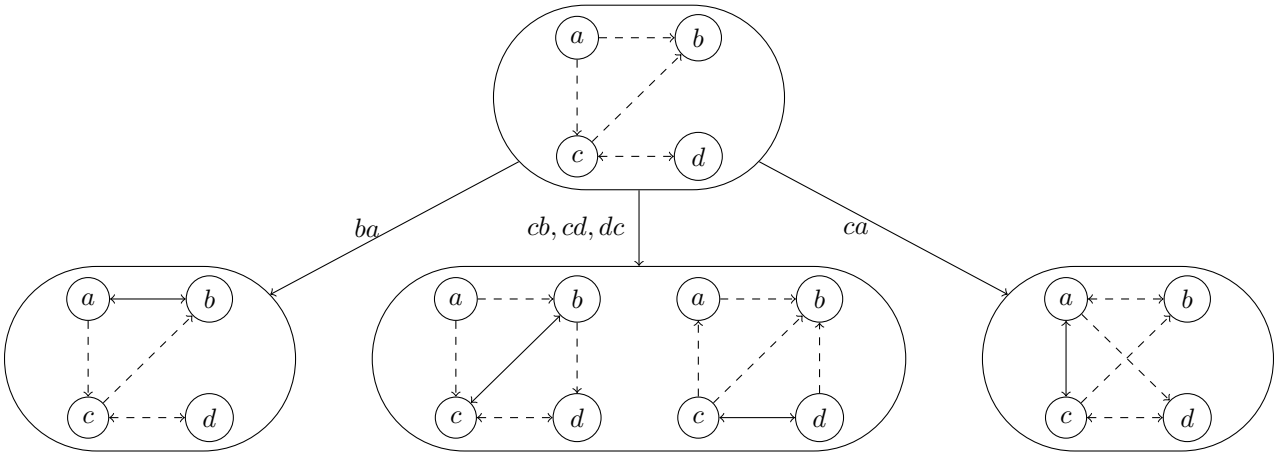


Figure 3.3

We can repeat this process ad infinitum; for a formula of the form $K_bK_a\phi$, we need to find the set of states that agents b considers possible, and then check $K_a\phi$ in these states[2].

In cases where we have some conjunction or negation of formulae with a knowledge modality in, we first build automata where the successful formulae are the conjuncts or formula to be negated. We then perform intersection or complement on these automata to produce an automata where the winning formula is the desired conjunction or negation. This is a very pleasing method and is a great display of why finite-state machines are so well-suited to this task.

Step three; we find paths through the finite-state machine we have built. In our implementation we simply search for a single path, using breadth-first search; however given that we have a conventional finite-state machine one could use a wide array of other techniques to enumerate the set of all successful paths through the automata.

After performing the breadth-first search, we either return the path found - a string of events $e_1e_2 \dots e_n$ - or a negative response to say that no path to a successful state exists. This satisfies the return requirement of a planning problem; we return a set of actions that take us from the initial state to some successful state, as required.

[1]This is not a *perfect* representation of what happens; however simplify the process for now, and we will come onto this later in this section.

[2]Note that in a situation where we have a formula $K_aK_a\phi$ we do not need to build another powerset automata as in an S5 model, $KK \dots KK = K$ (**ModalLogic**). However, for a formula $K_aK_b\phi$ we do; this is because we can see K_i and K_j as being different modalities for different agents i and j.

In our example, we take the automata in Figure 3.3 and search through it to find a state where the successful formula $\forall_{i \in Ag} E_i$ is satisfied. This is an easy task now, as we can evaluate such formulae positionally in $\widehat{\mathcal{ME}^*}$. This yields the string ba, ca, ad, ab, ac, as promised.

We now move onto the details of the system.

3.1 $\mathcal{ME}^*$

Put forward in Aucher, Maubert, and Pinchinat 2014, $\mathcal{ME}^*$ is an automata we use to represent all the worlds that can be reached and how we can move between them given an epistemic model $\mathcal{M}$ and an event model $\mathcal{E}$. Its name refers to how it represents the application of $\mathcal{E}$ onto $\mathcal{M}$ infinitely. Consider that $\mathcal{ME}^0$ represents $\mathcal{M}$, $\mathcal{ME}^1$ would be $\mathcal{M} \times \mathcal{E}$, and $\mathcal{ME}^2$ represents $\mathcal{M} \times \mathcal{E} \times \mathcal{E}$, and so on and so forth. Then we define $\mathcal{ME}^*$ as

$$\mathcal{ME}^* = \bigcup_{n \geq 0} \mathcal{ME}^n$$

One might wonder how we can reason about this; it's a seemingly infinite structure. The key thing to realise is that our worlds in $\mathcal{ME}^*$ can be indexed by the set of propositions that hold true at the world. This makes use of the bijection we discussed in Section 2.2.3. For instance, Figure 2.4b would be represented by the world $q_{\{N_{ab}, N_{ba}, N_{bc}\}}$[3]. Recall that the set of propositions that our epistemic model ranges over has to be finite; consequently, our set of world is finite[4]. The implication of this is that two completely different call paths from the same initial world could end at the same world in $\mathcal{ME}^*$, if the sets of propositions that hold true are the same. It becomes very easy to understand what $\mathcal{ME}^*$ does when we inspect its definition.

Let $\mathcal{M} = (W, R, V)$ be an epistemic model, $\mathcal{E} = (E, P, \mathsf{pre}, \mathsf{post})$ be an event model and $\phi \in \mathcal{L}_P(\Lambda)$ be a formula in propositional logic. ϕ is our goal formula; we want to reach a state in which this formula is satisfied. Then define the automaton $\mathcal{ME}^* = (\Sigma, Q, \delta, q_i, F)$, where $\Sigma = W \cup E, F = \{q_v \mid v \models \phi\}$ and $Q = q_0 \uplus \{q_v \mid v \subseteq \Lambda\}$. We define δ, the transition function, as follows:

$$\begin{array}{ll} \forall w \in W, \forall e \in E, & \\ \delta(q_0, w) = q_{V(w)} & \delta(q_0, e) = \bot \\ \delta(q_v, w) = \bot & \delta(q_v, e) = \begin{cases} q_{v'}, \text{where } v' = \{p \mid v \models \mathsf{post}(e, p)\} & \text{if } v \models \mathsf{pre}(e) \\ \bot & \text{otherwise.} \end{cases} \end{array}$$

$\mathcal{ME}^*$ is designed to accept words of the form $we_1e_2 \dots e_n$, and such a word is accepted if and only if

$$\delta(\dots \delta(\delta(w, e_1), e_2), \dots, e_n) \in F$$

that is, all of the calls are permitted by pre, and the occurence of all of the calls takes us to a world in which the winning formula is satisfied.

We now explain the definition. We can see that our alphabet is the set of worlds in $\mathcal{M}$ and the events in $\mathcal{E}$. See that $\delta(q, \sigma)$ is undefined when $q \neq q_i$ and $\sigma \in W$; we may only make a transition when $\sigma \in W$ if q is the initial world. Recall that the worlds in $\mathcal{ME}^*$ are indexed just by the propositions that hold true at them; we can see this in the definition of Q above. Then when we make a transition $\delta(q_i, w)$ we move to the world $q_{V(w)}$, where V is the valuation function from $\mathcal{M}$. This means that we move to the world indexed by the propositions true at w.

After this, we can make no more transitions $\delta(q, \sigma)$ where $\sigma \in W$. The only other transitions we can make are when $q \in F$ and $\sigma \in E$. In these cases, q is indexed by some set of propositions $v \subseteq \Lambda$. Then given this, we transition to a world $q_{v'}$ where v' is the set of all propositions p where the postcondition $\mathsf{post}(e, p)$ is modelled by the set of propositions v.

We give an example to try and aid understanding. Consider the gossip graph in Figure 3.4a. In $\mathcal{ME}^*$, this would be modelled by the world q_v where $v = \{N_{ab}, N_{ba}, N_{bc}\}$. When call ab occurs at Figure 3.4a, Figure 3.4b is the result. We see that this matches up with the definition of δ for $\mathcal{ME}^*$; take for example S_{ba}. $\mathsf{post}(ab, S_{ba}) = S_{ba} \vee S_{aa}$ is clearly modelled by v as S_{aa} is always true, hence this becomes true at our new world. Contrarily, consider S_{cb}. $\mathsf{post}(ab, S_{cb}) = S_{cb}$, as c is not involved in the call ab.

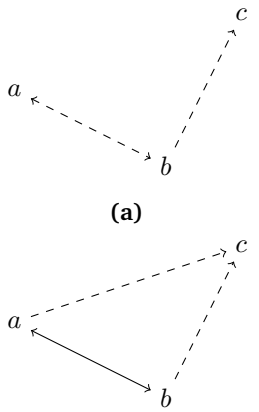


Figure 3.4

[3] Recall that we do not write the propositions N_{ii} or S_{ii}, although they still hold true.

[4] Indeed, we know the exact size of our set of worlds in $\mathcal{ME}^*$, which is $2^{|\Lambda|}$.

Clearly $v \not\models S_{cb}$, and as such S_{cb} is not included in the set of propositions that are true at the new world.

We also see that if the precondition of the event is not satisfied then the world moved to is undefined; we cannot make a transition with an event that may not happen at a world. In our implementation, this is addressed by wrapping up our resulting states in a `Maybe` monad. This means that for an unsuccessful transition we return `Nothing`, whereas in the case of a successful transition we return `Just` the value.

3.2 The Powerset Construction

In 2.5.3 we give a definition of the process used construct a powerset game arena, which constructs a super-arena in which the states of the arena contain in them the set of states *related* to the current one, given the initial game arena and a transducer relating pairs of events. Recall the way we compute the truth value of a formula $K_i\phi$ on some model; we find the set of all worlds indistinguishable to i from i's current world and compute the truth value of ϕ there.

However, $\mathcal{ME}^*$ is unable to perform this task. We have no way of computing the set of possible worlds; as mentioned before, in order to compute the set of states that i considers possible we would have to go back and compute the set of call sequences that i cannot distinguish between and find the state resulting from the application of all of these.

It's clear to see how the powerset construction would be useful in our context; it would give us the set of all states indistinguishable to i built-in to the states. Hence we can quickly evaluate a formula of the form $K_i\phi$ on a state, as we have the set of indistinguishable states already built in to the state - this is what is referred to in this thesis as *positional* evaluation. This is clearly superior to the method detailed above.

The construction of this automaton would let us perform epistemic planning when our successful formula is of the form $K_i\phi$. Currently, with $\mathcal{ME}^*$, we may only plan when our successful formula is in $\mathcal{L}_P(\Lambda)$. When planning we visit each state and evaluate the successful formula on it. We are unable to evaluate knowledge-based formulae on states in $\mathcal{ME}^*$; hence we are unable to plan for such formulae. The construction of a powerset automata of $\mathcal{ME}^*$ - referred to as $\widehat{\mathcal{ME}^*}$ - gives us this desirable property.

In order to perform this task we need to construct a transducer that relates pairs of events indistinguishable to i. This can be built very easily from the event model supplied, as in here these pairs are already stored. This is one of the requisites for the powerset construction as outlined in Section 2.5.3 and refined for our use-case in this section. In this section we give a method with which these transducers can be built, how we compose them to make them more useful for our powerset construction, and finally the adaptation of the powerset construction for finite state automata.

3.2.1 Event Transducers

In conjunction with $\mathcal{ME}^*$, we also need to produce a set of transducers $\mathcal{T} = \{\mathcal{T}_i \mid i \in A\}$, for the model in question. $\mathcal{T}_i$ relates pairs of events that agent i cannot distinguish between. This is critical to us performing the powerset construction on our $\mathcal{ME}^*$ automata. Hence we want $[\mathcal{T}_i]$ to be equal to the set of pairs of calls indistinguishable to agent i[5]. Formally, this means that, given an event model $\mathcal{E} = (E, P, \mathsf{pre}, \mathsf{post})$, $[\mathcal{T}_i] = P_i$.

In order to do this we need just one state in our transducer, which we simply call q. The choice of using a one-state transducer is slightly non-standard, however we hope it becomes clear once we see how it is used in Section 3.2.2. In short, we choose this because we just care about relating the states that are indistinguishable; the state of the model has no impact on this. Then given an event model $\mathcal{E} = (E, P, \mathsf{pre}, \mathsf{post})$, we define $\mathcal{T}_i = (\Sigma, Q, \Delta_i, q, F)$. $Q = F = \{q\}$ and $\Delta_i = \{(q, e, e', q \mid (e, e') \in P_i)\}$. It is quite simple to see that this definition will satisfy the above requirement.

3.2.2 Transducer Composition

Essential to this stage of the algorithm is the construction of a transducer to relate a state with the states considered possible after some event happens by some agent. This requires two components, the first of which is the transducer

[5] As an example, consider a gossip graph with three agents. Then $[\mathcal{T}_a] = \{(ab, ab), (ac, ac), (ba, ba), (ca, ca), (bc, bc), (bc, cb), (cb, bc), (cb, cb)\}$.

defined in 2.5.3, which relates pairs of events that are indistinguishable to an agent. This can be constructed with ease from the event model, as discussed earlier. The second component is the identity transducer over the automata $\mathcal{ME}^*$. This can be computed in another very straightforward manner. Given $\mathcal{ME}^* = (\Sigma, Q, \delta, q_0, F)$, we can construct the identity transducer $\mathcal{T}_{\text{id}} = (\Sigma, Q, \Delta, q_0, F)$ where $(q, e, e', q') \in \Delta$ iff $\delta(q, e) = q'$ and $e = e'$. This is useful as it lets us compute the effect of an event occuring and whether the event is permitted at the world in question. The single-state transducer computes which events *could* happen, whilst the identity transducer computes the effects of these events. $\mathcal{T}_{\text{id}}$ is not useful on its own, however it becomes important once we see the next step.

The next step is to compose the two. We use a slightly different method than those put forward in the literature (e.g. **ComposingFSTs**), given that one of our transducers has only one state; hence, it would not make sense to need to move on from the state in the first transducer, as is classic. For a single-state transducer $\mathcal{T}_1 = (\Sigma, \{q\}, \Delta_1, q, \{q\})$ and a transducer $\mathcal{T}_2 = (\Sigma, Q, \Delta_2, q_0, F)$, we define the composition $\mathcal{T} = (\Sigma, Q, \Delta, q_0, F)$ where

$$\Delta(s, e) = \{(e', s') \mid (e', q) \in \Delta_1(q, e), (e', s') \in \Delta_2(s, e')\}$$

Note that $\mathcal{T}$ is nearly identical to $\mathcal{T}_2$; the states, alphabet, initial and final states are the same. Recall that this is because we use $\mathcal{T}$ to relate a world - call pair to a set of worlds that agent i considers it possible to be in from the prior worlds.

Hence, if we give as input to Δ an input pair (q, e), where q is a state and e an event, it returns a set of pairs (e', q'), where e' is some event indistinguishable from e and q' is the state we reach through the occurence of e' at state q. This transducer is the one used in the definition of the transition relation in Section 3.2.3. This makes the transitions in the powerset automaton much more simple to implement.

3.2.3 The Construction

Given an automata $\mathcal{ME}^* = (\Sigma, Q, \delta, q_i, F)$ and a transducer $\mathcal{T}_i = (\Sigma, Q, \Delta, q_0, F)$[6], we construct a powerset automata $\widehat{\mathcal{ME}^*} = (\widehat{\Sigma}, \widehat{Q}, \widehat{\delta}, \widehat{q}_i, \widehat{F})$. In this automata, our states $\widehat{q} \in \widehat{Q}$ are of the form (v, S), where $S \subseteq Q$. This is because when we need to evaluate a formula of the form $K_i\phi$, we need to know the set of states indistinguishable to agent i at this moment. The set S serves this purpose; it contains all of the states that are indistinguishable from our current state to agent i. The components of $\mathcal{ME}^*$ are as follows.

- $\widehat{Q} = Q \times \mathcal{P}(Q)$
- $\widehat{\Sigma} = \Sigma$
- $\widehat{\delta}((v, S), \sigma) = (v', S')$ where
  - $v' = \delta(v, \sigma)$
  - $S = \{q' \mid \exists q \in S, \exists \sigma' \in \Sigma, (\sigma', q') \in \Delta(q, \sigma)\}$
- $\widehat{q}_0$ is the state (q_0, S) where S is the set of all worlds indistinguishable from q_0 in $\mathcal{ME}^*$.
- $\widehat{F} = \{(v, S) \mid v \in F\}$.

We see that this version is much simpler than the definition given in Section 2.5.3. This is because we don't need to keep track of the set of potential final states of the transducer's output tape; we only have to keep track of the other possible states of the transducer. The set of states in the transducer is exactly the same as the set of states in our $\mathcal{ME}^*$ automata, so these transducer states correspond directly to the states that we *could* be in in our $\mathcal{ME}^*$ automata.

Note that we can update $\widehat{F}$ to accept worlds in which $K_i\phi$ is made true very easily. Assuming that in the automaton the layer below, all the states in F are those which make ϕ true, the definition becomes $\widehat{F} = \{(v, S) \mid \forall w \in S, w \in F\}$[7].

From the definition of transitions in $\widehat{\mathcal{ME}^*}$, we can see that S' becomes the set of all states q' that can be reached by making a transducer transition with the input event σ from some state $q \in S$. If we recall the definition

[6] Remember that this transducer is the composition of the event-pair transducer and the identity transducer over $\mathcal{ME}^*$, as discussed in Section 3.2.2.

[7] Note that given that we are working in an S5 model, we know that $v \in S$. This is due to the *reflexivity* of the accessibility relations in an S5 model. For this reason it is enough to just inspect the states in S.

of our transducer from Section 3.2.2, a transition $\Delta(q, \sigma)$ gives us a set of pairs (σ', q') where σ' is some event indistinguishable from σ to agent i, and q' is the state we reach from making that transition from state q.

We now give an example execution of this process. Consider the gossip graph in Figure 3.5. We want to get to a state where the formula $K_a S_{bc}$ is true; that is, we want to get to a state where agent a knows that agent b knows the secret of agent c. One call sequence that will guarantee this is $bc; ab$; b calls c to discover c's secret, and then a calls b. After the call bc occurs, a knows that some call has happened between the other agents, but can not tell if it was from c to d or b to c; as such, a considers both of these events possible. a knows that they are in the state reached by the occurence of either event, but does not know which. Once a calls b, a learns that b knows the secret of c, and hence the formula $K_a S_{bc}$ is satisfied.

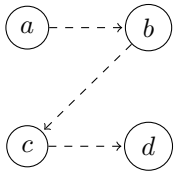


Figure 3.5

This is of course a very high-level description, and is unfortunately not how the program will reason about it. Let us step through one application of the event model $\mathcal{E}$ on this graph. In this case, our event model $\mathcal{E}$ is the set of all possible calls, similar to Figure 2.5; however only three are permitted. The effects of updating Figure 3.5 are seen in Figure 3.6. We can see this as being the automata $\mathcal{ME}^1$. We hope that the reader can see how the structure $\mathcal{ME}^*$ would arise by repetition of this.

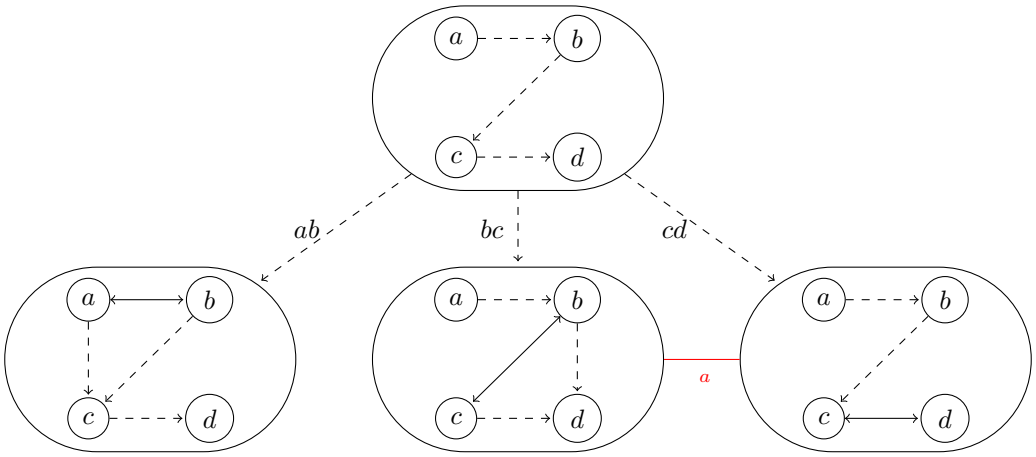


Figure 3.6: *The gossip graph in 3.5, updated with $\mathcal{E}$ once.*

Figure 3.6 shows the three possible states that we could be in afer making one call from the initial state displayed in Figure 3.5. The red line indicates that agent a cannot distinguish between these two states, as they were both reached by the execution of a call that agent a was not a member of. Note that in the algorithm this connection does not exist; it's here simply for labelling. We can see the three states on the lower row as a Kripke model, where the three are worlds labelled by their valuation, and the two connected by the red line are indistinguishable to a.

Now say that we wanted to evaluate the formula $\neg S_{ba}$ on the bottom-right world in Figure 3.6. This is very straightforward; we just need to check if b does indeed know a's secret; here it doesn't, and so the world satisfies $\neg S_{ba}$. But what if we want to evaluate the formula $K_a \neg S_{ba}$? We need to visit each of the states indistinguishable from the state in question and see if $\neg S_{ba}$ holds there. Recall that, although the red line connecting the two bottom states indicates that agent a cannot distinguish between these two worlds, this connection *does not exist* in the automata; it is there just for our intuitions. $\mathcal{ME}^*$ has *no notion* of two states being indistinguishable from one another. Hence, this is currently impossible; $\mathcal{ME}^*$ has no idea whether agent a cannot distinguish between this world and any others, and certainly no way of reaching such worlds. This is where the powerset construction comes into play; we pull the indistinguishable states into the states themselves, yielding the possibility for *positional evaluation* of $K_a \neg S_{ba}$. After this process, Figure 3.6 becomes Figure 3.7.

The difference between Figures 3.6 and 3.7 now is that the former has all of the states indistinguishable from itself to agent a *inside* the state. This means that we can evaluate the formula $K_a \neg S_{ba}$ at a state by simply inspecting the states indistinguishable from the one we're looking at to check a formula of such form. This is now an extremely straightforward task!

In the states in Figure 3.7 we display the *true* state on the left. This is the state that contains the true information of the state reached by making the corresponding call. This is very important, as when an agent

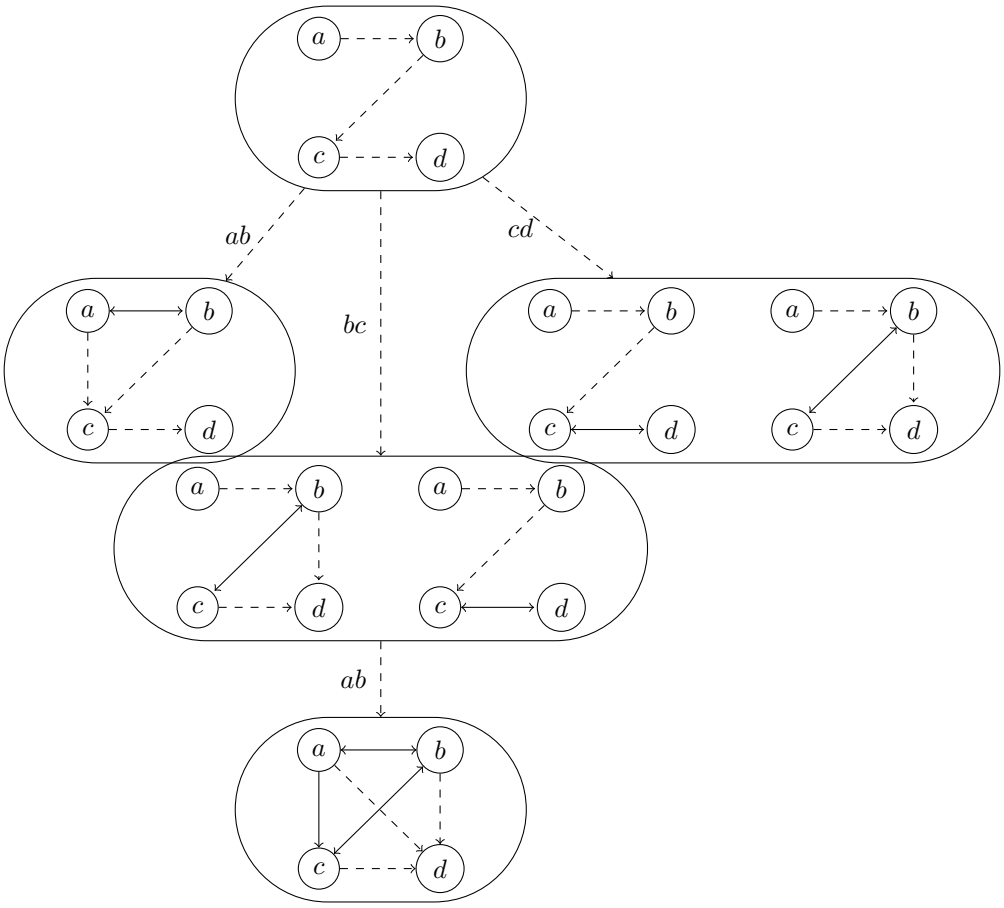


Figure 3.7: *The automata in Figure 3.6, now with the indistinguishable states in the states.*

calls another they become aware of the knowledge state of the other. To revisit the example from the start of this section, consider again the formula $K_a S_{bc}$). Recall that we said a successful call sequence would be bc, ab. Observe that in the state reached from the initial state by the call bc in Figure 3.7 there are two states that cannot be distinguished between by a. However, upon the event of a calling b, a learns that b knows the phone number of c - this is the *underlying* state that we talk about. Hence the state at the bottom of Figure 3.7 contains only one possible world; a no longer considers the other possible, as the knowledge of b is different. Happily, this leads us to a state in which $K_a S_{bc}$ is satisfied, just as we promised.

3.3 Combining Automata

When we encounter a formula of the form $\phi \wedge \psi$, where $\phi, \psi \in \mathcal{L}_P(\Lambda)$, we can simply apply the method described in Section 3.1 to produce an automata whose successful states are those in which $\phi \wedge \psi$ are made true. The production of an automaton whose successful states are those in which a formula χ is made true is a process that we call *constructing an automata to solve* χ.

However this is less straightforward when we encounter a formula of the formula $K_a\phi \wedge K_b\psi$ [8]; in this case, we want to produce powerset automata for $K_a\phi$ and $K_b\phi$ and then join them back up.

Luckily, intersection for finite automata is well-defined. We give the definition here. For two automata $A = (\Sigma, Q_A, \delta_A, q_{0A}, F_A)$, $B = (\Sigma, Q_B, \delta_B, q_{0B}, F_B)$, we define their intersection $A \cap B = (\Sigma, Q, \delta, q_0, F)$ where $Q = Q_A \times Q_B$, $F = F_A \times F_B$, $q_0 = (q_{0A}, q_{0B})$ and $\delta(q_A, q_B) = (\delta_A(q_A), \delta_B(q_B))$. This definition expands in the simple way when we want to take the conjunction of n automata.

We have a similar story for negation; given a formula $\neg K_a\phi$, to create an automata to solve this we build

[8] This occurs very frequently; in this gossip problem, we often want to reach a state where every agent knows that everyone is an expert.

the automata to solve $K_a\phi$ and then take the complement of this. This consists of simply changing the set of accepting states. Given automata $A = (\Sigma, Q, \delta, q_0, F)$, we define $A_\neg = (\Sigma, Q, \delta, q_0, F_\neg)$ where $F_\neg = Q \setminus F$.

To complete our little family of set operations, we should define disjunction too. Taking the union of automata is, algorithmically, less straightforward than the intersection; hence we just exploit de Morgan's law and use the identity $A \cup B = (A^C \cap B^C)^C$. We recognise that when we come to implementation this may be more costly than some alternative implementation, but for now this is of little importance.

3.4 Searching

Then given our solving automata, we want to traverse it to find paths that takes us from the initial state to some successful state. It is in here that the process of planning is really performed; everything up to this point has been merely preparation for this.

In this stage, we simply view the automata as a *directed graph* and perform breadth-first search on the automata. Each of the edges are weighted equally[9]. This way we find the *shortest path* from the root note to one of the final nodes. This is one reason why breadth-first search was chosen over depth-first; another is that breadth-first search is less prone to one of the pitfalls of **GithubGossip** which is that it can infinitely loop sometimes when using a gossip protocol like ANY. Protocols like ANY can allow for infinite call sequences. A depth-first search strategy will result in the searching of infinite call sequences; this means that the program will loop infinitely and never return. With a breadth-first search this is less likely to happen, although in the case of there being no successful state in the graph this is still possible. To remedy this we pre-process the automata to remove all transitions from a state to itself. This means that there is no scope to repeatedly investigate a state, and as such no risk of an infinite sequence of events where an event is repeated forever.

[9] If we had some situation where certain communication channels were more expensive or slower in a network, we could emulate this by making some of the edges weighted.

Chapter 4

Implementation

We now come to covering the implementation of the algorithm developed in the previous chapter. We will first give an overview of the structure of the software and then a summary of all design decisions taken throughout.

For the implementation of our algorithm, we chose to use Haskell. Haskell is a lazy functional programming language, which gives us several benefits for this particular task that other languages lack.

The first one is the syntax; its functional style lends itself very well to mathematical definitions. Haskell's list comprehension and pattern matching let us write code that very closely mimics the original notation, thus erasing a lot of the difficulty of converting a mathematical definition to code once we have derived it.

Another benefit is its laziness. In this algorithm we often use automata with a ridiculously large state space; however (thankfully) Haskell's laziness means that we never have to enumerate these states. We only access them when we need them. This saves a lot of processing time, as well as greatly reduces the space the program takes up.

Haskell's strong type system and typeclasses support us and give us safety guarantees, preventing us from writing erroneous code that in another language may only be detected at runtime. A simple example is that in our implementation we cannot represent a formula that is not well-formed[1]; a more sophisticated one is the ability to make sure that an epistemic model and an event model span the same set of atomic propositions and events. Typeclasses help us make code that is very reusable but also safe; if we wanted to use a new language of atomic propositions for an existing model, we would only need to give an instance of the `Prop` typeclass for our new language and hey presto.

The classification of functions as first-class objects also comes in very helpful. In the implementation we make the transition function of an FSM a function $\delta : Q \times \Sigma \to Q$. Often we want to "lift" the states Q to some other data type, e.g. $P(Q)$. We can very easily unwrap the datatype P to access the underlying Q and then use the previous δ function. This can be used to make another function operating on elements of type $P(Q)$. The ease with which we can create, manipulate and compose functions is unique to Haskell, and makes tasks like conjoining automata very simple.

Haskell offers a simple way to divide each file into a module and import a module into another. We give a brief overview of each module in the system, and highlight notable uses of any of the above language features in this process. We also highlight any notable design choices made.

All of the code discussed in this section and associated with this thesis can be found at `https://github.com/leopoulson/gossip-planning`. In the code samples in this thesis we often remove or shorten certain functions in order to better display the aspects of the code we *want* to show, and remove the irrelevant clutter.

4.1 `Model.hs`

This module is the foundation of the system. Here we implement the structures defined in Section 2.1; we have the language of our formulae, Kripke models, event models and updates thereof.

In Figure 4.1, we can see the implementation of both Kripke models and event models in Haskell. `states` and `events` respectively represent the states and events of the Kripke and Event models, and `eprel` and `evrel` respectively represent the indistinguishability relations between them. Note that for ease of implementation we

[1] This is one of my favourite examples; the ease at which one can formulate and give a semantics to sentences in a logical language in Haskell is so straightforward and easy that it's inconceivable to try to do this task in any other language.

use *partitions* (**EREL**) of sets rather than sets of pairs. This is done because we regularly want to just access the set of states we cannot distinguish an item between, and the use of partitions makes this a much simpler task.

```
data EpistM ev prp = Mo {
  states :: [State ev],
  agents :: [Agent],
  val :: [(State ev, [Form prp])],
  eprel :: [(Agent, [[State ev]])],
  actual :: [State ev],
  allProps :: Set prp
}
```

(a) *The Kripke model datatype*

```
data EventModel ev prp = EvMo {
  events :: [ev],
  evrel :: [(Agent, [[st]])],
  pre :: ev -> Form prp,
  post :: (ev, prp) -> Form prp
}
```

(b) *The Event model datatype*

Figure 4.1

In Figure 4.1 we see the first leverage of Haskell's type system. Through making our models parametric in the propositions of the language used (formally referred to as Λ) and the events used. These parameters mean that when we use them in a function they necessarily use the same events and propositions. Consider the function signature for `update`.

```
update :: Prop p => EpistM ev p -> EventModel ev p -> EpistM ev p
```

Figure 4.2

If we try and use the `update` function with models using different types of events or propositions we get a compiler error. This type safety, and ease of use of it, is one of the main attractions of Haskell for this project, giving us a type-safety unavailable in other languages.

One may also notice in Figure 4.2 the use of the typeclass `Prop p`. This typeclass guarantees us a function `evalState :: p -> EpistM st p -> st -> Bool`, which given a model, propositions and a state within the model, returns a boolean value expressing whether or not the proposition in question holds at the state. This may seem nonsensical - given that we have no information about what the state is from the type, we can not do anything too interesting. It becomes useful in the case of always true propositions like N_{ii}; an agent will always know their own phone number.

The user then defines their own pre- and post-condition functions. One deviation from the mathematical notation is the addition of the set of agents; this is purely a pragmatic thing, as it makes certain operations more straightforward later on.

```
update :: (Eq ev, Prop p) => EpistM ev p -> EventModel ev p -> EpistM ev p
update epm evm = Mo states' (agents epm) val' rels' (actual epm) (allProps epm)
  where
    states' = [stateUpdate s ev | s <- states epm,
                                  ev <- events evm,
                                  satisfies (epm, s) (pre evm ev)]
    rels' = [(ag, newRel ag) | ag <- agents epm]
    newRel agent = filterRel states' [liftA2 stateUpdate ss es |
                                      ss <- fromMaybe [] (lookup agent $ eprel epm),
                                      es <- fromMaybe [] (lookup agent $ evrel evm)]
    val' = [(s, ps s) | s <- states']
    ps s = [P p | p <- props, satisfies (epm, trimLast s) (post evm (lastEv s, p))]
    props = allProps epm
```

Figure 4.3: *The $\mathcal{M} \times \mathcal{E}$ function*

Now we see the definition for the update of a Kripke model with an event model. Whilst the agents, set of propositions and actual world remain the same, we update the states, epistemic relations and valuation function in

a way very similar to the definition in Section 2.1.2. This is a display of the merits of Haskell; our implementation stays very close to the specification, allowing for simple visual verification that the code that we have written matches the specification.

4.2 `FSM.hs` and `FST.hs`

These two modules hold our finite state machines. Given how much of the algorithm relies on state machines, it was very important in development to have a reliable structure for state machines. Furthermore, we needed to do some non-standard operations on the machines we implemented (namely, the composition mentioned in Section 3.2.2). To do this easily we need low-level access to the states and a good understanding of the library.

The two libraries that seemed most suitable were **HaskellFST** and **HaskellMachines** However, both libraries carried complexity unnecessary for our use-case; furthermore it was particularly unclear how we would go about implementing our single-state composition as mentioned above.

Fortunately, implementation of these finite-state machines is incredibly straightforward. We can see in Figure 4.4 that the datatypes nearly identically mimic the tuple definitions in Section 2.5.2.

```haskell
data FSM ch st = FSM {
  alphabet :: [ch],
  states :: [st],
  transition :: (st, ch) -> Maybe st,
  initial :: [st],
  accepting :: st -> Bool
 }
```

(a) *The FSM datatype*

```haskell
data FST ch st = FST {
  alphabet :: [ch],
  states :: [st],
  bitransition :: (st, ch) -> [(ch, st)],
  initial :: [st],
  accepting :: st -> Bool
}
```

(b) *The FST datatype*

Figure 4.4

Our FSM's transition returns a `Maybe st` to encode that it is *nondeterministic*; meaning that it is not guaranteed to return a state on a transition. This comes in useful in practise when we have as input to δ some pair (w, e) where e is not permitted at w. In this case, $\delta(w, e) =$ `Nothing`. An example of this is a call ij in a gossip model where i does not know j's phone number.

Similarly, the FST transition returns a list of (e, q) pairs. This again encapsulates the nondeterminism of the FST. This is useful in the case of Section 3.2.2, where we want to return from $\Delta(q, e)$ the set of all pairs (e', q') where e' is some event indistinguishable from e and q' is the state reached by making the transition $\delta(q, e')$.

A decision we made in the design of the machines was the encoding of transitions as *functions*, rather than as a map. We chose to do this for a number of reasons. Firstly, we never wish to enumerate the set of transitions in the graph, a task which is made easier through the use of maps. Further, use of a map requires us to "be strict in our keys, but lazy in the values". Although better than being strict in keys *and* values, this is still quite unpleasant; consider $\mathcal{ME}^*$. We want to be able to mimic from every state to every other, even the impossible states[2]. When using a map, this would mean we would require an entry for all of these impossible state. This means that a lot of time would be spent on entering keys into the map for states that *will never be used*. However, when we use a function, these inputs are covered for free. This is a much easier way to handle transitions. A similar argument holds for the `accepting` function.

The use of functions here is a method that can only be easily utilised in languages where functions are treated as first-class objects (Haskell is one of these). This treatment of functions also makes operations like intersection and complement of FSMs a simplicity; in the latter case, we simply need to compose the `not` function before the accepting function.

4.3 `ME.hs`

[2]We can imagine an "impossible" state as a state that we would never reach in the model. For example, in the gossip problem, an impossible state would be one in which everyone knows everyone else's secret, but no one knows anyone else's phone number.

We now come onto the module that handles construction of the automata $\mathcal{ME}^*$. This is a slightly more interesting case.

In this module we make use of one of the many language extensions Haskell offers us, namely `MultiParamTypeClasses`. This gets around the limitation that GHC enforces of being unable to use a typeclass with more than one type parameter, allowing us to write typeclasses like this in Figure 4.5.

```
class (Ord st, Prop p) => EvalState
  evalState :: Form p -> st -> Bool
```

Figure 4.5

This lets us ensure in our functions that the propositions we're using can be evaluated at the states we want them to be evaluated at. This prevents us from trying to evaluate formula of a certain language on a world it is incompatible with. This is another display of the way we can use Haskell's type system to our advantage.

```
meTrans :: (Prop p, Eq ev) => EpistM ev p -> EventModel ev p -> ((QState p, ev)
-> Maybe (QState p))
meTrans (Mo _ _ _ _ _ props) evm    (Q ps, ev)
    | not $ ps `models` pre evm ev  = Nothing
    | otherwise                     = Just . Q $ filter (\p -> ps `models` post evm (ev, p)) props
```

Figure 4.6

In Figure 4.6, we can see the implementation of the transition function of $\mathcal{ME}^*$; we give an epistemic model and an event model and we are returned a function of type `(QState p, ev) -> Maybe (QState p)`. The `Nothing`s encode a call that is not permitted to occur, and as such is undefined. When an event is permitted, we travel to the corresponding state as per the definition of the transitions in $\mathcal{ME}^*$ in Section 3.1. The definition of $\mathcal{ME}^*$ lends itself perfectly to the use of transitions as functions rather than maps; we behave the same no matter what the inputs are; there are no special cases we need to look out for. In this way, $\mathcal{ME}^*$ represents each possible transition between states.

Remind yourself that our problem with $\mathcal{ME}^*$, and the motivation for Section 2.5.3, is that $\mathcal{ME}^*$ has no means to handle indistinguishable states. This is still true here; our implementation has no notion of what an indistinguishable state is. In the case that our successful formula is in $\mathcal{L}_P$, we may simply use $\mathcal{ME}^*$ to produce a structure which we can use to solve the planning problem; however if our successful formula is epistemic, then we need to apply the powerset construction to $\mathcal{ME}^*$.

4.4 `Powerset.hs`

```
data PState st = PList [PState st]
               | PCon (PState st)
                      [PState st]
               | PVar st
```

Figure 4.7: *The `PState` datatype.*

This module contains all of the functions pertaining to the powerset construction, as specified in Section 3.2.

The general process is as follows. Say that we want to plan with an epistemic model $\mathcal{M}$, an event model $\mathcal{E}$, and a successful formula $K_a\phi$, where $\phi \in \mathcal{L}_P(\Lambda)$. We first produce an $\mathcal{ME}^*$ automaton with $\mathcal{M}$ and $\mathcal{E}$, and set the successful formula to be ϕ. We now want to perform the powerset construction and bring all of the possible states we could be in into our states, so that $K_a\phi$ can be evaluated positionally. We convert the states in $\mathcal{ME}^*$ to states of type `PState` (these are discussed below), and then perform the powerset construction detailed in Section 3.2 on this automata. Finally, we have an automata with which we can evaluate the winning condition positionally; we can easily compute a successful path now.

In Figure 4.7, we see our first deviation from the algorithm in Section 3.2. Recall that we said that states in an automata $\widehat{A}$ are of the form (v, S), where v is a state of A and S is a subset of the states of A. The `PCon` constructor solves this; holding in it the state v in the first argument and the set S in the second. `PVar` serves to hold the *underlying states* - this can be seen as it contains a value of type `st`. These are used in the step discussed above, where we convert out states in $\mathcal{ME}^*$ to states of type `PState`. Finally, `PList` serves to help us perform intersection and union operations on automata.

`PVar` is the place where the formulae are actually evaluated. For instance, when we have a formula $K_a\phi$, our automata states are of the form `PCon (PVar st) [PVar st]`. The evaluation of the knowledge is simple; we just

go to each `PVar` in the list and see if ϕ holds there; this will be done by checking ϕ against the formulae that are true at the states in `PVar`. This process is the same if our successful formula is of the form $K_a K_b \phi$; we take our states[3], take the set of indistinguishable states, and evaluate $K_b \phi$ there. For any modal formula $K_i \phi$ we perform the same process, no matter what ϕ is.

The reason we do this is so that we can create our automata recursively. Consider the function in Figure 4.8[4]. This is the function we use to create the automata we traverse to find successful sequences of events; it pattern matches on the formula given in the first argument to decide the "shape" of the returned automata.

Let us first consider the bottom case. We receive some formula, e.g. $S_{ab} \wedge S_{bc}$. Then we build an automata $\mathcal{ME}^*$ whose successful states are those which satisfy the formula $S_{ab} \wedge S_{bc}$, through the procedure in Section 4.3.

If it receives a formula $K_a \phi$, then we call the `buildPSA` function, which performs the powerset creation process as detailed earlier. Note that it does so with the automata that solves the formula ϕ, invariant of the form of ϕ - it could be an atomic proposition or another modality, or a conjunction of modalities.

```
cSA :: Form p -> EpistM (State ev) p -> EventModel ev p
       -> FSM (Character ev) (PState (QState p))
cSA form@(K agent phi) ep ev
  = buildPSA
              (cSA phi ep ev)
              (buildComposedSS agent ep ev (cSA phi ep ev))

cSA (Not phi) ep ev = complement $ cSA phi ep ev tfilter

cSA (And phis) ep ev = case includesK (And phis) of
  True  -> toPList $ intersection $ map (\phi -> cSA phi ep ev) phis
  False -> makeP $ buildMEStar (And phis) ep ev

cSA phi ep ev = makeP $ buildMEStar phi ep ev
```

Figure 4.8: *The `createSolvingAutomata` function.*

When our successful formula is of the form $K_b K_a \phi$, we can very easily *lift* our transducers, as detailed in Section 2.5.4. Recall that we compose the single-state transducer that relates pairs of events with the identity transducer of $\mathcal{ME}^*$ in order to perform the powerset construction on $\mathcal{ME}^*$ to yield $\widehat{\mathcal{ME}^*}$. In order to produce $\widehat{\widehat{\mathcal{ME}^*}}$, we repeat the powerset construction, this time stepping through $\widehat{\mathcal{ME}^*}$ to create an automata whose states are of the form (v, S) where v is a state in $\widehat{\mathcal{ME}^*}$ and every $s \in S$ is a state in $\widehat{\mathcal{ME}^*}$. We compose the transducer relating pairs of events with the identity transducer over $\widehat{\mathcal{ME}^*}$ to generate the automata used in the powerset construction. This process then telescopes in the natural way depending on the number of K modalities in the successful formula.

[3] Here our states are of the form `PCon (PCon (PVar st) [PVar st]) [PCon (PVar st) [PVar st]]`.
[4] For clarity and brevity, we omit some of the supporting function calls, and also omit some of the cases.

Chapter 5

Evaluation

5.1 Plan for Testing

Throughout development we used the Haskell library `HUnit`. This provides a wealth of combinators that we can use to concisely write unit tests for our functions. This was used in a standard manner; before implementing a function we would write tests for it, and then implement the function, ensuring it passes all tests and as such functions correctly.

However, more interesting is our plan for functional testing. This is the part of testing in which we check that the system *correctly functions*, and will tell us if what we have implemented is correct or not.

Although our system is capable of planning for any epistemic model and propositional event model, we chose to test solely on gossip models. This is for several reasons:

- Gossip models cover every part of the system (epistemic models, $\mathcal{ME}^*$, the powerset construction) and as such let us test all the functionality of the code;
- Testing with a certain class of models tells us that all of the code works; the algorithm does not depend on what the particular model is. Hence if we know it is correct for a certian class of models, it will also be correct for any other class of models;
- We have an existing model checker specialised for the Gossip problem that we can use (**GithubGossip**);
- It is the class of models which I have the most experience with; hence if an error arises it should be quite easy to understand.

This in mind, we chose to use **GithubGossip** to verify our results. It was chosen over the other two tools (**SMCDEL DEMO-S5**) due to its ease of accessibility; we can provide it with just a list of lists encoding who knows who's phone number, and it will produce an epistemic model for us. This is very easily produced from the epistemic models in our software. This is a sharp contrast to **SMCDEL** which uses a text file based input - output interface which would have added significant complication to the testing process. **DEMO-S5** was unsuitable for testing due to it only being able to model *public announcements*. Events in the Gossip problem are not public, rather being between just two agents; hence **DEMO-S5** is unable to model calls in the Gossip problem, and as such can not be used to verify our results.

Then the question arises of how we use the selected tool. We generate a sequence of calls with our tool (as a solution to the planning problem). We then convert the initial gossip graph, the successful formula and the successful call sequence to their representation in **GithubGossip** and check that the call sequence applied to the initial graph take us to a state in which the successful formula is made true. In the case that our tool tells us there is no successful call sequence given the initial model, we use the other tool to check that there's no successful path to take us to a successful state. We define the former case as a *positive* result from our tool, and the latter a *negative* result from our tool. We say that the two tools *agree* if the call sequence returned from our tool takes the gossip graph in question to a successful state; or, in the case of a negative result from our tool, that there are no successful paths returned from the other tool.

5.2 Testing Frameworks

The next decision to be made was how to actually go through with the testing process. Haskell offers the seminal property-based library `QuickCheck`, which, when given a function `a → Bool` and a way to generate arbitrary values of type `a`, will test the function on these arbitrary values and ensure that the function returns `True`. All that we need to provide in order to generate these arbitrary values is an instance of the `Arbitrary` typeclass. We can see an example of this in Figure 5.1.

```
instance Arbitrary (EpistM StateC GosProp) where
  arbitrary = standardEpistModel agents <$> (sublistOf $ allNumbers agents)
```

Figure 5.1: *An `Arbitrary` instance for an epistemic model.*

Once we have this, we perform a breadth-first search on the automata generated by this epistemic model and the target formula in order to find either a path that takes us to a successful state, or a response telling us that no such path exists. Whatever the outcome, we then send this result and the original model to a function that converts our model to a model in the style of **GithubGossip** and uses the model checker there to verify our answer.

`QuickCheck` was working well, however I quickly wanted to be able to check for every possible gossip graph. `QuickCheck` will sometimes repeat instances [1]; hence to get every single possible gossip graph I simply created a list with every single graph in and performed the same process of using the model checker from **GithubGossip** on each graph. Naturally, this technique is impractical for numbers of agents greater than 4; there are thousands of possible gossip graphs for 5 agents, and to enumerate them would be a long, slow and mostly boring task. However, there is little difference between testing with 4 agents and testing with 5[2]; hence it suffices to exhaustively test each 4-agent gossip graph and use `QuickCheck` to test gossip graphs of 5 or more agents. This combination of the two testing methods ended up being the most practical for our situation.

The generation of gossip graphs is very straightforward; we just produce some random set of propositions $T \subseteq \{N_{ij} \mid i, j \in A, i \neq j\}$. Once we have this set T we union it with the set $ID = \{N_{ij} \mid i, j \in A, i = j\} \cup \{S_{ij} \mid i, j \in A, i = j\}$. This latter set is the set of "identity" propositions; we can see them as being vacuously true, as all of the agents are bound to know their own phone number and secret.

We only look for cases where the agents start out not knowing the secret of any agent, only phone numbers - we call these *phonebook* graphs. There are a few reasons for this;

- These cases are the only interesting ones. Any gossip graph where some agent already knows the secret of another will either have the same call sequence or a shorter one than the same graph, where no agent knows the secret of any other.
- These cases are the most faithful to the applications of the gossip problem in reality.
- We want to generate well-formed test cases, in order strictly test *realistic* inputs. The only requirement that we put on a test case for it to be well-formed it that an agent cannot know the secret of another agent without knowing their phone number as well. If we remove the chance for an agent to know another's secret in the initial test cases, we satisfy this requirement.
- On a pragmatic note, it is more straightforward to convert phonebook graphs to the representation in **GithubGossip** than non-phonebook graphs.

5.3 Correctness Results

The results from functional testing were on the whole positive. Our tool and **GithubGossip** agreed on all test cases where the gossip graph was of a size less than 4. However, as mentioned earlier, these test cases are the "uninteresting" ones; indistinguishable states are not generated, and as such the whole breadth of the system is

[1] To verify this, simply open up your nearest Haskell REPL, import `QuickCheck`, and run the line of code `verboseCheck ((\s -> s == s) :: Bool -> Bool)`

[2] In contrast, consider the difference between testing with 3 agents and 4. When we have 3 agents, a call that does not involve an agent a can only be from either b to c or c to b; however, these are essentially the same call. With 4 agents, a call not involving agent a can be from b to c, or b to d, or c to d, and so on. The latter case generates interesting indistinguishable states; the former does not.

not being tested. As such, we will only look at cases where the number of agents in the gossip graph is greater than or equal to 4.

Having fixed the size of gossip graphs to be 4, we now vary the successful formula in question. We abbreviate the statement "all agents are experts" to the proposition E. We investigate two things; the *conjunction* of formulae and the *order* of formulae[3].

For E and $K_a E$[4], the two tools agree on all gossip graphs.

Once we address second-order and higher successful formulae, we start to get disagreeing results. We tested each possible initial gossip graph, of which there are 4096. For the successful formula $K_d K_c E$, we have 3788 positive results and 308 negative results. The other tool agreed on all 3788 positive results, however it only agreed on 290 of the negative results, leaving 18 gossip graphs which the other tool could find a successful path for, but ours could not. This increases as the number of knowledge modalities increases; for the successful formula $K_b K_d K_c E$, the two tools agree on all 3676 positive results but disagree on 124 negative results[5].

Similar results were encountered when conjoining modalities. With the formula $K_a E \wedge K_b E$, our tool gave us 12 negative results out of 4096 that were not agreed with by the other tool. With the formula $\bigwedge_{i \in Ag} K_i E$ our tool gave us 213 negative results of 4096 that were not agreed with by the other tool.

There are several reasons for why this may be the case. The first one is that the equivalence relation used in **GithubGossip** has some nuances that changed it from being identical to the one used in this tool. These nuances can best be seen in Page 166 of **SMCDEL** This first caused problems when planning for formulae of the form $K_i \phi$ where $\phi \in \mathcal{L}_p(\Lambda)$[6]. Whilst the problems were fixed and all tests agree for these cases, it is possible that our fix is faulty for conjunctions and higher-order knowledge. Further work into the intricacies of knowledge could remedy this. It could also be an issue with the lifting of transducers, discussed in Section 2.5.4.

5.4 Profiling

One of the project aims was to perform a space and time analysis of our implemented system. We have a number of reasons to do this;

- Firstly, it lets us give a quantitative comparison of our program's efficacy compared to the existing tools.
- It also lets us find the parts of our code to try and optimise. It is said that a program spends 90% of its time in 10% of its code; a profiler helps us find this 10%.
- It also lets us find the weaknesses of our program in comparison to other tools, and as such find the weaknesses of the algorithm put forward.

In order to do this we used the profiling tool built into GHC. This gives us a detailed cost-centre breakdown of in which functions the majority of the time is spent, as well as the space used by these functions. We can see an example in Figure 5.2.

In the following section, we will give an analysis of the time and space used for several sizes of gossip graph and several different winning formulae, and compare them against the corresponding results in **GithubGossip** We use **GithubGossip** as it is the only tool capable of planning.

Note that the way that our tool and **GithubGossip** plan are quite different. In our tool we return just one successful sequence of events; in **GithubGossip** all possible sequences are returned. In the interests of fairly testing the two systems, we make an adjustment to **GithubGossip** to make it return the first valid call sequence it comes across. We perform comparisons with both versions.

5.4.1 Profiling Framework

In our profiling we will generate random graphs as in Section 5.2, use them in both tools and then take averages. This is because certain graphs will take much less time than others [7], and we want to give the most accurate

[3] Where first-order is $K_a \phi$, second-order is $K_b K_A \phi$, ...

[4] Note that it does not matter which agents knowledge we reason about; as we test for *each* graph, all cases are covered. The same goes for second- and higher-order knowledge; the only requirement is that we do not have $K_i K_i \phi$. Refer to footnote 2 on page 19 to be reminded why.

[5] A happy result is that when using the successful formula $K_a K_a \phi$, the two tools agree on all results. This is a testament to what we have talked about in footnote 2 on page 19; $K_a K_a \phi$ is exactly the same as $K_a \phi$. The observed result is a testament as it shows that $K_a K_a \phi$ is not in the same class as $K_a K_b \phi$.

[6] This fix can be seen in the `knowFilter` function in the code.

[7] Consider a graph where no one knows anyone elses phone number; both tools will very quickly decide that there is no successful path through this. A graph where there is a successful path will take much longer to compute the path, in comparison.

```
      Mon Apr 15 14:23 2019 Time and Allocation Profiling Report  (Final)

         Main +RTS -p -RTS

      total time  =        1.46 secs   (1457 ticks @ 1000 us, 1 processor)
      total alloc = 1,036,538,824 bytes  (excludes profiling overheads)

COST CENTRE            MODULE                  %time %alloc

models                 ME                       20.7    3.3
compare                Model                    14.3    0.0
compare                ME                        9.7   22.0
meTrans                ME                        9.2   11.5
enqueue                BFSM                      5.2   10.8
```

Figure 5.2: *An example cost centre output from GHC.*

statistics.

We then perform the planning process on the generated gossip graph. As mentioned earlier we run for some amount of graphs and then take the average.

5.4.2 Profiling Results

Gossip

When profiling, we generated gossip graphs of size 4 (by which we mean graphs with 4 agents in). This is not an arbitrary number; 4 is the smallest "interesting" gossip graph, meaning the smallest graph where a call occuring, not including some arbitrary agent a, has interesting indistinguishable calls; it could be from b to c, or b to d, or c to d [8]. Furthermore, graphs of size greater than 4 take an impractically long time - it is in the interests of time to use graphs of size 4.

When testing, the variable that we varied is the complexity of target formula. This might seem like an odd choice, but it is well-motivated. This will give us a way to investigate the differences in the representations of knowledge in the two tools. Furthermore, increasing the number of the agents in the models usually drastically increases runtime for both tools without giving us any new information.

We tabulate the results here. We abbreviate the statement "all agents are experts" to the propositional variable E. n is the number of agents in the initial gossip graph. All of figures in Table 5.1 and Table 5.2 are averages of many test cases.

	E	K_aE	K_bK_aE	$K_cK_bK_aE$
Our tool, $n = 3$	0.53ms, 0.23Mb	0.72ms, 0.3Mb	0.74ms, 0.38Mb	0.91ms, 0.49Mb
Other tool, $n = 3$	0.41ms, 0.56Mb	0.84ms, 0.93Mb	0.99ms, 1.4Mb	1.36ms, 1.96Mb
Our tool, $n = 4$	9.2ms, 7.36Mb	15ms, 10.15Mb	24ms, 14.12Mb	
Other tool, $n = 4$	145ms, 251Mb	724ms, 1250Mb	3849ms, 6425Mb	

Table 5.1: *Profiling Results between our tool and* **GithubGossip**

We can clearly see that our tool is much more time and space efficient than the existing tool. However this is a quite straightforward victory; the other tool enumerates *all* of the possible paths through a gossip graph (for a graph of size 4, there thousands of these), and then checks the success of each one. Compare this to our tool, which simply find the first, shortest, successful path it can take to reach some successful state.

Hence to get a more accurate comparison, we slightly modify the code of the existing tool to exit once it finds a single successful path. This means that it closer mimics our tool, and as such will let us get a better performance comparison between the two.

[8] Compare this to a gossip graph of size 3; a call not involving some agent a can only be from either b to c or c to b.

	E	K_aE	K_bK_aE	$K_cK_bK_aE$
Our tool, $n = 4$	9.2ms, 7.36Mb	15ms, 10.15Mb	24ms, 14.12Mb	96.5ms, 65.1Mb
Other tool, $n = 4$	0.6ms, 0.694Mb	1.5ms, 2.006Mb	4.2ms, 4.6104Mb	7.4ms, 11.077Mb
Our tool, $n = 5$	7390ms, 8173Mb	33100ms, 38489Mb	-	-
Other tool, $n = 5$	1.86ms, 28.1Mb	732ms, 1089Mb	2085ms, 3643Mb	6390ms, 11304Mb

Table 5.2: *Profiling Results between our tool and* **GithubGossip** *with modification*

We see that the other tool now outperforms ours, in both space and time efficiency. We can see that, when we have 4 agents in the gossip graphs, the times are relatively similar; however once we increase the number of agents the runtime and space consumption balloons. This is due to the extra cost of needing to reason about *8* more calls; this is nearly as many calls as we have in total with 4 agents. This large increase in time and space consumption occurs again in our analysis in Section 5.4.2. Although the other tool suffers from slow down when increasing the number of agents in our gossip graphs, its increase is much less than in our tool. **Add in the why**.

In light of these results, we must remember the benefits our tool provides over the other. The most obvious is the facility of our tool to plan for generic epistemic models and propositional event models. There are many optimisations that could be made to our system if we were specialising the system to only plan for gossip models; for example, the technique discussed in Section 5.5.4. These optimisations would perhaps bring our system's performance up to speed with **GithubGossip** The effect of these optimisations can be seen when comparing times taken to check whether a sequence is successful in **SMCDEL** to the times taken for the other tool to plan in 5.2. Although **SMCDEL** is a much more advanced tool, using symbolic model checking, it is outperformed by **GithubGossip** This is most likely due to the Gossip-specific optimisations used in **GithubGossip**

Another benefit our system exercises over **GithubGossip** is its capability to plan for systems permitting infinite sequences of events.

n Prisoners and a Lightbulb

As a testament to our tool working for other classes of epistemci models as well as gossip models, we implemented the "n-prisoners and a lightbulb" problem. This is a problem very similar to the gossip problem; we care about what the prisoners *know*, how their knowledge *changes* due to *observations*, and how their *actions* effect the state of the world.

The riddle is as follows. n prisoners are all together in the prison cafeteria. They are told they will all be put into isolation cells, and they will be brought out from the cells for interrogation one by one in a room containing a light with an on/off switch. The only way prisoners can communicate with each other is by toggling the light switch. There is no fixed order of interrogations, nor fixed interval between interrogations. When interrogated, a prisoner may either do nothing, toggle the light switch, or announce that all prisoners have been interrogated. If the prisoner is correct in announcing that all prisoners have been interrogated, they are all set free; but the announcement is incorrect they are all executed. Whilst in the dining room, can the prisoners agree on a protocol that will set them free?

The answer is of course yes. We take the simplest protocol listed in **Prisoners** In this, the n prisoners appoint one prisoner to be the counter[9]. All of the non-counting prisoners then follow the following protocol; the first time they enter the room when the light is off, they turn it on; on all other occasions, they do nothing. The counter follows a different protocol. The first $n - 2$ times the light is on when he enters the interrogation room, he turns it off. The next time he enters the room, he (correctly) announces that everybody has been interrogated.

This is not the most efficient protocol; however it is the simplest and the best suited to our setting, as it can be perfectly modelled as an epistemic model and a propositional event model. We leave out the details of the modelling; they are perfectly covered in **Prisoners** and our implementation is nearly identical to this. Notable in our implementation is that we only care about the knowledge of the agent who is the counter; hence we *do not* model the knowledge states of the other $n - 1$ prisoners.

The reason that we chose to use the n prisoners problem is that it's very easy to scale the problem's complexity up or down; we simply add more prisoners. Despite the fact that we only care about the knowledge state of one agent, increasing the number of prisoners yields an increase in events; and most importantly increases the number of events *indistinguishable* to the agent in question. This yields a large increase in the amount of states we have in

[9]This is the crux of the solution; the typical problem solver does not think about having different prisoners behave differently.

our automata. The effects of this can be seen in Figure **??**; our time and space complexity increase exponentially as we increase the number of prisoners.

Although through careful programming we could bring the size of the constants down in the graphs, it would take some rethinking of the algorithm to avoid the exponential increase that we suffer from here. **Maybe top this off better?**

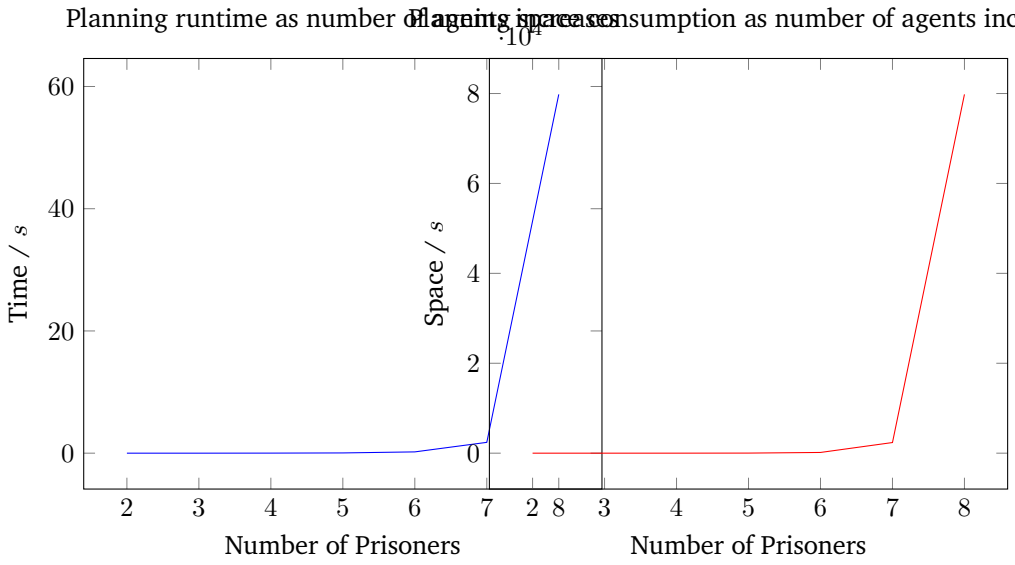


Figure 5.3: *Complexity analysis graphs.*

5.5 Results Analysis

5.5.1 `Models`

Through looking at the cost centres given to us by GHC, we can see where our program spends the bulk of the time computing. In our tool, the culprit is `models`, which can be seen in Figure 5.4. Unfortunately there's no more we can do to speed up this function [10], suggesting that the problem lies in the amount of times we call it. The function is one of the essential parts of the program; it lets us move from state to state.

The issue seems to arise from the final part of the definition of transitions in $\mathcal{ME}^*$. Recall that this is defined as:

```
models :: Prop p => Set.Set p -> Form p -> Bool
models _  Top          = True
models ps (Not form)   = not $ models ps form
models ps (P form)     = Set.member form ps
models ps (Or forms)   = any (models ps) forms
models ps (And forms)  = all (models ps) forms
```

Figure 5.4: *The `models` function*

$$\delta(q_v, e) = q_{v'}, \text{where } v' = \{p \in \Lambda \mid v \models \mathsf{post}(e, p)\} \text{ if } \models \mathsf{pre}(e) \tag{5.1}$$

where, given a world where a set of propositions v is true and an event e, we go through each proposition p in the set of propositions for the current model and check if the set of propositions v *model* the postcondition of the given proposition p and the event e. This occurs every time we wish to make a transition between two sets in our automata; clearly, this piece of code is being called a lot. However, this does not seem to be a particularly bad thing in itself; rather the opposite, it seems essential that this piece of code would be called a lot.

[10] Short of using a HashSet, which gives us quicker lookup times.

5.5.2 Search Strategies

To understand the differences in efficiency, we mainly need to consider the algorithmic differences. The other tool first enumerates the possible set of call sequences in a depth-first manner, after which it finds the first successful sequence of these calls. It does this latter task like a conventional model checker, and as such can use more powerful model checking techniques. In contrast, our tool performs a slightly different task; it builds a structure with which it can find paths through the gossip graph, and then finds the shortest.

The point at which this difference is really shown is in the generation of indistinguishable states. In our tool we progressively keep track of the states indistinguishable from our current one, and update each of these states with each transition, leading to a lot of time spent in the `models` function. In contrast, the other tool takes the string of calls to be checked against and generates all of the other strings of calls indistinguishable to it, and then checks if these model the target formula. The latter technique is much more computationally efficient than ours; the generation of these indistinguishable call sequences is much more straightforward and requires much less heavy lifting than maintaining this set of indistinguishable states.

One might ask why we don't copy this method of simply maintaining the set of strings of indistinguishable events which could have happened in the states of our automaton. This is a fine idea, however a problem arises when we come to considering the successful states. In the other tool it is known that the sequence of calls is "final", in the sense of a model checker - we want to know if this is successful or it isn't. But in ours we only stop either when we can make no more calls, or we are in a successful state. As such, we need to know if we are in a successful state *every time we enter one*, and the only way we can do this is by performing each string of indistinguishable calls on the initial state and evaluating the target formula at the resultant states; it is clear that this will be less efficient.

5.5.3 `BFSM`

Another method we spend a lot of time on is the BFSM method `enqueue`. In this we take the existing queue and append onto it the set of states that are next to be visited. This function uses a lot of memory and time as it has two memory-costly functions inside it; namely `filter` and `(++)`. This was an interesting function to investigate and try and optimise. We only need to perform two actions on our queue:

1. Read an item from the front
2. Append items onto the back

In Haskell, list concatenation has runtime $O(n)$, where n is the length of the list being appended to[11]. This is far from ideal - however the `Data.Seq` package offers us constant-time access to the first and last element of a sequence. This seems like it would be perfect for our situation, however **for some reason currently beyond me** the time nearly doubles and the space used balloons.

5.5.4 Monotonicity

Another thing we could have used to reduce computation is the monotonicity of the propositions true at a state. At a state in the gossip protocol, if some set v of propositions are true and we make some call e, then the set of propositions at the state we move to, v', is guaranteed to be a superset of v. Consider what this means; there is no phone call that can be made that will make anyone know any *less*. Hence we could change Equation 5.1 to

$$\delta(q_v, e) = q_{v'}, \text{where } v' = \{p \in \Lambda \setminus v \mid v \models \mathsf{post}(e, p)\} \cup v \text{ if } \models \mathsf{pre}(e) \tag{5.2}$$

This is because we know that the propositions in v will be true at the state $\delta(q_v, e)$; hence we do not need to check again whether or not they will be true.

However I fear this would be an irresponsible change to make to our code. Remember that we do not want our system to solely plan for the Gossip problem; rather we want it to be capable of doing so for *any* epistemic model and propositional event model. We have an example of a model where this occurs in Figure 2.3; although this is quite a toy example, it does indeed show that such cases are possible. In the interests of portability then, it would be unwise to make such a change.

[11] I.e., in the command `xs ++ ys`, n is the length of `xs`.

5.5.5 ANY and related protocols

Early on in this thesis we mentioned that **GithubGossip** is unable to plan for gossip graphs with protocols that allow for infinite call sequences (an example of this is ANY). One benefit that our tool demonstrates over **GithubGossip** is the ability to plan using such protocols. This is because before the process of searching through the gossip graph we crop the automata to remove any calls that do not have an effect. This prevents the problem that can arise when using ANY, where two agents can just repeatedly call each other over and over again.

This manifests itself as a great advantage over the existing tool. ANY is a very practical protocol for use in the gossip problem as it requires no reasoning on behalf of the agents; they do not need to consider whether or not they can make a call. **Check this this is quite bad**.

Chapter 6

Conclusion

We now take some time to reflect on the work done in this project. Our overarching aim in the project was to design an implement a system with which we can perform epistemic planning. This document's ordering reflects the order in which the challenges of the project were accomplished; we first reviewed the literature, then designed the algorithm, and then implemented it. Chapters 2, 3 and 4 are respectively dedicated to each subtask.

In Chapter 2, we explained the concepts necessary to understand the algorithm put forward. When starting work on this project the understanding of the processes put forward in the literature was the first major difficulty, and as such care has been taken in the writing process to make these definitions as clear as possible.

In Chapter 3, we put forward the algorithm designed to perform the task of planning. We hope that this communicates the magnitude of the wrok done here, as well as serves as a sufficient explanation for the process implemented. Whilst the existence of this algorithm was postulated in Aucher, Maubert, and Pinchinat 2014, it did not exist in a simple, digestable form; the work in Chapter 3 is the first explanation of the process in this manner. We hope that if someone were to continue work in this area, this document would aid there understanding.

Chapter 4 contains writing on implementation of the process. This details the design decisions made in the process, and argues a case for the use of Haskell in development of such a tool.

In Chapter 5 we gave an analysis - both quantitative and qualitative - of the algorithm developed and implementation. We compared our tool to an existing tool specialised for planning for gossip graphs, as well as investigating our tool's performance for the n prisoners problem.

6.1 Future Work

It is clear to see that our system is very slow; when run on a gossip graph with greater than 6 agents the takes so long that it's not worth waiting for the results. Whilst there is scope for implementation-based optimisations[1], this is mainly due to the algorithm; in Aucher, Maubert, and Pinchinat 2014 it is shown that the propositional epistemic planning problem is in $k+1$-EXPTIME, where k is the depth of the deepest-nested knowledge operator.

The method that we use is very explicit; we will always explore a branch, no matter what its state is. We may be wasting computation by either exploring a branch that is never going to succeed, or is identical to another branch. It would be interesting to see the implementation of some checker for recomputed branches, a la dynamic programming.

It also seems possible that there's scope for a symbolic approach, like in **MalvinThesis** The main issue with our system - and all model checking software as well - is the *state explosion* problem; we have to consider thousands of states, making the program slow and memory hungry. Symbolic approaches are used in many practical model checkers, **SMCDEL** being an example. The work done in this thesis on the production of an automata-based method for epistemic planning can act as a good foundation for this.

Epistemic preconditions are investigated in van Ditmarsch et al. 2016. These allow for protocols like "a may call b if a knows b's phone number and a considers it possible that b knows something that a does not know.". These protocols are clearly much more expressive than protocols like LNS. In our implementation, we encode preconditions in the pre function in the event models. However, recall that in this thesis we investigate the *propositional* fragment of the epistemic planning problem, in which our pre conditions may only be propositional;

[1] By which we mean optimisations like changing a `Set` to a `HashSet` and so on.

hence such epistemic protocols are not possible. Despite this, there is a sensation that it would be possible to use these protocols. We could evaluate the protocols on the states in question , as long as they are of the `PState` form. However, there might need to be significant restructuring of the algorithm to accommodate for this; currently protocols are restricted to just being propositional and as such are evaluated at the atomic, `PVar` states. Furthermore, it is proven in Aucher and Bolander 2013 that non-propositional preconditions can yield undecidability; hence we prepare ourselves for this to not be possible.

Bibliography

Aucher, Guillaume and Thomas Bolander (2013). “Undecidability in Epistemic Planning”. In: *IJCAI - International Joint Conference in Artificial Intelligence*.

Aucher, Guillaume, Bastien Maubert, and Sophie Pinchinat (2014). “Automata Techniques for Epistemic Protocol Synthesis”. In: *Proceedings of the 2nd International Workshop on Strategic Reasoning*.

Bolander, Thomas and Mikkel Birkegaard (2011). “Epistemic planning for single- and multi-agent systems”. In: *Journal of Applied Non-Classical Logics*.

Ghallab, Malik, Dana Nau, and Paolo Traverso (2004). *Automated Planning and Acting*. Cambridge University Press.

Haeupler, Bernhard et al. (2016). “Discovery through Gossip”. In: *Random Structures and Algorithms*.

Tijdeman, R. (1971). “On a telephone problem”. In: *Nieuw Archief voor Wiskunde (3)* 19, pp. 188 –192.

van Ditmarsch, Hans et al. (2016). “Epistemic Protocols for Dynamic Gossip”. In: *Journal of Applied Logic,*

— (2018). “Dynamic Gossip”. In: *Bulletin of the Iranian Mathematical Society*.

Yu, Quan, Ximing Wen, and Yongmei Liu (2013). “Multi-Agent Epistemic Explanatory Diagnosis via Reasoning about Actions”. In: *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*.