Title Most Definitely Pending

Anders Eide

Sometime in the future

Contents

1 Preface **3**

2 Introduction **4**

2.1 Motivation 4

2.2 Structure 5

3 Background **6**

3.1 Model checking 6

4 Theoretical work **8**

4.1 Introduction 8

4.2 Language and semantics 8

4.3 Bisimulation 12

5 Finite representation **15**

5.1 Enumeration of announcements in single agent cases 15

5.2 Generalizing the single agent case 17

5.3 Proof of suitability 18

5.4 Algorithm for bisimilarity check 19

5.5 Smallest bisimilar structure 21

5.6 Enumerating the set of announceable extensions 21

5.7 Complexity of the model checking problem 24

6 Implementation **25**

6.1 Visualization of Kripke structures 26

6.2 Checking formulas 27

6.3 Visualizing the checking process 29

6.4 Visualizing group announcements 31

6.5 Choice of platform 33

6.6 Language and interpretation 34

6.7 Model checking 35

6.8 Future work 38

7 Conclusion **40**

8 Appendix A. List of features **41**

1 Preface

Abstract:
In this paper we will present a graphical model checking tool for group announcement logic called GALMC, capable of visualizing the process of checking formulas in a step-by-step fashion. We will define how to enumerate the set of ways a given coalition can restrict a model as well as present pseudocode algorithms describing how we have translated our definitions in our model checker.

2 Introduction

Abstract

In this paper we will present a graphical model checking tool for group announcement logic called GALMC, capable of visualizing the process of checking formulas in a step-by-step fashion. We will define how to enumerate the set of ways a given coalition can restrict a model as well as present pseudocode algorithms describing how we have translated our definitions in our model checker.

2.1 Motivation

(ASK TRULS: Greit med førsteperson i motivasjon?)
(TODO: Skriv om hele introduksjonen av GAL)
(TODO: Hente ned bilder av JFLAP i aksjon, med illustrasjoner av hvordan verktøyet og stepperen i det fungerer)

(TODO: Write sub-section on inspiration for tool about JFLAP, educational benefits ect)

(ASK TRULS: Footnotes i en masteroppgave? Lenke til nettsiden for JFLAP)

In 2010, Ågotnes et al. submitted a paper on their extension of Public Announcement Logic called Group Announcement Logic [?]. Having previously used other visual tools such as JFLAP[1] to aid student learning while teaching courses and seeing how helpful it can be to visualize the way something works, I wanted to create something similar that could aid students in learning how group announcement logic works. Looking at other branches of logic such as Computational Tree Logic it is also clear that the field of model checking epistemic logics such as GAL lags far behind. While there does exist model checkers for similar kinds of logic such as DEMO[2] for Public Announcement Logic, in my opinion DEMO would not lend itself very well to teaching as it requires the user to structure their models as plain sets of numbers and letters through the command line rather than let them see and build their models in a more easily understood manner. As such we not only wanted to make a model checker that supports group announcement logic, but also make it useful in an educational context, by making able to present how the various operators in the language work in a more visual manner. There is also an interesting theoretical challenge involved in model checking GAL, as the semantics behind the logic's group announcement operator involve quantifying over an infinite set of formulas that a given coalition can

[1] Available from: `http://www.jflap.org/`

[2] Haskell source files and user guides available from: `https://homepages.cwi.nl/~jve/software/demo_s5/`

announce. To this end, we will come up with an algorithm for enumerating this infinite set of formulas by exploring the properties of minimal bisimilar structures and grouping these formulas by which states in our structures they are satisfied in.

(TODO: Nevne modellsjekker og læringsfordeler ved visualisering)
(TODO: Nevne APAL vs GAL, ulike måter å løse lignende 'problem', APAL tar ikke hensyn til agentenes kunnskap)
(TODO: Nevne interessante egenskaper ved GAL, de re/de dicto)

2.2 Structure

This paper is divided into the following sections:

A background section containing background information regarding the field of dynamic epistemic logic and Group Announcement Logic as well as a lot of the various definitions that will be used in the following sections.

A section containing the theoretical work presented in this paper, where we discuss the semantics of group announcement logic and will explore their definitions in order to find definitions that are more easily translatable into algorithms we can use in our model checker.

A section where we translate these definitions into algorithms presented through pseudocode and explain and discuss any differences from the logical definitions.

And finally a section where we present and discuss our implementation of these algorithms and the working software they are implemented in.

(TODO:
Discuss different approaches towards coalitional ability in dynamic epistemic logic, mention coalitional logics like ATL?
Discuss educational benefits from usage of model checker, visualization of semantics behind operators
Discuss scope of implementation, not suitable for research work, but potentially useful in educational setting)

3 Background

In this chapter we will give the reader some insight into the background of these systems of logic while determining the state of the art, before introducing various concepts and terms that will be used later on in this thesis.

The logic this thesis will be working with, group announcement logic, is also one of these extensions of epistemic logic, but in addition to reasoning around information change also allows us to reason around what coalitions of agents are able to achieve through making public announcements in unison based on how they can change the information available to other agents in the system. Group announcement logic, like many other forms of epistemic logic revolve around the notion of so-called ‘possible worlds’ and agents which may or may not be able to distinguish between them. When working with these possible worlds, we usually refer to them as states in some system we are trying to simulate. We describe which agents are able to distinguish between these states based on an accessibility relation for each agent, consisting of pairs of states they are unable to differentiate. Since we are solely working with epistemic models in this thesis, also known as S5 models, these accessibility relations will also be equivalence relations (meaning they are reflexive, symmetric and transitive), and we will from here on also refer to them as such.

These simulations are then grouped into models consisting of a set of possible worlds or states, a set of equivalence relations for each agent we wish to model, as well as a set of boolean propositions which may or may not hold in a given state. Based on this set of possible worlds and these accessibility relations between them we can model what each agent in our system knows by defining knowledge as being

3.1 Model checking

Model checking refers to the act of checking whether or not a formula holds in a given model. A simple example of model checking might be to check if its always true that Alice knows whether or not it is raining outside in the context of some model. A more interesting example could be to check whether or not there exists a sequence of actions that can be taken which might make a distributed system deadlock. Model checking is a general problem not just within knowledge representation and artificial intelligence, but also in other fields of computer science such as formal verification of communication protocols and software correctness.

With model checking utilities being such useful tools for exploring the properties of more complex models, it should come as no wonder that there exists plenty of model checkers out there for various forms of logic. Most traditional model checkers however focus on answering simple yes or no questions in regards to whether something is true or not. One of the closest examples of this is DEMO_S5 for dynamic epistemic logic (DEL), being a command-line application that lets the user types in their formulas and check it against models formatted as long strings of text. We aim to create a tool which can assist users in understanding how these systems work by providing a graphical editor that can visualize both the models themselves as well as the checking process.

4 Theoretical work

In this section we will present our theoretical work on exploring the group announcement operator in GAL. In doing this, we will be building on the definitions that were established and introduced in the previous section.

4.1 Introduction

Throughout the rest of this thesis, we will refer to the previous by the following definition:

Definition 4.1 (Models) *Given a group of agents A, and a set of propositions P, a model M is a structure $M = (S, \sim, V)$, where*

- *S is a set of states*
- *$\sim$ is a function from every agent $a \in A$ to a's equivalence relation $\sim_a \subseteq S \times S$*
- *$V(p)$ is a valuation function that for every proposition $p \in P$ returns the set of states where p is true*

We will also frequently use $\sim$ as $s \sim_a t$, where $a \in A$ to signal that agent a is unable to distinguish between states s and t. When we wish to highlight specific states in a model we will denote this by writing (M, s) to specify that we are referring to state s in the model M, commonly known as a pointed model. This is the form that we will use when we are discussing the properties of a state s in the context of some model M.

A simple practical example of an epistemic model could be us imagining someone that doesn't know whether or not it is raining outside. If we focus solely on whether or not it is in fact raining outside, we might denote this proposition as p, where $p =$"It is raining outside". Since this agent, a does not know whether it is raining, this means that in their mind there must be at least two possible worlds, or states; one where p is true, and one where p is false and that a is unable to tell these two states apart. If we label these states as s_0 and s_1, we end up with the following model shown in Figure 4.1, where we use $\neg p$ to signify that p is not true in (M, s_1) (TODO: Fix this reference so it points to the correct figure)

4.2 Language and semantics

Continuing with the model shown in Figure 4.1, we might want to express certain properties or verify that p is indeed true in (M, s_0), for this we would write $(M, s_0) \models p$, or that s_0 in the context of M *satisfies* p. More formally, $(M, s_0) \models p$ holds *iff* $s_0 \in V(p)$ where if we go back to Definition 4.1, we can see that V is a valuation function for propositions, meaning that it returns the set of states in M, where p holds.

Figure 4.1: A basic model with two states

$$\begin{tikzcd}\underset{p}{s_0} \arrow[r, dash, "a"] & \underset{\neg p}{s_1}\end{tikzcd}$$

Note that since we are working with S5 models, all nodes (states) also have reflexive edges to themselves, even if they are not explicitly drawn in these figures

In general however, we might want to build more complex formulas to check against our models than simple propositions, for this we need connectives; and logics that define their semantics.

Definition 4.2 (Group Announcement Logic)

$$\varphi ::= p \mid \neg\varphi \mid (\varphi \ \wedge \ \psi) \mid (\varphi \ \vee \ \psi) \mid \varphi \to \psi \mid K_a\varphi \mid [\varphi]\psi \mid [G]\varphi$$

where p *is a proposition such that* $p \in P$, a *is an agent (i.e.* $a \in A$*) and* G *is a set of agents (i.e.* $G \subseteq A$*).*

Definition 4.2 defines the language of $\mathcal{L}_{GAL}$ or more specifically, the set of legal formulas that can be built in accordance to GAL. It inductively specifies in BNF-notation how each operator in the language can be used to create new formulas. This notation however merely specifies the *syntax* of the language, which ways we are allowed to arrange the symbols. In order to specify what each operator does, we also need semantics, which we will add through the satisfaction relation $\models$.

Definition 4.3 (GAL Semantics)

$\mathcal{M}, s \models p$ *iff* $s \in V(p)$

$\mathcal{M}, s \models \neg\varphi$ *iff* $\mathcal{M}, s \not\models \varphi$

$\mathcal{M}, s \models \varphi \wedge \psi$ *iff* $\mathcal{M}, s \models \varphi$ *and* $\mathcal{M}, s \models \psi$

$\mathcal{M}, s \models \varphi \vee \psi$ *iff* $\mathcal{M}, s \models \varphi$ *or* $\mathcal{M}, s \models \psi$

$\mathcal{M}, s \models \varphi \to \psi$ *iff* $\mathcal{M}, s \not\models \varphi$ *or* $\mathcal{M}, s \models \psi$

$\mathcal{M}, s \models K_i\varphi$ *iff for every* t *such that* $s \sim_i t$, $\mathcal{M}, s \models \varphi$

$\mathcal{M}, s \models [\varphi]\psi$ *iff* $\mathcal{M}, s \models \varphi$ *implies that* $\mathcal{M}|\varphi, s \models \psi$

$\mathcal{M}, s \models [G]\varphi$ *iff for every set* $\{\psi_i : i \in G\} \subseteq \mathcal{L}_{el}$, $\mathcal{M}, s \models [\bigwedge_{i \in G} K_i\psi_i]\varphi$

(TODO: Mention tautologies and falsehoods, plus valid and satisfiable formulas ect)

Starting with the most basic component of our formulas we have propositions, commonly referred to as atoms or atomic formulas as propositions themselves are also formulas. If we want to check if our pointed model $\mathcal{M}, s$ satisfies a property p (denoted by $M, s \models p$), then as we can see in Definition 4.3 we do this by seeing if the state s is an element in the set of states where p is true, more formally if $s \in V(p)$. We denote the fact that the pointed model M, s does not satisfy formula φ by $M, s \not\models \varphi$.

For the explanation of our operators we will be using φ and ψ to represent arbitrary formulas. Negation $\neg$, is relatively trivial, with $\neg\varphi$ simply meaning the negation of φ and that $\mathcal{M}, s \models \neg\varphi$ if, and only if $\mathcal{M}, s \not\models \varphi$. Conjunction and disjunction $\wedge$ and $\vee$, are also pretty simple, translating to 'and' and (inclusive) 'or', as can be seen from their definitions in Definition 4.3. Implication, $\rightarrow$ can be somewhat loosely translated to 'if a, then b', except if a doesn't hold, then the implication automatically holds regardless of b. More specifically, the only time an implication is false is when a is true and b is false. We will also use $\varphi \leftrightarrow \psi$, to denote biimplications, meaning $(\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi)$ throughout this thesis.

The previous operators are all relatively basic and only express properties regarding the structures they are checked against, the remaining operators in our language are more interesting, as they also express properties regarding the knowledge of the agents in our systems shifting us towards epistemic logic. The K_i operator is the first of these, with $\mathcal{M}, s \models K_a p$ expressing that in our pointed model agent a knows that the proposition p is true. Verifying that this is the case however is where things get interesting, as the previous operators only express properties regarding single states, their scope is also limited to that single state, whereas the K_i operator also requires us to check all the states that this agent is incapable of distinguishing from the current state. In order to check if $\mathcal{M}, s \models K_a\varphi$, we need to ascertain that all states a is unable to distinguish from s also satisfy φ. More specifically, that for all states $t \sim_a s$, that $\mathcal{M}, t \models \varphi$. This is because if there exists any such state that does not satisfy φ, then agent a considers it possible that φ is not satisfied since they cannot tell if t or s is the actual state they are in.

While the K_i operator lets us reason about the knowledge of our agents, the public announcement operator $[\varphi]\psi$ lets us update it, or at least add to it by informing the agents through a truthful announcement that some formula is true in the current state as defined by the pointed model we are checking the announcement in. In Definition 4.3 we use the notation $\mathcal{M}|\varphi, s$ to denote $\mathcal{M}$ 'updated' by φ, meaning $\mathcal{M}$ minus all states which do not satisfy φ.

Definition 4.4 (Model Updates) $\mathcal{M} = \langle S, \sim, V \rangle$ *updated by* φ *is defined* *as the following:* $\mathcal{M}|_\varphi = \langle S', \sim', V' \rangle$ *where*

$$S' = \{s | s \in S \textit{ and } \mathcal{M}, s \models \varphi\}$$
$$\sim'_a = \sim_a \cap (S' \times S') \; \forall a \in A$$
$$V'(p) = V(p) \cap S' \; \forall p \in P$$

Informally, Definition 4.4 can be read as filtering out all states in $\mathcal{M}$ which do not satisfy φ and then constraining the equivalence relations for each agent and the valuation function to that filtered set of states.

The final and most interesting operator in $\mathcal{L}_{GAL}$ is $[G]\varphi$, or the group announcement operator, where G is some coalition of agents such that $G \subseteq A$. $\mathcal{M}, s \models [G]\varphi$ expresses that there is no way for the group G to announce anything that can make φ false in $\mathcal{M}, s$. Note however that the formulas the agents in $[G]$ can announce are constrained to basic epistemic formulas, or to $\mathcal{L}_{el}$ as defined in Definition 4.5 in order to avoid making our definition of group announcements circular. The definition of satisfaction of the group announcement connective reduces to a quantified public announcement (note that this is a different connective with only superficial similarity). Is is defined as the statement that the conjuntion of all sets of formulas $\{\varphi_i : i \in G\}$ may be announced without making φ true in the current state. More intuitively, that G is unable to make φ come about.

The syntax of $\{\psi_i : i \in G\}[\bigwedge_{i \in G} K_i\psi_i]\varphi$ intuitively means 'the conjunction of all formulas $K_i\psi_i$ such that each agent i knows their formula ψ_i and that after this conjunction is announced, φ is true'. As the definition in Definition 4.3 specifies that this has to hold for *every* set of $\{\psi_i : i \in G\}$ however, it might be easier to think of it as there not existing a set such that $\mathcal{M}, s \not\models [\bigwedge_{i \in G} K_i\psi_i]\varphi$ or that G does *not* have the ability to prevent φ from coming true.

Definition 4.5 (Language of epistemic logic $\mathcal{L}_{el}$)

$\mathcal{M}, s \models p$ *iff* $s \in V(p)$

$\mathcal{M}, s \models \neg\varphi$ *iff* $\mathcal{M}, s \not\models \varphi$

$\mathcal{M}, s \models \varphi \wedge \psi$ *iff* $\mathcal{M}, s \models \varphi$ *and* $\mathcal{M}, s \models \psi$

$\mathcal{M}, s \models \varphi \vee \psi$ *iff* $\mathcal{M}, s \models \varphi$ *or* $\mathcal{M}, s \models \psi$

$\mathcal{M}, s \models \varphi \rightarrow \psi$ *iff* $\mathcal{M}, s \not\models \varphi$ *or* $\mathcal{M}, s \models \psi$

$\mathcal{M}, s \models K_i\varphi$ *iff for every* t *such that* $s \sim_i t$, $\mathcal{M}, s \models \varphi$

where all operators have their semantics defined as in Definition 4.3.

We will also be using the duality of the two announcement operators, $\langle\varphi\rangle\psi$ and $\langle G\rangle\varphi$ which are defined as follows in Definition 4.6.

Definition 4.6 (Dual announcement operators)

$\mathcal{M}, s \models \langle\varphi\rangle\psi$ *iff* $\mathcal{M}, s \models \varphi$ *and* $\mathcal{M}|_\varphi, s \models \psi$
$\mathcal{M}, s \models \langle G\rangle\varphi$ *iff there exists a set* $\{\psi_i : i \in G\} \subseteq \mathcal{L}_{el}$ *such that:*

$$\mathcal{M}, s \models \langle \bigwedge_{i\in G} K_i\psi_i\rangle\varphi$$

The difference between $[\varphi]\psi$ and $\langle\varphi\rangle\psi$ is that the box notation 'implies' that ψ is true in the updated model, while the diamond notation requires both that $\mathcal{M}, s \models \varphi$ and $M|_\varphi \models \psi$. For group announcements, the box notation expresses that something has to hold after any possible set of announcements the coalition can make, while the dual $\langle G\rangle$ expresses that there exists at least one set of announcements for the agents in G such that after their announcements φ is true. An interesting observation to make is that like in modal logics we have the same relation between $[\psi]\varphi$ and $\langle\psi\rangle\varphi$ as $\Box\varphi$ and $\Diamond\varphi$, namely that $[\psi]\varphi \leftrightarrow \neg\langle\psi\rangle\neg\varphi$ and $[G]\varphi \leftrightarrow \neg\langle G\rangle\neg\varphi$.

When we are discussing announcement operators we are usually also interested in the knowledge of the agents in our system, as such it is useful to be able to refer to the set of states an agent considers indistinguishable from the given state. For this we will be using the notion of equivalence classes, defined as the following.

Definition 4.7 (Equivalence classes) *Given a state s and some agent a, a's equivalence class for s is denoted by $[\![s]\!]_a$ where*

$$[\![s]\!]_a = \{t \mid s \sim_a t\}$$

Somewhat related to equivalence classes, we will also be using the concept of formula extensions, referring to the set of states a formula is satisfied in, denoted by $||\varphi||$, but also to refer to a set of formulas with the same extension, i.e are satisfied in the same states.

Definition 4.8 (Formula extensions) *For some formula φ and some model $\mathcal{M}$, the extension of φ in $\mathcal{M}$ is denoted as $||\varphi||_\mathcal{M}$ where*

$$||\varphi||_\mathcal{M} = \{s \in S \mid \mathcal{M}, s \models \varphi\}$$

4.3 Bisimulation

(ASK TRULS: Hvordan best skille mellom bisimilære tilstander og modeller)

Sometimes we may want to express that two models are 'the same', i.e. they satisfy exactly the same set of formulas, despite possibly being structurally different. For this, we have the notion of bisimilarity, denoted by $M \leftrightarrows M'$. As the concept of bisimilarity is quite central to our exploration of the semantics of the group announcement operator, we introduce the definition of bisimilarity as follows in Definition 4.9

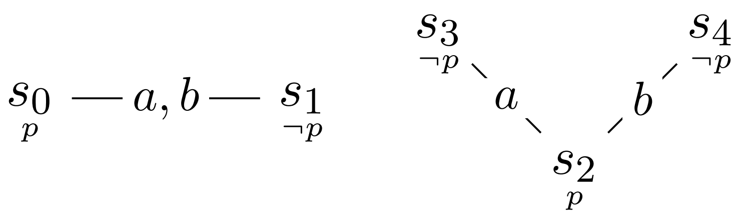
Definition 4.9 (Bisimulation) *Given two models $M = (S, \sim, V)$ and $M' = (S', \sim', V')$, a non-empty relation $\mathfrak{R} \subseteq S \times S'$ is a bisimulation between M and M' iff for all $s \in S$ with $(s, s') \in \mathfrak{R}$:*

atoms *for all $p \in P$: $s \in V(p)$ iff $s' \in V'(p)$;*

forth *for all $a \in A$ and all $t \in S$: if $s \sim_a t$, then there exists a $t' \in S'$ such that $s' \sim'_a t'$ and $(t, t') \in \mathfrak{R}$;*

back *for all $a \in A$ and all $t' \in S'$: if $s' \sim'_a t'$, then there exists a $t \in S$ such that $s \sim_a t$ and $(t', t) \in \mathfrak{R}$;*

Figure 4.2: Two bisimilar, but structurally different models



Building on the concept of bisimilar states, the bisimulation contraction of a model $\mathcal{M}$ is the smallest bisimilar structure to $\mathcal{M}$, obtained by merging each set of bisimilar states in $\mathcal{M}$ into a single state.

(TODO: Update description of algorithms relating to bisimulation contracting after finishing this clusterfuck)

Definition 4.10 (Bisimulation contracted models) *A model M is said to be bisimulation contracted iff there is no model M' which is smaller and bisimilar to $\mathcal{M}$. A bisimulation contracted model of a model M, is any model M' which is bisimulation contracted and bisimilar to M. Given a model $\mathcal{M}$, one of it's smallest bisimilar structures $\mathcal{M}'$ is*

The reason why this concept is interesting is that since these bisimulation contracted models do not contain any bisimilar states, this means that per the definition of bisimilarity, there has to exist some set of formulas that uniquely identify each state in our model by being satisfied only in that specific state of our model. We will refer to these as labeling formulas.

Definition 4.11 (Labeling formulas) *Given a bisimulation contracted model $\mathcal{M}$, there exists at least one formula φ_s for every $s \in S$ such that $\mathcal{M}, t \models \varphi_s$ iff $s = t$. More precisely in terms of formula extensions:*
$\forall s \in S,\ \exists \varphi_s$ such that $||\varphi_s|| = \{s\}$.

Additionally, we will also be referring to the set of labeling formulas for a given bisimulation contracted model $\mathcal{M}$ as $F_{\mathcal{M}}$, defined as follows.

Definition 4.12 (Set of labeling formulas) *Given a bisimulation contracted model $\mathcal{M}$, the set of formulas uniquely identifying each state in the model is defined as* $F_{\mathcal{M}} = \{\varphi_s | s \in S\}$

5 Finite representation

In this section we will describe the process of translating the definitions from the previous chapter into algorithms that will be used in our model checker.

Going back to our revised semantics behind the group announcement operators in Definition 5.6 we can see that once we can determine how to enumerate the set of announceable extensions $\mathcal{A}_G$, implementing the semantics behind the group announcement operator is fairly straightforward. However, as our definition of $\mathcal{A}_G$ uses the concept of bisimulation contracted models, we will first need to cover how we can check whether states are bisimilar and then apply that check to models as a whole in order to reduce them to their smallest bisimilar structures.

5.1 Enumeration of announcements in single agent cases

Recall the definition of the semantics behind $\mathcal{M}, s \models \langle G \rangle \varphi$ in Definition 4.6, an informal way of explaining it would be that there exists at least one set of formulas the coalition can announce such that φ is true after their announcements.

Using our definitions of labeling formulas and formula extensions in Definition 4.8 and 4.11 to look at what kinds of formulas any given agent can announce, we note that the goal is to convey information to the other agents in the system. Therefore we do not need to look at the formulas themselves, but only at what consequence announcing them would have and whether or not the agent is able to announce them. For this reason we will be using the concept of formula extensions from Definition 4.8 instead of concrete formulas as announcing any given formula will eliminate all states not in that formula's extension from the updated model as defined by the semantics of model updates.

Figure 5.1: A basic bisimulation contracted model.

$$\begin{tikzcd}
\underset{p,\, q}{s_1} \arrow[r, dash, "a"] \arrow[d, dash, "b"] & \underset{p,\, \neg q}{s_2} \arrow[d, dash, "b"] \\
\underset{\neg p,\, q}{s_3} \arrow[r, dash, "a"] & \underset{\neg p,\, \neg q}{s_4}
\end{tikzcd}$$

If we attempt to determine which sets of formulas some agent i can announce in the model in Figure 5.1 we can see that the set of announceable formula extensions will always be a subset of the power set of states in the model. Or more precisely, the set of announceable formula extensions for some agent i in a given pointed model $\mathcal{M}, s$ is a subset of the power set of states in $\mathcal{M}$, where each set is announceable by i if and only if that set follows the following rules in Definition 5.1.

Definition 5.1 (Rules for eliminating formulas by their extension) *For any formula φ and bisimulation contracted pointed model $\mathcal{M}, s$, the formula's extension in $\mathcal{M}$, $||\varphi||_{\mathcal{M}}$, must satisfy the following rules in order for the formula to be announceable by some agent i in coalition G:*

- $||\varphi||_{\mathcal{M}}$ *must contain the actual state in our pointed model*
- $\forall s \in ||\varphi||_{\mathcal{M}}, \forall t \sim_i s \Rightarrow t \in ||\varphi||_{\mathcal{M}}$

The reasoning behind these two rules is based on the semantics of the group announcement operators in Definition 4.3, we can see that we are essentially searching for a combination of formulas which when announced, make φ false. Because of this, having an agent announce that they know something which is false, or something they do not actually know, will simply make the public announcement trivially true. This means there is no point in checking any announcement containing such a formula, therefore the formula:

(1) has to be satisfied in the 'actual' state of our pointed model

(2) has to be satisfied in every state the agent is incapable of distinguishing from that 'actual' state

Proposition 5.2 *For every extension $||\varphi||_{\mathcal{M}}$ that satisfies the rules in Definition 5.1 for some agent, any formula with the same extension can be announced by that agent*

From our definition of formula extensions in Definition 4.8, the extension of a formula is simply the set of states in which this formula is satisfied. Therefore if φ and ψ share the same extension in some model $\mathcal{M}$, then $\mathcal{M} \models \varphi \leftrightarrow \psi$. From this we can further infer that $\mathcal{M} \models K_a\varphi \leftrightarrow K_a\psi$ for every ψ with the same extension as φ

Definition 5.3 (The set of announceable extensions) *The set of announceable formula extensions $\mathcal{A}$ for some agent i, given a pointed model $\mathcal{M}, s$ is defined as the following:*
$\mathcal{A}_{i,(\mathcal{M},s)} \subseteq \wp(\{||\varphi||_{\mathcal{M}} \mid \varphi \in F_{\mathcal{M}}\})$ where $||\varphi||_{\mathcal{M}} \in \mathcal{A}_{i,(\mathcal{M},s)}$ iff it follows the rules in Definition 5.1. Note that since every formula in $F_{\mathcal{M}}$ is only satisfied in a single state in S and every state in S satisfies exactly one formula in F_M, $\wp(\{||\varphi||_{\mathcal{M}} \mid \varphi \in F_{\mathcal{M}}\})$ can be simplified to $\wp(S)$.

For a more practical explanation we will apply these rules to the model in Figure 5.1 for agent a. Using s_1, denoted 1 in the next example, as the actual state of our pointed model, we start by generating the power set of states in our model and get the following:

$$\wp(S) = \{\emptyset, \{1\}\{2\}, \{3\}, \{4\}, \{1,2\}, ..., \{1,2,3\}, ...\{1,2,3,4\}\}$$

After applying rule (1) from Definition 5.1 to filter out all formula extensions not containing the actual state of our pointed model we get:

$$\{\{1\}, \{1,2\}, \{1,3\}, \{1,4\}, \{1,2,3\}, \{1,2,4\}, \{1,3,4\}, \{1,2,3,4\}\}$$

Applying rule (2) to remove all extensions relating to formulas that a does not know leaves us with:

$$\{\{1,2\}, \{1,2,3,4\}\}$$

In other words, given the model in Figure 5.1 agent a is able to announce any given formula φ iff $||\varphi||_{\mathcal{M},1} \in \{\{1,2\}, \{1,2,3,4\}\}$. Therefore, $\mathcal{A}_{a,(\mathcal{M},1)} = \{\{1,2\}, \{1,2,3,4\}\}$.

An interesting thing to note here is that we can combine our labeling formulas with disjunctions to create a formula which is satisfied in any subset of the original set of states we want. So for example, in order to create a formula with the extension of $\{s_1, s_2\}$ all we have to do is put disjunctions between the labeling formulas for s_1 and s_2 such that $\varphi_{\{s_1,s_2\}} = \varphi_{s_1} \vee \varphi_{s_2}$.

If we then want to apply this to checking whether $\mathcal{M}, s_1 \models \langle a \rangle K_b p$ where $\mathcal{M}$ is still the model in Figure 5.1 then we can see that since $\{s_1, s_2\} \in \mathcal{A}_{a,(\mathcal{M},s_1)}$ there exists a formula extension that a can announce which would eliminate s_3 from the model. This would cause $K_b p$ to be satisfied in the updated model and therefore satisfy $\langle a \rangle K_b p$.

5.2 Generalizing the single agent case

Expanding what we have presented so far to encompass coalitions comprised of multiple agents is actually very easy. When we are assessing the ability of an agent to announce something which may change the valuation of a formula, we are simply checking whether or not that agent is capable of eliminating certain states in the updated model. In other words, if we wish to assess the ability of a whole coalition, all we need to do is look at which sets of states each agent in the coalition is able to eliminate in unison.

An interesting observation to make is that an agent i is always capable of eliminating any state they can distinguish from the actual state of some pointed model. This means that in order to find out which states a coalition can eliminate, we can simply take the power set of states and limit it to the combinations of states which fit the rules of Definition 5.1, where we slightly tweak rule 2 to the following:

Definition 5.4 (The set of extensions announceable by coalitions) *The set of announceable extensions $\mathcal{A}_{G,(\mathcal{M},s)}$ for some coalition G, given a bisimulation contracted pointed model $\mathcal{M}, s$ is defined as the following:* $\mathcal{A}_{G,(\mathcal{M},s)} = \{S' \subseteq S \mid \forall s' \in S', \neg\exists t \forall i \in G\ t \sim_i s', t \notin S'$ *and* $s \in S'\}$

A candidate set $S' \subseteq S$ not satisfying the condition is clearly not announceable because it contains a state s which no agent in the coalition can distinguish from the excluded state t.

5.3 Proof of suitability

(TODO: reword intro)

In this section we will compare our definitions and work so far with the definitions presented by Ågotnes et al. in their paper on GAL. In their paper they describe how to check formulas of the kind $\langle G \rangle \varphi$ in the following manner:

Definition 5.5 (Definition of $\langle G \rangle \varphi$ by Ågotnes et al.) $\mathcal{M}, s \models \langle G \rangle$ *iff there is a definable restriction* $\mathcal{M}'' = (S'', \sim'', V')$ *of* $\mathcal{M}$ *such that* $S'' = \cap_{i \in G} C_i$ *where* C_i *are unions of classes of equivalence for* $\sim_i$ *and* $s \in S''$ *and* $\mathcal{M}', s \models \varphi$

Decomposing their definition we end up with S'' being the intersection of the unions of some subset of each agent's equivalence relation. More specifically, for each agent i, we choose which equivalence classes should be part of that agent's union of equivalence classes C_i and then check if there exists some combination of values for each C_i such that restricting the set of states to the intersection of these C_is gives us a model which both contains the original state s and satisfies φ.

Comparing this definition to our definition of the announceable set of extensions for a coalition, we argue that our definition of $\mathcal{A}_{G,(\mathcal{M},s)}$ defines exactly these intersections of possible combinations of C_is from the definition of Ågotnes et al. If we further decompose their definition, we end up with the two following restrictions:

$$S'' = \bigcap_{i \in G} C_i \text{ where } C_i \subseteq S, \forall s \in C_i, [\![s]\!]_i \subseteq C_i \quad (1)$$

$$s \in S'' \quad (2)$$

From this, we argue that our definition of $\mathcal{A}_{G,(\mathcal{M},s)}$ incorporates the same two restrictions, in turn quantifying the set of all possible restricted sets S''. Decomposing our definition the same way we did Definition 5.5, we end up with $\mathcal{A}_{G,(\mathcal{M},s)}$ defined as the set of all subsets of S, S' which satisfy the following two restrictions:

$$\forall s' \in S' \neg\exists t \forall i \in G, t \sim_i s', t \notin S' \quad (3)$$

$$s \in S' \quad (4)$$

Expanding on our restriction in (3), we could write it out as 'if there exists a state t indistinguishable by all agents from s, then either $t \in S'$ or $s \notin S'$'. A simpler way of phrasing this would be to say that for every state in S', the intersection of its equivalence classes for all agents in the coalition has to be a subset of S'. Changing (3) to fit this simpler phrasing gives us

$$\forall s' \in S' \cap_{i \in G} [\![s']\!]_i \subseteq S' \tag{5}$$

which should further clarify that combining restriction (4) and (5) provides the same set of possible values as (1) and (2). Based on this, we revise the semantics for the group announcement operators into the following:

Definition 5.6 (Group announcement operators, revised)

$$\mathcal{M}, s \models [G]\varphi \textit{ iff } \forall Ext \in \mathcal{A}_{G,(\mathcal{M},s)} \textit{ then } \mathcal{M}, s \models [\varphi_{Ext}]\varphi$$
$$\mathcal{M}, s \models \langle G\rangle\varphi \textit{ iff } \exists Ext \in \mathcal{A}_{G,(\mathcal{M},s)} \textit{ such that } \mathcal{M}, s \models \langle\varphi_{Ext}\rangle\varphi$$

where φ_{Ext} *is a formula of the form* $\bigvee_{s \in Ext} \varphi_s$

Our reason for revising these definitions is that the original definitions in 4.3 define satisfaction of $[G]\varphi$ by using a quantifier over an infinite set of formulas making it unfit for our goals of implementing a model checking tool. We will therefore be implementing the revised definition in 5.6 instead as the set of announceable extensions is far easier to enumerate than the set of announceable formulas. This is still equivalent to the original definition as we are simply grouping the infinite set of announceable formulas into a finite set of announceable extensions.

5.4 Algorithm for bisimilarity check

Using the definition of bisimilarity in Definition 4.9 we present our recursive function for checking bisimilarity between states. Given a set of states $States$, a set of agents $Agents$ and two states s and s' such that $s, s' \in States$, the function (recursive bisimulation check) $rbc(s, s', States, Agents)$ determines whether s and s' are bisimilar is defined as Algorithm 1.

The resulting algorithm is fairly similar to the logical definition, with the *props* check being equivalent to the *atoms* clause and the $knowledgeCheck$ function replacing the $forth$ and $back$ clauses. Note that our valuation function is flipped, going from a state to a set of propositions, rather than a proposition to a set of states. The pseudocode for the $knowledgeCheck$ function used to translate $forth$ and $back$ is shown in Algorithm 2.

Comparing our algorithm to the original definition of bisimilarity the main point of interest is how it is finite, since for each recursive call to *rbc* we prevent the two current states from being checked again, meaning that at some point the algorithm is guaranteed to halt. The function $neighbours$

Algorithm 1 Recursive Bisimulation Check

```
function $rbc(s, s', States, Agents)$
    if $s = s'$ then
        return $true$
    else if $props(s) \neq props(s')$ then
        return $false$
    else if States = $\emptyset$ then
        return $true$
    else
        $States' \leftarrow States \setminus \{s, s'\}$
        $forth \leftarrow knowledgeCheck(s, s', States', Agents)$
        $back \leftarrow knowledgeCheck(s', s, States', Agents)$
        return $(forth\ and\ back)$
    end if
end function
```

Algorithm 2 Knowledge Check

```
function $knowledgeCheck(s, s', States, Agents)$
    for all $(t, Ags)\ in\ Neighbours(s, Agents, States')$ do
        for all $a\ in\ Ags$ do
            $hasMatching \leftarrow false$
            for all $(t', Ags')\ in\ neighbours(s', Agents, States')$ do
                if $a \in Ags'$ and $rbc(t, t', States', Agents)$ then
                    $hasMatching \leftarrow true$
                    break
                end if
            end for
            if !hasMatching then
                return $false$
            end if
        end for
    end for
    return $true$
end function
```

used in the *knowledgeCheck* function simply returns a set of tuples for each state visible from s and the set of agents it is visible to, limited to states in *States*.

5.5 Smallest bisimilar structure

Building on the previous algorithm, the next step is using it to create an algorithm for constraining a model to one of its smallest bisimilar structures, by filtering out all bisimilar states. While we normally would not need to worry about which states are filtered, as we are generating a checking log to visualize the checking process for our users, we wanted to avoid having the 'actual' state of our pointed model filtered out. As such, we simply make sure that the actual state lies first in our set of states when calling *bisimContract* in our application.

Algorithm 3 Bisimulation contraction

```
function $bisimContract(States, Agents, Edges)$
    $bisimMap \leftarrow Map < State, State >$
    for all $(state)$ $in\ States$ do
        for all $(otherState)$ $in\ States \setminus (state\ \cup\ keys\ in\ bisimMap)$ do
            if $rbc(state, otherState, States, Agents)$ then
                $bisimMap.put(otherState, state)$
            end if
        end for
    end for
    $CS \leftarrow States\ \setminus\ (keys\ in\ bisimMap)$
    $CE \leftarrow \{e \in Edges \mid startState(e) \notin CS\ and\ endState(e) \notin CS\}$
    $contractedModel \leftarrow (CS, Agents, CE)$
    return $contractedModel$
end function
```

5.6 Enumerating the set of announceable extensions

As we have now laid the groundwork of formulating how we can compute bisimulation contractions of models, we can move on to presenting how we can generate the set of announceable extensions. For this we will be using the definition of announceable extensions presented in Definition 5.4. While Ågotnes et al.'s definition in 5.5 is cleaner and more compact, our definition more closely resembles its pseudocode translation.

As can be seen, the algorithm for generating the set of announceable extensions is pretty much just generating the power set of states in the model, i.e. the set of all possible extensions, and then filtering it according to the rules in Definition 5.1. The *genEqClass*$(state, agents)$ function used is

Algorithm 4 Generating a coalition's set of announceable extensions

```
function $genAnnExts(contractedModel, actState, coalition)$
    $states \leftarrow contractedModel.states$
    $extensions \leftarrow \wp(states)$
    for all $extension\ in\ extensions$ do
        if $actState \notin extension$ then
            remove $extension$ from $extensions$
        else
            for all $state$ in $extension$ do
                $eqClass \leftarrow genEqClass(state, states, coalition)$
                for all $eqstate$ in $eqClass$ do
                    if $eqstate \notin extension$ then
                        remove $extension$ from $extensions$
                        break
                    end if
                end for
            end for
        end if
    end for
    return $extensions$
end function
```

mostly identical to the *neighbours* function used previously, except it returns the list of states that the coalition as a whole considers indistinguishable from the input state.

Now that we have not only our bisimulation contracted model, but also a way to generate all of the announceable formula extensions for any coalition, describing the algorithm for checking whether a group announcement formula is satisfied or not becomes entirely trivial. As these extensions are really just stand-ins for some announcement made by the agents as per the original definition in Definition 4.3, we can also regard them as constraints on our model. Going by our revised definition of the semantics behind the group announcement operator from definition 5.6, all our algorithm needs to do is to check whether all of the constrained models we get from applying these constraints to our bisimulation contracted model satisfy the post condition of the group announcement. As such, we translated the semantics of the group announcement operator into the following checking function, shown in Algorithm 5.

Like we previously mentioned we here make sure the set of states passed to *bisimContract* starts with the actual state we're checking our formula in, in order to simplify the visualization of our checking process. It should also be mentioned that the *check* function that gets called is implemented as an abstract function overridden by each operator in our system and that

Algorithm 5 Check function for group announcement operator

```
function $check(state, innerForm, model, coalition)$
    $contractedModel \leftarrow bisimContract(model)$
    $extensions \leftarrow genAnnExts(contractedModel, state, coalition)$
    for all $extension$ in $extensions$ do
        $constMdl \leftarrow constrainMdlBy(contractedModel, extension)$
        $extSatisfiesForm \leftarrow check(state, innerForm, constMdl, coalition)$
        if $!extSatisfiesForm$ then
            return $false$
        end if
    end for
    return $true$
end function
```

Algorithm 5 only shows the implementation for the group announcement operator. The *constrainMdlBy* function used here is merely a special case of updating a model through announcing the set of states directly instead of announcing a formula and constraining our model to the set of states that satisfy the given formula.

5.7 Complexity of the model checking problem

In this section we will be discussing and analyzing the run-time complexity of this model checking problem based on the algorithms we have presented so far.

For the discussion we will only be looking at the complexity of the group announcement operator and its related challenges, as the other operators of GAL are all fairly trivial to check, being O(1) operations, with the exception of the knowledge operator being O(n), where n is the number of states indistinguishable to the one we are checking the formula in.

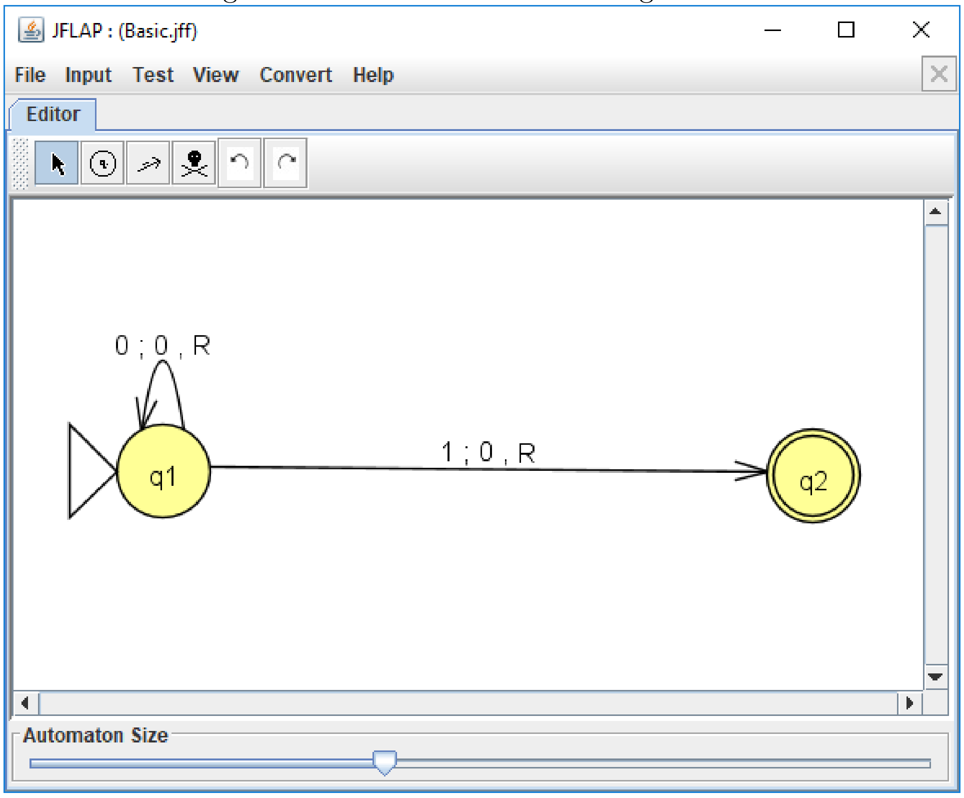
6 Implementation

In this chapter we will present our own implementation of a model checker for Group Announcement Logic called GALMC and describe its inner workings. Our initial goal with GALMC was to create an educational tool that could aid students with learning epistemic logic in a more visual manner. Therefore, while most other model checking utilities such as DEMO [?] stick to answering the user's queries with simple yes and no answers, we wanted to go beyond that by showing them not just whether their formulas hold, but also visualize why and provide the user with an easier, visual way of drawing models almost as they would on paper or blackboard. The reasoning behind this was that by allowing the user to manipulate the model through a simple click-and-drag interface and seeing how it can affect the valuation of various formulas they might gain a deeper understanding of the semantics involved.

One of the big questions that needed to be answered when we started building GALMC was how we wanted to not just present and visualize these highly abstract models, but also allow the user to manipulate them in a way that would be intuitive and easy to grasp. As I had previously used a tool called JFLAP[3] with great success when teaching students as a TA about Turing machines and finite state automata (FSAs) I ended up drawing most of my inspiration from it when drawing up my initial sketches for how I imagined my own tool might look. While my own experience with JFLAP is mostly limited to visualizing, editing and playing with Turing machines and finite state automatas, it is a fairly sophisticated package of graphical tools covering also covering many other concepts of formal languages and automata theory. JFLAPs editor is relatively simplistic, but it still helped my own and my students' understanding of FSAs tremendously by allowing us to interact and play with what is otherwise a really abstract concept and as such, I wanted to see if I could create a similarly potent learning aid for epistemic logic.

[3]www.jflap.org

Figure 6.1: A basic two-state turing machine

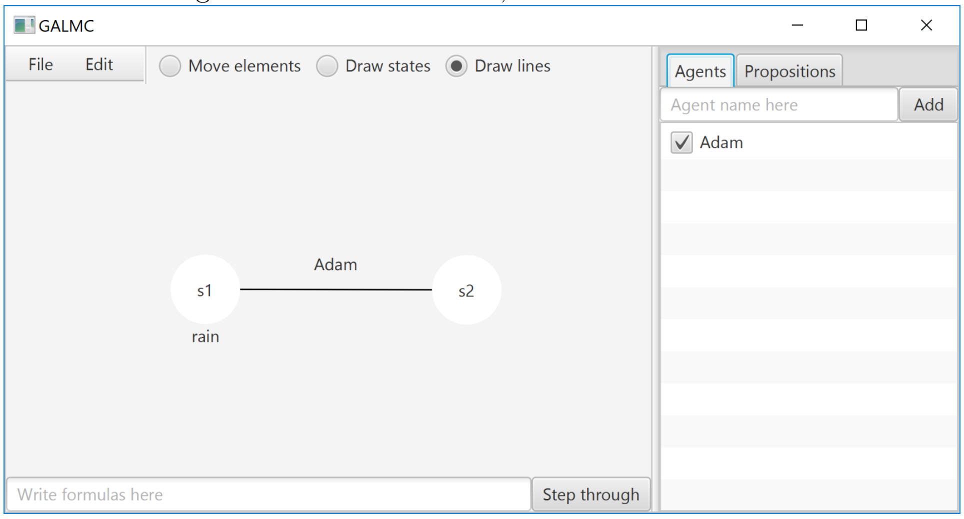


While I could have created a simpler non-graphical model checking tool for GAL similar to DEMO, I wanted to create something more powerful that could help visualize epistemic logic the same way that JFLAP visualizes Turing machines. This ended up taking a lot of work, but my justification is that although a similar non-graphical tool would probably have helped me teach my students how FSAs and Turing machines work as well, I highly doubt it would have been anywhere near as effective without being able to visualize the 'how's and 'why's and instead only gave 'yes' or 'no' answers to our queries in the same manner that most model checking utilities do.

6.1 Visualization of Kripke structures

Although finite state automatons and epistemic logic might initially seem relatively far detached, the Kripke structures we use share a fair few similarities to FSAs that made me realize I could visualize our models in almost the same manner as JFLAP does its automata. For instance, they both consist of a set of states, and while FSAs and Turing machines have transition rules, and Kripke models have an indistinguishability relation, they can both basically just be visualized as edges in a graph where the nodes are our states. Whereas JFLAP labels its edges with the each transition rule, we label ours with the set of agents that considers our pair of states indistinguishable and additionally label each state with the set of propositions that hold in it. We present our tool visualizing a basic model in Figure 6.7.

Figure 6.2: A basic model, visualized in GALMC



As can be seen in Figure 6.7, the UI of GALMC is fairly close to that of JFLAP, as I wanted to recreate its way of drawing models almost as one does with pen and paper as much as possible so that the interface feels 'natural' and intuitive to someone with little previous experience with Kripke structures. As such, the editor was made to be as simple as possible, consisting of only three main tools, one for selecting and moving elements, one for drawing new states and one for creating edges between them. Additionally, the editor provides two side panels for managing the lists of properties and agents in our models, where the user can also select which ones to use when drawing new components or to apply to existing ones.

6.2 Checking formulas

After the user creates their model, the next step is to start checking formulas against it. One trade-off that had to be made here was whether or not to stick to the original symbols for the various operators in our language or to come up with replacements which are easier to type in. As requiring the user to memorize half a dozen unicode character codes would probably not offer the best user experience, I opted to replace them with more easily accessible replacements which would presumably make sense to most users. By the assumption that most users of my tool would be at least somewhat familiar with programming, I ended up taking most of my inspiration from symbols used to represent boolean operators in programming, such as replacing $\vee$ and $\wedge$ with | and & for disjunctions and conjunctions respectively, as they can basically be seen as 'or' and 'and'. A full list of operators and their replacement symbols are displayed in Table 1. Note that while announcements are identical to how they are in GAL, I elected to drop the curly braces around the set of agents in group announcements to save the user the effort of typing them in. The knowledge operator is fairly similar, except that GALMC forces agent names to start with an upper-case letter and that the parentheses around the known formula are required. The reason behind these changes was partly to be able to make it easier for the user to tell names of agents apart from propositions in more complex formulas, (GALMC forces agent names to be capitalized, whereas propositions have to be lower case) but it also ended up making the parser a lot simpler to write, which will be presented and discussed a later section. That said, the editor translates any formula the user types in back into proper legal formulas when displaying them, which should make the tool less confusing to use, once the user gets familiar with the tool and the logic itself [4].

Table 1: Table of operators and their symbols in GALMC

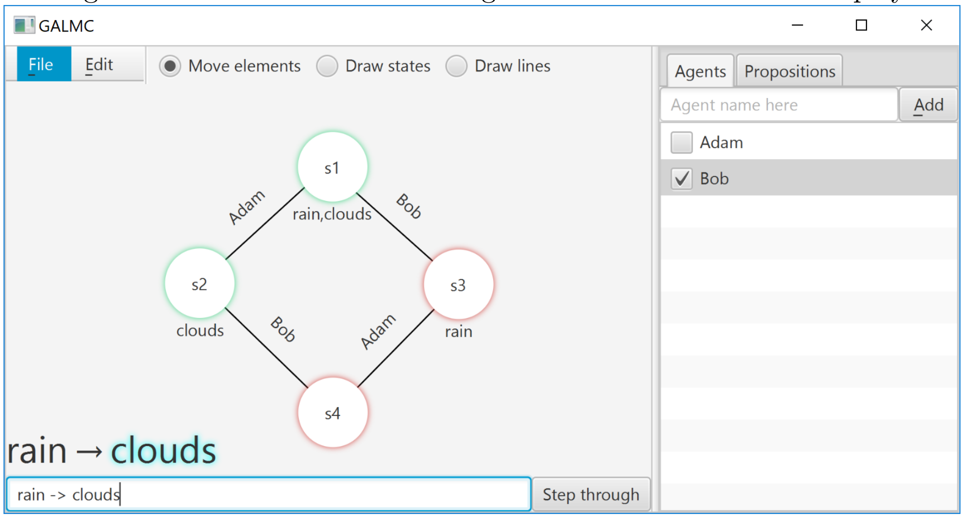
Operator	Logical symbol	Replacement
Negation	$\neg$	!
Conjunction	$\wedge$	&
Disjunction	$\vee$	|
Implication	$\rightarrow$	->
Knowledge	$K_{ann}\varphi$	$KAnn(\varphi)$
Announcement	$[\varphi]\psi$	$[\varphi]\psi$
Group Announcement	$[\{ann, bob\}]\varphi$	$[Ann, Bob]\varphi$

When the user finishes typing in their formula, the tool checks their formula against each state in their model and colors each state based on whether or not the user's formula is satisfied in that state. However, in addition to translating the user's input into a legal formula and showing which states of the model the formula holds in, GALMC also lets the user can also mouse over parts of their original formula in order to check the valuation of it's subformulas, which can be seen in 6.7. The idea here was that by allowing the user to quickly break formulas apart and see why their various subformulas change the valuation of their parent the way they do, I could visualize the semantics behind each operator in our language in a playful manner that might make the logic more enjoyable to learn.

(TODO: replace image with one that shows mouse pointer)

[4]For a more to the point user manual, see: https://github.com/AndersKaareEide/MCGAL/wiki

Figure 6.3: Illustration of mousing over interactive formula display



rain → clouds

rain -> clouds

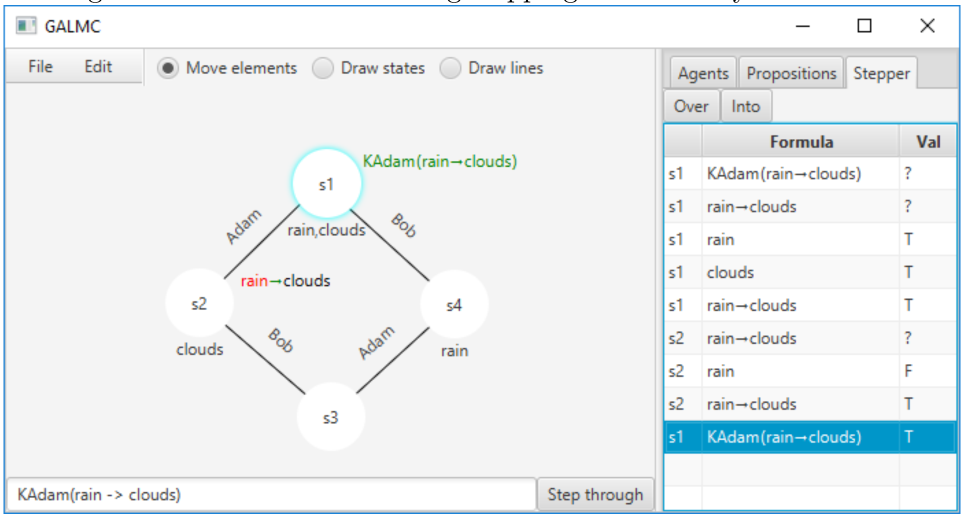
6.3 Visualizing the checking process

One feature of JFLAP that I have not yet mentioned however, is its stepping feature, which was also one of its most useful features when I was using it to teach my students. As you present your Turing machine in JFLAP with a set of input, the tool also allows you to step through the program of your machine one instruction at a time, while also visualizing how it manipulates the contents of its input and which state the machine is currently in. Obviously this feature was remarkably useful in showing how the user's machines operate, letting my students debug their instruction sets in almost the same manner they would debug a program in a high level programming language. As such, it not only made Turing machines and FSAs much easier to grasp, but it also made getting them to work correctly a far less frustrating process, meaning it was not only easier for me to teach them, but I also ended up making the exercises more complex than they had been previous years, without giving my students any problems solving them.

Naturally, I wanted to create something similar for my own tool as well, something that could visualize the process behind checking each operator in the user's formula the same way that JFLAP shows each instruction being executed and the effects of doing so. I decided that the simplest way of creating such a visualization would be to make a log of each check the tool makes when evaluating a formula, logging not just the operator being checked, but also its current valuation, if known, and which state it is being checked in. Continuing with the model from our previous figure, Figure 6.3 shows us using our tool to generate a log of how the program checks the given formula against a specific state. From this, the user can get a fully reproducible guide they can follow when checking other formulas on their own, which I imagine will be quite helpful in learning how the operators work. The user can also step forwards or backwards through this process at any time by either clicking the step they want to skip to, or browsing with the arrow keys.

Note that in addition to this log, the program also visualizes which (sub)formulas are being checked against the various states as colorized labels that change color as the user steps through their formula based on the valuation of the operators the labels represent.

Figure 6.4: Illustration of using stepping functionality in GALMC

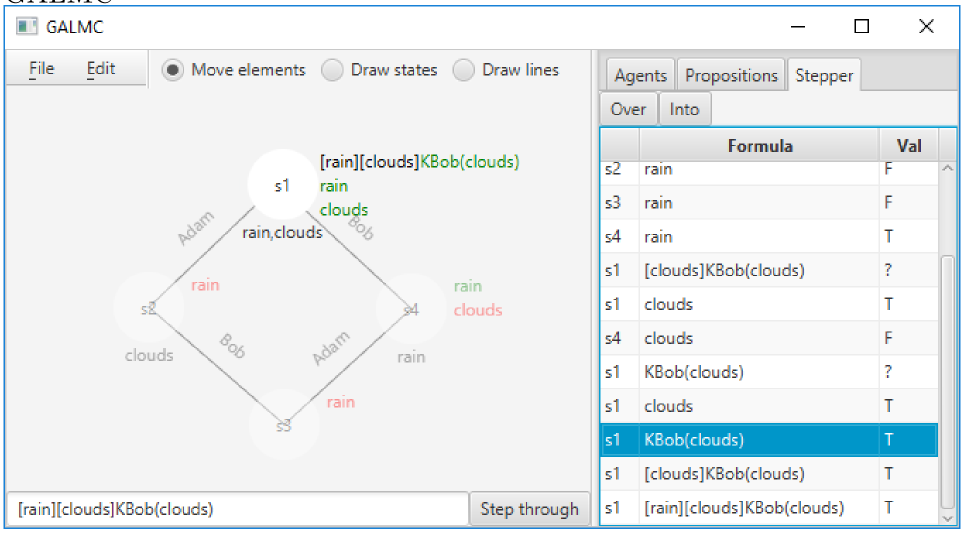


KAdam(rain -> clouds)

	Formula	Val
s1	KAdam(rain→clouds)	?
s1	rain→clouds	?
s1	rain	T
s1	clouds	T
s1	rain→clouds	T
s2	rain→clouds	?
s2	rain	F
s2	rain→clouds	T
s1	KAdam(rain→clouds)	T

Returning to our previous comparison between the stepper in JFLAP and the one I wished to implement, there is still an interesting challenge. While the instruction sets and states of a Turing machine are static, the epistemic models we use in dynamic epistemic logic are naturally quite dynamic, and can be updated based on public announcements in PAL or more complexly, group announcements in GAL. I chose to represent these model updates by graying out the states and edges that were removed by the update and to update which states should be 'hidden' based on which branch in the tree-like structure of the original formula the user is currently stepping through. Naturally, the tool also supports formulas with multiple announcements nesting them in any fashion the language allows, always visualizing the relevant sub-model that formulas are being checked against, an example of which can be seen in Figure 6.3. Note that the tool also generates formula labels for which sub-formulas under knowledge or announcement operators have been checked in the various states, so that the user can immediately see why for example, an agent does not know something, or why a state has been filtered out in a model update.

Figure 6.5: Visualization of the effects of chained public announcements in GALMC



[rain][clouds]KBob(clouds)

	Formula	Val
s2	rain	F
s3	rain	F
s4	rain	T
s1	[clouds]KBob(clouds)	?
s1	clouds	T
s4	clouds	F
s1	KBob(clouds)	?
s1	clouds	T
s1	KBob(clouds)	T
s1	[clouds]KBob(clouds)	T
s1	[rain][clouds]KBob(clouds)	T

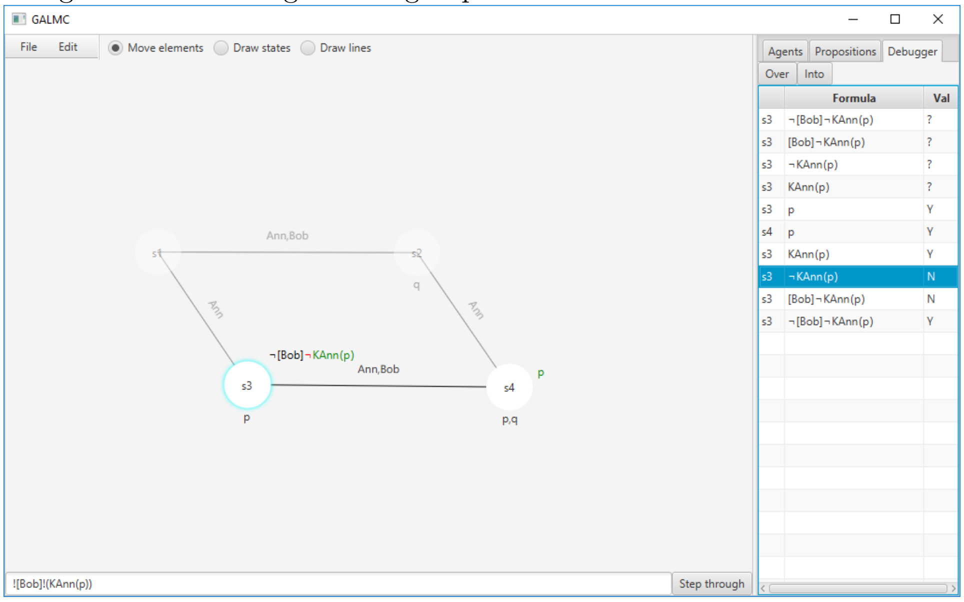
6.4 Visualizing group announcements

Now that I have presented most of the tool I have created and how it solves the various challenges of visualizing the process of checking formulas containing the other operators in our language, it is time to move on to the final, and by far the most interesting operator in GAL, group announcements. As our motivation behind creating GALMC was to create a learning tool that could help new logicians understand how the semantics of GAL work, being able to generate examples which highlight interesting properties of our models is highly important. As such, GALMC was designed from the ground up to be able to trace its steps through the model checking process so that it can also display each step of this process in an intuitive manner, which can be seen in our previous figures. Elaborating on the logging process mentioned before behind this tracing, the tool also keeps track of sub-formulas and formula depth as it goes through the operators. The reason behind keeping track of this in the logs was to create a more easily navigable tree-like structure, as the number of steps required to check a formula containing group announcements can quickly explode. Because of this, I wanted to give the user the ability to skip through chunks of the process they might not be particularly interested in. This tree-structure allows them to for example view each state the tool checks against a particular knowledge operator, or even skip through each of the possible updated models a coalition can reduce a model to through their announceable extensions, without having to step through the checking of the inner formula every time.

(TODO: Flette inn med resten av det over)

As the tool provides a view of the model and keeps track of what the

Figure 6.6: Checking a basic group announcement formula in MCGAL



![Bob]!(KAnn(p))

	Formula	Val
s3	¬[Bob]¬KAnn(p)	?
s3	[Bob]¬KAnn(p)	?
s3	¬KAnn(p)	?
s3	KAnn(p)	?
s3	p	Y
s4	p	Y
s3	KAnn(p)	Y
s3	¬KAnn(p)	N
s3	[Bob]¬KAnn(p)	N
s3	¬[Bob]¬KAnn(p)	Y

model it is currently checking against looks like, it can also visualize the effects of announcing multiple different formulas. This means the tool can also visualize the result of constraining the model to the various formula extensions a coalition can announce, which is what is being displayed in Figure 6.4. In this example, we are checking whether the formula $\neg[Bob]\neg K_{Ann}[p]$ holds in state s3 of our model. The formula roughly translates to: 'It is not the case that Bob is unable to make Ann know p' or more simply in its dual form: 'Bob is able to make Ann know p'. From the visualization of our model in Figure 6.4 we can see that since Bob is able to reduce the model to only states where p holds, Ann also trivially knows that p holds in this updated model, satisfying our original formula. In this somewhat trivial example, the tool ended up only having to try announcing a single formula extension before it found an extension that made our original formula true. If we were to check a more complex formula however, it might end up checking many different announcements, each generating a different updated model after its announcement which the tool will helpfully visualize, giving the user insight into a coalition's capabilities. Note that the previous example could also have been made easier by rewriting the formula as $\langle Bob\rangle K_{Ann}[p]$, but unfortunately the tool does not (yet) support diamond operators.

(TODO: Vise eksempler på visualisering av kunnskap de re vs de dicto. Forklare hvorfor skillet er interessant. Burde være mulig å bare knabbe både formler og modeller fra GAL-paperene (de re blir mest sannsynlig for klumpete å vise))

6.5 Choice of platform

(TODO: Write introduction of what Kotlin is)

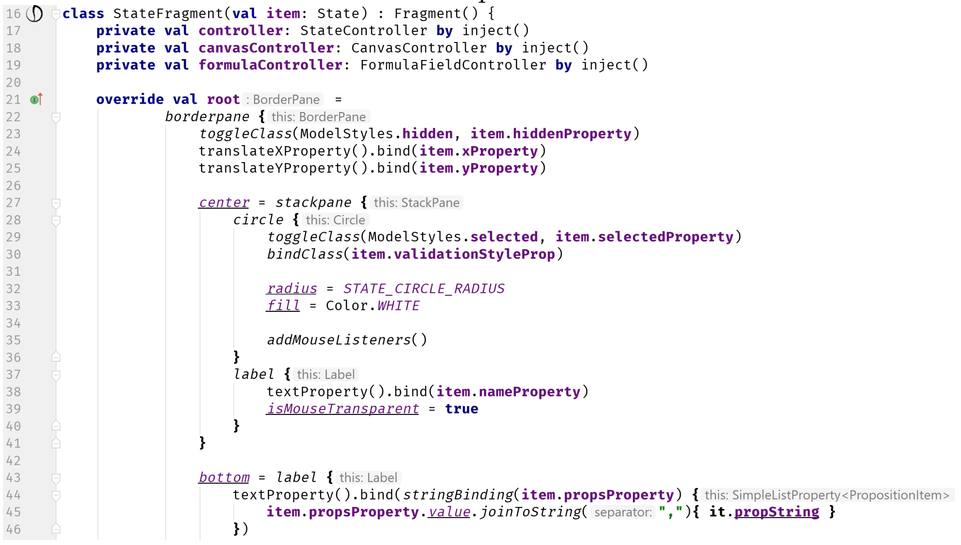
Our model checker was implemented in Kotlin as it is a new and exiting language while also being based on the JVM, providing access to the vast libraries and tools written for Java, including JavaFX and a Java hook for ANTLR which will be discussed in more detail later in this section. Kotlin additionally has a smooth and concise syntax as well as lending itself very well towards writing functional code despite being object oriented through shorthands for lambda expressions, support for proper function types and distinctions between mutable and immutable structures and variables. Among the other benefits of the language is that Kotlin being a relatively fresh and recent language, has been able to draw inspiration from other languages meaning it has language features such as type inference, string interpolation and default values for making function arguments optional, so you no longer have to write three different nearly identical constructors for the same class as one would in for example, Java. While the language is still relatively young, being first unveiled in 2011[?], it is being backed by JetBrains, who are well known for their suite of plugins and IDEs such as IntelliJ and PyCharm, so the language has excellent tooling support despite its young age.

As for the choice of graphics library to build the application's user interface in, the choice fell on a Kotlin wrapper for the de facto standard Java GUI library of JavaFX, called TornadoFX[5]. While JavaFX is a widely used and mature library with powerful features, we also feel it tends to be somewhat clunky and verbose unless one dumps the layout and positioning of components into specialized XML-files. While this helps clean up classes representing UI-components, we also find it makes dynamic component generation clumsier. Our reasons for wanting to use TornadoFX over plain JavaFX is that TornadoFX, being a Kotlin library is able to have a much cleaner and prettier API by utilizing Kotlin features such as lambda expressions attached to receivers to create composable builder functions which generate your JavaFX component hierarchy in an imperative fashion. Another excellent feature of TornadoFX is its concise shorthands for creating dynamic bindings between UI components and observable sets of data. An example of this would be using the standard bindChildren() function to dynamically create and destroy UI components representing states in the user's epistemic model based on changes to that internal list of objects which represent states in a single line of code, by simply feeding the bindChildren() function a reference to the observable list of states and a transformation function that converts the data structure representing a state, to the corresponding UI component, in our case simply the constructor of this component. If anything, I would say that TornadoFX's greatest feature is how its

[5]Library homepage at: `https://tornadofx.io`

composable builder functions allows you to circumvent the normally inverse order of declaration and creation of UI components compared to their order in the component hierarchy, which can be seen in Figure 6.5.

Figure 6.7: Code handling how the UI components representing states are built. Note the conciseness due to implicit contexts.



```
class StateFragment(val item: State) : Fragment() {
    private val controller: StateController by inject()
    private val canvasController: CanvasController by inject()
    private val formulaController: FormulaFieldController by inject()

    override val root : BorderPane  =
            borderpane { this: BorderPane
                toggleClass(ModelStyles.hidden, item.hiddenProperty)
                translateXProperty().bind(item.xProperty)
                translateYProperty().bind(item.yProperty)

                center = stackpane { this: StackPane
                    circle { this: Circle
                        toggleClass(ModelStyles.selected, item.selectedProperty)
                        bindClass(item.validationStyleProp)

                        radius = STATE_CIRCLE_RADIUS
                        fill = Color.WHITE

                        addMouseListeners()
                    }
                    label { this: Label
                        textProperty().bind(item.nameProperty)
                        isMouseTransparent = true
                    }
                }

                bottom = label { this: Label
                    textProperty().bind(stringBinding(item.propsProperty) { this: SimpleListProperty<PropositionItem>
                        item.propsProperty.value.joinToString( separator: ","){ it.propString }
                    })
```

6.6 Language and interpretation

(TODO: Introduser ANTLR og forklar hva det egentlig er)

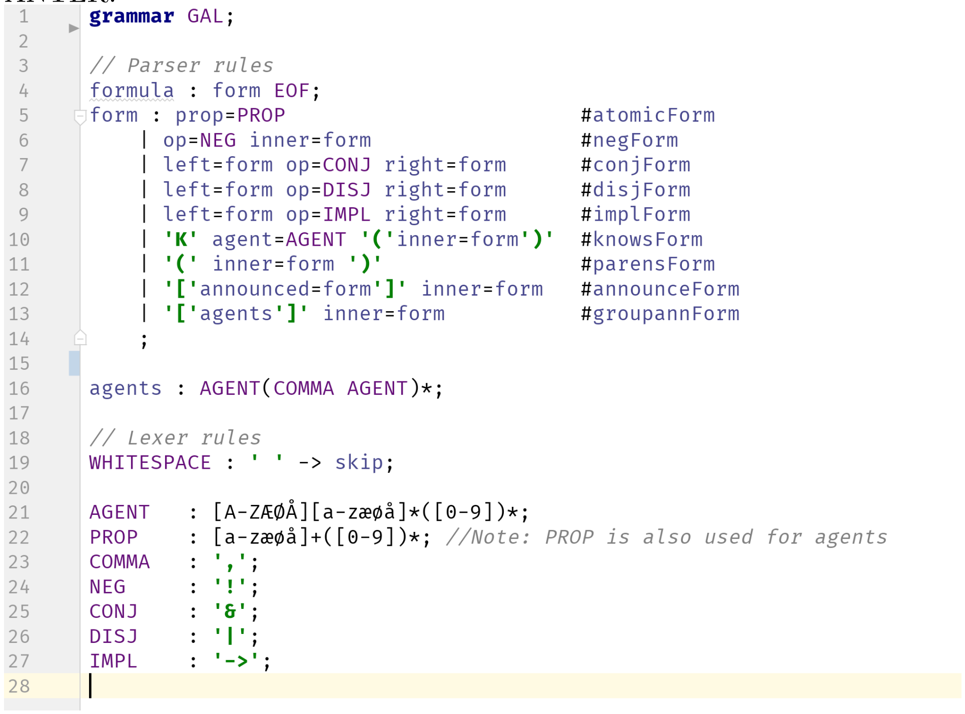
As for parsing the formulas, I went with ANTLR[6] for generating the parsers I needed. ANTLR is a pretty powerful tool for generating parsers and lexers based on easily definable grammatical rules like the ones in Figure 6.6

These grammatical rules are used by ANTLR to generate parser and lexer classes which implement these rules in a manner that allow me to easily translate from raw text and lexical tokens into data structures of my own design which represent the various operators and encapsulate their semantics. One thing to note in Figure 6.6 are the signs used to represent negation, conjunction and disjunction. As the commonly used symbols for these operators do not exist on normal keyboards, the UI would either need extra buttons to facilitate inserting these symbols or use more easily inputable surrogate symbols. I chose the latter, going for the boolean equivalents of using exclamation marks for negations, not, ampersand for conjunctions, and, as well as the vertical bar, or 'pipe' character commonly representing the or operator. Note that GALMC automatically converts these characters to their 'correct' symbols when displaying the input formula however,

[6] `http://www.antlr.org/`

(TODO: Include and draw comparisons between ANTLR and BNF)

Figure 6.8: Grammatical rules for parsing formulas in GALMCwith ANTLR.



```
grammar GAL;

// Parser rules
formula : form EOF;
form : prop=PROP                               #atomicForm
    | op=NEG inner=form                        #negForm
    | left=form op=CONJ right=form             #conjForm
    | left=form op=DISJ right=form             #disjForm
    | left=form op=IMPL right=form             #implForm
    | 'K' agent=AGENT '('inner=form')'         #knowsForm
    | '(' inner=form ')'                       #parensForm
    | '['announced=form']' inner=form          #announceForm
    | '['agents']' inner=form                  #groupannForm
    ;

agents : AGENT(COMMA AGENT)*;

// Lexer rules
WHITESPACE : ' ' -> skip;

AGENT   : [A-ZÆØÅ][a-zæøå]*([0-9])*;
PROP    : [a-zæøå]+([0-9])*; //Note: PROP is also used for agents
COMMA   : ',';
NEG     : '!';
CONJ    : '&';
DISJ    : '|';
IMPL    : '->';
```

as can be seen in Figure ??. Another small concession I had to make in order to differentiate between agents and propositions was to force agent names to be capitalized as I would otherwise have to resort to using different symbols to differentiate between regular public announcements and group announcements.

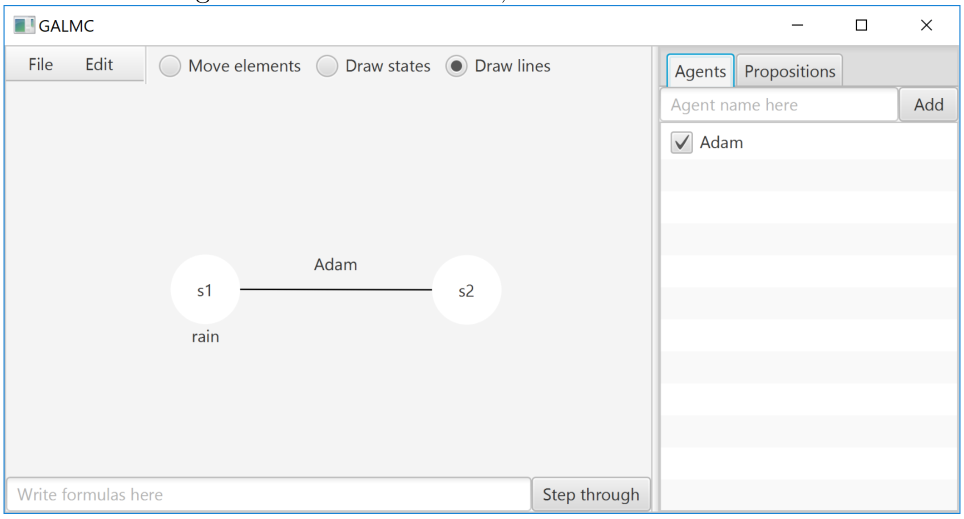
6.7 Model checking

As the focus when making the application was mainly on making the it as easy to use as possible while presenting information in a visual manner that is easy to grasp, we ended up making several deviations from the logical definitions of the various structures that have been discussed throughout the thesis. Here we will discuss some of them as well as why these deviations were useful to us. Additionally we will also describe the inner workings of our tool, the technologies and frameworks it is built on as well as our reasons for choosing these.

A key difference here is how we chose to represent the components of our epistemic models themselves. In Definition 4.1 we present $\sim$ as a function from each agent to their respective equivalence relation for every state in the model, whereas in GALMCwe instead chose to represent these equivalence relations as a set of edges represented as objects consisting of pairs of states and sets of agents. The reasons for this ties back into how we wanted to present an interactive view of the models in our application as it felt cleaner represent each edge between states as a concrete objects which we could then bind a UI-component to, than hold onto sets of sets of states for each agent. Since each edge also has a reference to the set of agents it is valid for, it becomes trivial to visualize this information as well.

We also chose to flip the valuation function by letting states hold a reference to the set of propositions they satisfy rather than each proposition being linked to the set of states they are satisfied in. Our reasoning here is much the same as we highlight which propositions are true in which states through bindings to each state's set of propositions, which is far simpler than having to go through the entire set of propositions each time the user updates which propositions a given state satisfies. An example of how this looks can be seen in Figure 6.7, with both states and the edges between them being just bindings to simple data structures, updating whenever their underlying data does.

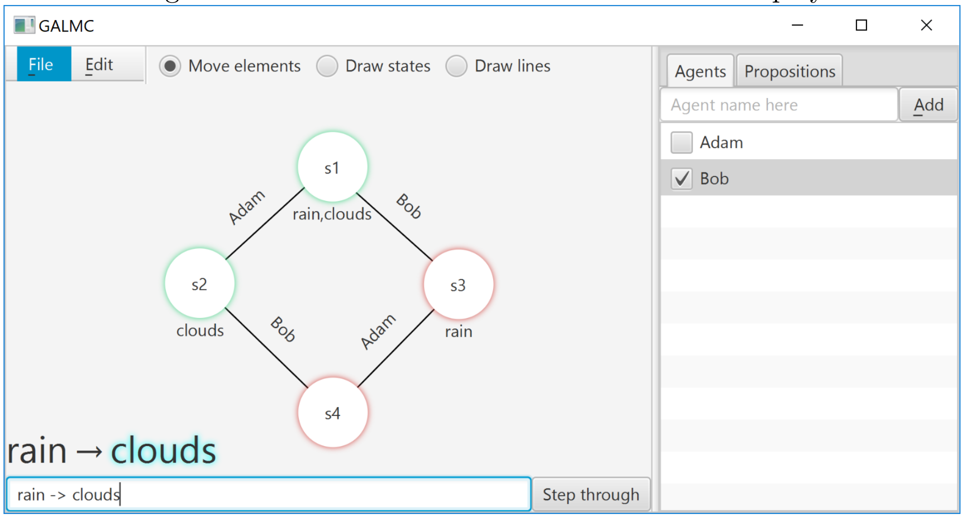
Figure 6.9: A basic model, visualized in GALMC



Additionally, we also made each state aware of which edges it is connected to, trivializing the logic behind finding 'neighboring' states for a given agent as we simply filter the set of edges our state is connected to based on each edge being valid for said agent and return a list of states these edges lead to.

Continuing our discussion of how we implemented our formula structures in GALMC, we use a text parser generated from a small ANTLR grammar file in order to parse formulas entered by the user into a tree-like structure representing their formula. When checking this formula the user then has a choice of whether they simply want to see which states in the model satisfy their formula, or whether they want a more detailed visualization of why their formula is or is not satisfied in a specific state. If the user then opts for the first option and check their formula against the whole model, then the tool will highlight the states as either green or red, depending on whether that state satisfies the input formula. The tool also generates an interactive visual representation of the formula where the user can mouse over each operator to check the sub-formula that operator represents against the model as seen in Figure 6.7. The idea behind this being that the user might be interested in quickly seeing which states in their model satisfy the various sub-formulas to help better understand how the original operator works. The way this was implemented was to basically create UI components for each operator in the formula and the sub-formula they represent and then check each sub-formula against the model as the user mouses over them, highlighting which part of the formula they represent.

Figure 6.10: Illustration of interactive formula display

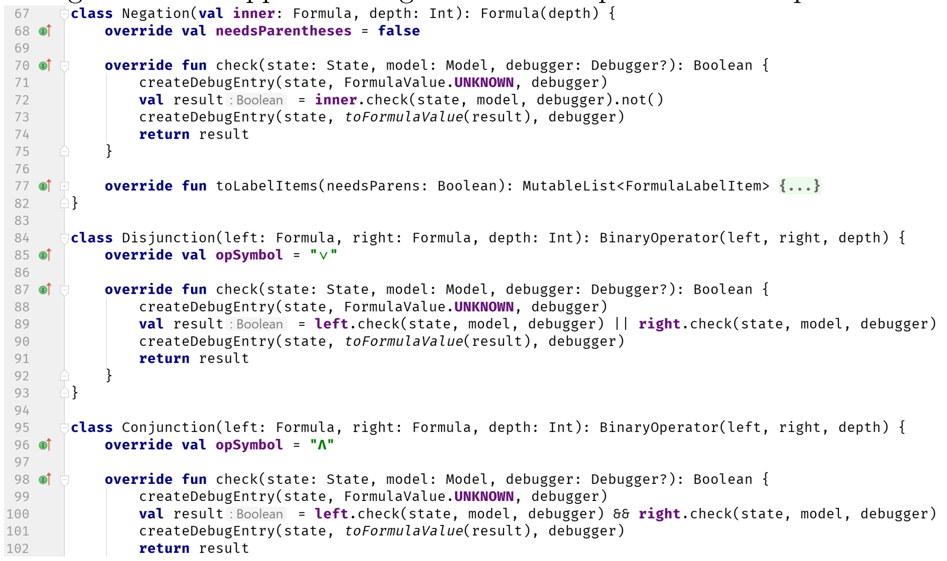


rain → clouds

rain -> clouds

How step-by-step visualization works is somewhat more involved. A high-level description of how it is implemented is that the 'debugger' hooks into the *check* function of the formula, and logs each step of the checking process. Each log then carries information about what the valuations of each operator was at that point in the process, so that this checking process can be played back and visualized in the form of showing the various operators change color as their valuations become known. One pretty cool feature of this step-by-step visualization is that it allows the user to see which states get filtered out by model updates and even see the checking process play out in each formula extension a given coalition can announce in the case of checking group announcement formulas.

Figure 6.11: Snippet showing how various operators are implemented



```
class Negation(val inner: Formula, depth: Int): Formula(depth) {
    override val needsParentheses = false

    override fun check(state: State, model: Model, debugger: Debugger?): Boolean {
        createDebugEntry(state, FormulaValue.UNKNOWN, debugger)
        val result : Boolean  = inner.check(state, model, debugger).not()
        createDebugEntry(state, toFormulaValue(result), debugger)
        return result
    }

    override fun toLabelItems(needsParens: Boolean): MutableList<FormulaLabelItem> {...}
}

class Disjunction(left: Formula, right: Formula, depth: Int): BinaryOperator(left, right, depth) {
    override val opSymbol = "∨"

    override fun check(state: State, model: Model, debugger: Debugger?): Boolean {
        createDebugEntry(state, FormulaValue.UNKNOWN, debugger)
        val result : Boolean  = left.check(state, model, debugger) || right.check(state, model, debugger)
        createDebugEntry(state, toFormulaValue(result), debugger)
        return result
    }
}

class Conjunction(left: Formula, right: Formula, depth: Int): BinaryOperator(left, right, depth) {
    override val opSymbol = "∧"

    override fun check(state: State, model: Model, debugger: Debugger?): Boolean {
        createDebugEntry(state, FormulaValue.UNKNOWN, debugger)
        val result : Boolean  = left.check(state, model, debugger) && right.check(state, model, debugger)
        createDebugEntry(state, toFormulaValue(result), debugger)
        return result
```

(TODO: Create UML diagram of data structures and how they are connected)

6.8 Future work

(TODO: Skrive om å måle pedagogisk verdi, muligens peke ut som fremtidlig masteroppgave)

Having developed what we consider to be a potentially highly useful education tool, the big elephant in the room when discussing future work is actually measuring its educational benefit. As both user testing and educational impact studies go beyond the scope of this thesis, we certainly hope that someone might be willing to take up the torch and prove the value of GALMC in such settings. While GALMC is fully functional and usable as an educational aid in its current state, there are also a fair few features that we simply did not have time to implement, a few of which we will discuss.

One of the simpler additions to the tool is implementing the dual, also known as 'diamond' version of the announcement operators. As these operators do not add any additional expressiveness or capabilities to the model checker they were never prioritized as they can always be expressed through the negation of the box operators. It would still be nice to have support for these operators directly however, to cut down on formula length and complexity when visualizing larger formulas. As both the ANTLR grammar and related formula components are easily extendable, implementing these operators would be fairly trivial as their underlying semantics are basically already implemented.

There are also a fair few additions to the UI we would have liked to implement, such as separately displaying the set of formula extensions a coalition can announce or provide the user with more informative error messages when attempting to parse syntactically incorrect formulas. Visualizing these formula extensions should also not prove all too challenging as the extensions are already generated when checking formulas. Similarly, the ANTLR parsers provide most of the context necessary to present inform the user of which part of their string caused an error, although more complex reasoning around how to parse ambiguous structures in regards to how to handle missing parentheses and the like might be more challenging.

(TODO: Fix apostrophe positioning somehow)

There were also plans for generalizing GALMC's model serializer in order for to be able to export the models as formats beyond its current basic binary format such as GEXF[7] in order to be able to view these models in other tools such as Gephi[8]. As the intended users of the tool are mainly students attempting to gain a better understanding of the semantics of the operators and structures in group announcement logic however, the feature was eventually scrapped as the models created would likely not be all that interesting to visualize in external tools anyway and the work involved would be fairly substantial for a feature that would probably go unused by most users. Continuing on model serialization, we would also have liked to be able to store additional information or metadata about each model, such as being able to write notes about interesting properties a model might have or formulas that highlight said properties when checked against these models.

[7]`https://gephi.org/gexf/format/`
[8]`https://gephi.org/`

7 Conclusion

Conclusion pending

8 Appendix A. List of features

Conclusion pending