To be announced:

Understanding and model checking Group Announcement Logic

University of Bergen

Master's thesis in Information science

Anders Eide

Supervisor: Truls Pedersen

May 31, 2019

Abstract

In this master's thesis we 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 also define how to enumerate the set of ways a given coalition can restrict a model as well as present pseudocode algorithms describing how we translated these definitions in our model checker.

Contents

1 Background

1.1 Motivation

In 2010, Ågotnes et al. published a paper on their extension of Public Announcement Logic called Group Announcement Logic (GAL) [1]. While the group announcement operator introduced in this logic is highly expressive and enables us to quantify over possible announcements groups of agents can make and their ability to impact the system around them, this expressiveness can also be somewhat confusing and complicated to understand at first.

For this reason, having a learning aid of some sort, capable of visualizing how the various operators of this logic (and group announcements in particular) work could greatly aid future students and logicians in understanding how GAL and related logics work, especially seeing how other visual tools such as JFLAP[1] have previously been used to great effect in teaching similar topics.

When we work with these logics, we are commonly trying to assert whether or not our models exhibit some property. While we will return to GAL and define and discuss it later in this thesis, it might be a good idea to start with a discussion of how these logics are typically used. This brings us to the topic of model checking, which is one of the key concepts this thesis will be covering.

1.2 Model checking

Phrased abstractly, model checking is a way for us to further our understanding of complex systems or structures by formalizing ways of analyzing them. Generally it involves the process of verifying whether or not our model exhibits certain properties. Depending on the type of logic we are working with, our models are commonly represented as Kripke structures [15] (which we will be using and discussing later in this thesis), but can also be described in more purely algebraic forms. While we use these models to model certain aspects of the world around us, we also require ways of describing its properties, which is where our logics come in. The logics define not only what constitutes a legal formula (i.e. a syntax), but also their meaning (i.e. the semantics of all valid ways of building formulas).

Model checking has a rich tradition in the field of computer science and has seen a lot of practical use in formal software and hardware verification. A few examples of properties which can be interesting to verify would be liveness (regardless of system state, we can always reach a 'reset' state and guarantee that the system will acknowledge and answer an incoming request, i.e. not deadlock) and safety (ensuring that the system can never enter a potentially dangerous state). While model checking is in most cases not a replacement for traditional forms of software

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

and hardware testing, it can be seen as a complementary technique to be used in high-risk scenarios where the cost of failure offsets the investment into utilizing more formal verification methods to ensure correctness.

A more concrete example that also highlights one such high-risk scenario is the case of the Ariane 5 rocket, mentioned by Clarke et al. in their handbook on model checking [5], as an example of why we need formal methods for verifying software and hardware integrity. This example, where the rocket ended up exploding shortly after takeoff due to a conversion error in the software controlling the guidance system, shows just one of many situations where we are forced to rely on systems to carry out tasks of critical importance to us. As these systems grow not just in complexity, but also in importance and their effects on our daily lives, it also becomes increasingly important to improve our methods of verifying the behavior of these systems.

Traditionally, practical applications of model checking has been dominated by simpler forms of logic such as Computation Tree Logic [4] (CTL) and Linear time Temporal Logic [13] (LTL), whereas more complex logics such as Public Announcement Logic [12] (PAL), Alternating-time Temporal Logic [2] (ATL) and Group Announcement Logic tend to be regarded as more 'academical pursuits'. While these simpler logics have the obvious advantage of being far simpler to efficiently model check, leading to a wide variety of available tools such as BLAST [8] or SPIN[2] [9] (which at the time of writing will soon have its 30th anniversary while still being actively maintained), they are also less expressive than their more modern academic counterparts. However as our systems grow ever more distributed (especially in today's focus on 'autonomous' systems and IoT appliances) it might no longer be enough to verify the behavior of our systems in isolation. For example, if we are to guarantee the safety of tomorrow's fully self-driving cars, it could be useful to simulate the actions of these systems as autonomous agents, individually capable of influencing the state of a more complex network of such agents. It is when it comes to modeling more complex situations and networks such as these, that we might require more expressive forms of logic such as for example ATL, its epistemic extension ATEL [14] (Alternating-time Temporal Epistemic Logic), or as we will be exploring, GAL. As these epistemic logics allow us to not only reason around the knowledge of each agent in our systems, but also their knowledge of other agents' knowledge, it makes sense that these epistemic logics could greatly benefit the field of formal software verification, given enough time, interest and support in the form of powerful model checking tools. Unfortunately however, the complexities of these logics also mean that the problem of automated model checking for them is not sufficiently well explored. Looking back at our previously mentioned logics like CTL and LTL, we believe that the field of model checking

[2]Public repository for SPIN can be found at: https://github.com/nimble-code/Spin

dynamic epistemic logics lags far behind, especially in the case of GAL, which does not yet have a model checker implemented for it at all.

1.3 State of the art

While model checkers do exist for similar kinds of logic such as DEMO_S5[3] for PAL and SMCDEL[4] [7] for DEL (Dynamic Epistemic Logic) most of them tend to be rather hard to use and often require a good deal of time and expert knowledge in order to learn how to use. This, combined with the fact that both of our previously mentioned examples base themselves on using a Haskell REPL (read-eval-print-loop) for user interaction means they would most likely also lend themselves poorly to teaching. While there does exist some fairly powerful visualization libraries/modules generating visualizations of Kripke structures such as Gattinger's KripkeVis module[5], they still require a fairly great deal of technical competence in order to set up correctly, and at least in the case of KripkeVis, only produce static visualizations, instead of facilitating direct manipulation of the models [6].

A key reason for why these more complex logics have not seen much use in practical applications of model checking yet, is that their expressiveness not only leads to increased complexity in terms of the implementation of model checkers, but also in terms of computational complexity of the model checking process itself. While there has been some recent developments in terms of speeding up model checking of epistemic logics by way of symbolic representations such as using binary decision diagrams [3] (BDDs) instead of verbatim models for DEL [7], this thesis is more concerned with the creation of an easily approachable tool for learning, rather than for research or practical applications which might require more efficient algorithms.

This brings us to where we wish to make a contribution to the field. By developing an easy to use model checking tool that can be utilized in an educational context, we hope to spark more interest in these logics and help make them more accessible to others by providing intuitive visualizations of how the logic works. As such we not only wanted to make a model checker that supports group announcement logic, but also make it as intuitive as possible by giving it a graphical user interface that allows the user to interact with their models in a more direct manner than more algebraic formats used by other tools to describe theirs.

Our reasoning behind this is that by moving away from the Haskell REPLs used by DEMO_S5 and SMCDEL and crafting an application that has a more

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

[4]Public repository found at https://github.com/jrclogic/SMCDEL

[5]Haskell source files and installation guide available from `https://w4eg.de/malvin/illc/kripkevis/`

intuitive graphical user interface, we could significantly lower the barrier of entry and make our tool far easier to adopt in educational settings, than its counterparts. Additionally, by making it capable of visualizing each step of the checking process, we believe that it can greatly benefit students and fledgling logicians trying to understand how the various operators of GAL and epistemic logic in general work.

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 of agents can announce. To this end, we will present an algorithm for grouping this infinite set of formulas into a finite set of groups by which states in our structures they are satisfied in through exploring the properties of minimal bisimilar structures (models stripped of any redundant information).

1.4 Structure

This thesis is divided into the following sections:

- This background section dedicated to introducing the reader to the field of dynamic epistemic logic, discussing the topic of model checking as well as its usages and outlining the niche we aim to fill with our own model checking tool.
- A section about the logic this thesis will be working with, Group Announcement Logic, where we will be presenting most of the definitions that will be used in the following sections.
- The next section will contain the majority of our theoretical work behind this thesis, where we discuss the semantics of group announcement logic and will revisit its definitions in order to rework them into definitions that are more readily translatable into algorithms which we will use in the implementation of our model checker.
- We will then translate these definitions into algorithms presented through pseudocode in another section, discussing any differences between the logical semantics and our algorithms.
- After this, we will present the results of our work in the form of a fully-functional graphical model checking tool for GAL, that also doubles as a teaching aid. Here we will present its various features and our rationales for implementing them.
- Once we have finished our high-level introduction and presentation of our tool, we will then transition into a more in-depth discussion of how the

features from the previous section were implemented, as well as our choices of technologies and libraries and which advantages and disadvantages they brought.

- Finally, we will recap with a discussion of our results and what potential future work could be done to improve our tool further.

2 Group Announcement Logic

In this section we will introduce and discuss the various concepts and features of epistemic logic, starting by formally defining and explaining the most central concepts we will be working with. We will then present the language of Group Announcement Logic and aim to give the reader an understanding of the semantics behind it, before we transition into our own theoretical work on exploring its group announcement operator in the next chapter. This will lay the foundations for our implementation of the model checking tool that will be presented in the second half of this thesis.

2.1 Models

In our introduction we briefly mentioned the concept of Kripke structures and epistemic models. These models are structures consisting of states, typically representing 'possible worlds', and agents which may or may not be able to distinguish them. Which states our agents are incapable of distinguishing is represented by an equivalence relation consisting of pairs of states this agent is unable to tell apart and a function from each agent in our model, to that individual agent's equivalence relation. These models also contain a set of boolean propositions, typically describing properties which may or may not hold in each state, as well as a valuation function, which for each proposition in our model, returns the set of states where the given proposition holds.

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

Definition 2.1 (Models)**.** *Given a group of agents A, and a set of propositions P, a model $\mathcal{M}$ is a structure $\mathcal{M} = (S, \sim, V)$, where*

- *S is a set of states*
- *$\sim$ is a function from every agent $a \in A$ to a's equivalence relation, denoted $\sim_a$, such that $\sim_a \subseteq S \times S$*
- *$V(p) \subseteq S$ 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 $s \sim_a t$, for some $a \in A$ to denote 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 $(\mathcal{M}, s)$ to specify that we are referring to state s in the model $\mathcal{M}$, commonly known as a pointed model. This is the form that we will use when we are discussing the properties of a model plus a designated 'actual' state.

A simple example of an epistemic model could be to imagine 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 can 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 exist at least two possible worlds, or states; one where p is true, and one where p is false which they are unable to distinguish. If we label these states as s_0 and s_1, we end up with the model shown in Figure 2.1, where we use $\neg p$ to signify that p is not true in $(\mathcal{M}, s_1)$.

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

Figure 2.1: A basic model with two states

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.

2.2 Language and semantics

Continuing with the model shown in Figure 2.1, we might want to express certain properties such as p being true in $(\mathcal{M}, s_0)$, for this we would write $(\mathcal{M}, s_0) \models p$, or that $\mathcal{M}$ *satisfies* p at s_0. More formally, $(\mathcal{M}, s_0) \models p$ holds *iff* $s_0 \in V(p)$ where if we go back to Definition 2.1, we can see that V is a valuation function for propositions, meaning that it returns the set of states in $\mathcal{M}$, where p holds.

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

Definition 2.2 (The language $\mathcal{L}_{GAL}$)**.**

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

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

Definition 2.2 defines the language of $\mathcal{L}_{GAL}$. 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, $\models$. When some pointed model $(\mathcal{M}, s)$ is not in the satisfaction relation with some formula φ (i.e, φ does not hold in $(\mathcal{M}, s)$), we write $(\mathcal{M}, s) \not\models \varphi$.

Definition 2.3 (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* $\mathcal{M}, s \models \psi$

$\mathcal{M}, s \models \varphi \rightarrow \psi$ *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* $s \sim_i t$, $\mathcal{M}, t \models \varphi$

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

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

Model updates, ($\mathcal{M}|\varphi$), and the language of epistemic logic, ($\mathcal{L}_{el}$), will be covered later.

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 $\mathcal{M}, s \models p$), then as we can see in Definition 2.3 we do this by checking whether the state s is an element in the set of states where p is true, more formally if $s \in V(p)$.

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 shown in Definition 2.3. Implication, $\rightarrow$, can be somewhat loosely translated to 'if a, then b'. If a does not hold however, then the implication automatically holds regardless of the valuation 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 boolean 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, bringing us into 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 of the current state, their scope is also limited to that current state, whereas the K_i operator potentially requires us to check every state that this agent is incapable of distinguishing from our 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 such that $t \sim_a s$, 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. Hence, the agent is not certain whether φ holds.

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 2.3 we use the notation $\mathcal{M}|\varphi, s$ to denote $\mathcal{M}$ 'updated' by φ, meaning $\mathcal{M}$ without all states that do not satisfy φ.

Definition 2.4 (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*

$$\begin{aligned} S' &= \{s | s \in S \text{ and } \mathcal{M}, s \models \varphi\} \\ \sim'_a &= \ \sim_a \cap (S' \times S') \ for\ all\ a \in A \\ V'(p) &= V(p) \cap S' \ for\ all\ p \in P \end{aligned}$$

Informally, Definition 2.4 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 as well as the valuation function to that filtered set of states.

The final and most interesting operator in $\mathcal{L}_{GAL}$ is the group announcement operator, $[G]\varphi$. In $[G]\varphi$, 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 2.5 in order to avoid making the 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). It is defined as the statement that the conjuntion of all sets of formulas $\{\varphi_i : i \in G\}$ may be announced without making φ false in the current state. More intuitively, that G is unable to avoid φ coming 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 Definition 2.3 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 is unable to prevent φ from coming true.

Definition 2.5 (Semantics of epistemic logic $\mathcal{L}_{el}$).

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

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

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

$\mathcal{M}, s \models \varphi \lor \psi$ *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* $s \sim_i t$, $\mathcal{M}, t \models \varphi$

where all operators have their semantics defined as in Definition 2.3.

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 2.6.

Definition 2.6 (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 2.7 (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 2.8 (Formula extensions). *For any formula φ and any model $\mathcal{M}$, the extension of φ in $\mathcal{M}$ is denoted $||\varphi||_{\mathcal{M}}$ and defined as*

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

Note that as $[\![s]\!]_a$ contains all states a is unable to distinguish from s, for a to know any formula φ, this means that a's equivalence class for s has to be a subset of φ's extension, as otherwise a would consider it possible that φ does not hold. Even stronger, this idea holds both ways, and we get $M, s \models K_a\varphi \Leftrightarrow [\![s]\!]_a \subseteq ||\varphi||_{\mathcal{M}}$. In contexts where the model is implicit, we will also refer to $||\varphi||_{\mathcal{M}}$ simply as $||\varphi||$.

2.3 Bisimulation

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 concept of bisimilarity, denoted by $\mathcal{M} \leftrightarroweq \mathcal{M}'$.

Briefly explained, two models $\mathcal{M}$ and $\mathcal{M}'$ are bisimilar iff for any state $s \in S$, then for any formula φ, $\mathcal{M}, s \models \varphi$ iff there also exists some $s' \in S'$ such that $\mathcal{M}', s' \models \varphi$, regardless of $\mathcal{M}$ and $\mathcal{M}'$'s structures.

As bisimilarity will be central to our exploration of the semantics of the group announcement operator in the next chapter, we introduce the definition of bisimilarity as follows in Definition 2.9.

Definition 2.9 (Bisimulation). *Given two models $\mathcal{M} = (S, \sim, V)$ and $\mathcal{M}' = (S', \sim', V')$, a non-empty relation $\mathfrak{R} \subseteq S \times S'$ is a bisimulation between $\mathcal{M}$ and $\mathcal{M}'$ 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 $(t', t) \in \mathfrak{R}$;*

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

$$\begin{tikzcd}
& & & & \underset{p}{s_5} \arrow[dl, dash, "b"'] \arrow[dr, dash, "a"] & \\
\underset{p}{s_0} \arrow[r, dash, "{a,b}"] & \underset{\neg p}{s_1} & & \underset{\neg p}{s_3} \arrow[dr, dash, "a"'] & & \underset{\neg p}{s_4} \arrow[dl, dash, "b"] \\
& & & & \underset{p}{s_2} &
\end{tikzcd}$$

Figure 2.2: Two bisimilar, but structurally different models.

Definition 2.10 (Bisimulation contracted models). *A model $\mathcal{M}$ is said to be bisimulation contracted iff there is no model $\mathcal{M}'$ which has fewer states and is bisimilar to $\mathcal{M}$. A bisimulation contraction of a model $\mathcal{M}$, is any minimal model $\mathcal{M}'$, bisimilar to $\mathcal{M}$.*

The reason why this concept is interesting is that since these bisimulation contracted models do not contain any bisimilar states, 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 a choice of these as labeling formulas.

Definition 2.11 (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 $||\varphi_s|| = \{s\}$.*

An important property of these labeling formulas is that the result of publicly announcing a labeling formula will remove all states from our model, except the state this formula uniquely labels. As such, we can reduce our model to any submodel possible, by combining these labeling formulas with disjunctions and announcing the resulting formula. We will additionally be referring to the set of such labeling formulas for a given bisimulation contracted model $\mathcal{M}$ as $F_{\mathcal{M}}$, defined in Definition 2.12.

Definition 2.12 (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 $F_{\mathcal{M}} = \{\varphi_s | s \in S\}$.*

3 Model checking Group Announcement Logic

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 3.6 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.

3.1 Enumeration of announcements in single agent cases

Recall the definition of the semantics behind $\mathcal{M}, s \models \langle G \rangle \varphi$ in Definition 2.6, 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 2.8 and 2.11 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 2.8 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.

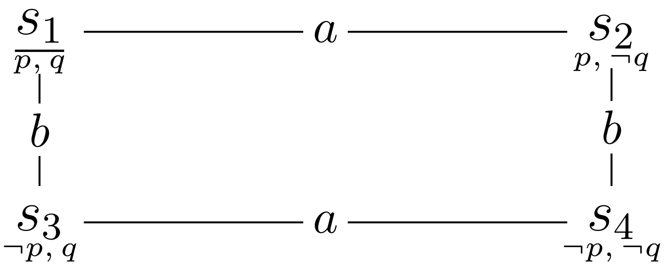


Figure 3.1: A basic bisimulation contracted model.

If we attempt to determine which sets of formulas some agent i can announce in the model in Figure 3.1 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 3.1.

Definition 3.1 (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*
- $[\![s]\!]_a \subseteq ||\varphi||_{\mathcal{M}}$

The reasoning behind these two rules is based on the semantics of the group announcement operators in Definition 2.3, 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, and

(2) has to be satisfied in every state the agent is incapable of distinguishing from that 'actual' state, meaning the agent's equivalence class for that state must be a subset of the formula's extension in our model.

Proposition 3.2. *For every extension $||\varphi||_{\mathcal{M}}$ that satisfies the rules in Definition 3.1 for some agent, any formula with the same extension can be announced by that agent.*

Proof. Assuming we have two formulas φ and ψ, which share the same formula extension, i.e $||\varphi|| = ||\psi||$, in some pointed model $(\mathcal{M}, s)$, then φ is announceable by some agent a, iff ψ is, since the formulas hold in exactly the same states, as per the definition of formula extensions. □

From our definition of formula extensions in Definition 2.8, 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 3.3 (The set of announceable extensions).
The set of announceable formula extensions $\mathcal{A}$ for some agent i, given a bisimulation contracted pointed model $(\mathcal{M}, s)$ is defined as the following:

$$\mathcal{A}_{i,(\mathcal{M},s)} = \{S' \subseteq S \mid [\![s]\!]_i \subseteq S'\}$$

Note that this definition encapsulates both of our rules in Definition 3.1.

For a more practical explanation we will apply these rules to the model in Figure 3.1 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 3.1 to filter out all formula extensions not containing the actual state of our pointed model we get:

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

Examining what a knows in this model, we can see that $[\![1]\!]_a = \{1,2\}$. As such, 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 3.1 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 3.1, 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. Verifying this, we can see that as $\mathcal{M}, s \models K_a\varphi$, announcing this would cause $K_b p$ to be satisfied in the updated model and therefore $\mathcal{M}, s \models \langle\{a\}\rangle K_b p$.

3.2 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 3.1, where we slightly tweak rule 2 to the following:

Definition 3.4 (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:*
$\mathcal{A}_{G,(\mathcal{M},s)} = \{S' \subseteq S \mid \forall s' \in S', \neg\exists t \forall i \in G\ t \sim_i s', t \notin S' \text{ and } s \in S'\}$

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.

3.3 Proof of suitability

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 3.5 (Definition of $\langle G \rangle \varphi$ by Ågotnes et al.[1])**.** *$\mathcal{M}, s \models \langle G \rangle$ iff there is a definable restriction $\mathcal{M}'' = (S'', \sim'', V')$ of $\mathcal{M}$ such that $S'' = \bigcap_{i \in G} C_i$ 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:

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

$$s \in S'' \tag{2}$$

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 3.5, 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:

$$\forall s' \in S' \neg \exists t \forall i \in G, t \sim_i s', t \notin S' \tag{3}$$

$$s \in S' \tag{4}$$

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

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

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 3.6 (Group announcement operators, revised)**.**

> $\mathcal{M}, s \models [G]\varphi$ *iff for all* $Ext \in \mathcal{A}_{G,(\mathcal{M},s)}$ *then* $\mathcal{M}, s \models [\psi_{Ext}]\varphi$
> $\mathcal{M}, s \models \langle G \rangle \varphi$ *iff there exists* $Ext \in \mathcal{A}_{G,(\mathcal{M},s)}$ *such that* $\mathcal{M}, s \models \langle \psi_{Ext} \rangle \varphi$

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

Our reason for revising these definitions is that the original definitions in 2.3 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 3.6 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.

3.4 Algorithm for bisimilarity check

Using the definition of bisimilarity in Definition 2.9 we present our recursive function for checking bisimilarity between states in the same model. Given a set of states *States*, a set of agents *Agents*, a set of labeled edges representing the equivalence relations of these agents and two states s and s' such that $s, s' \in States$, the function (recursive bisimulation check) $rbc(s, s', States, Agents, Edges)$ determining whether s and s' are bisimilar is defined in Algorithm 1.

Before diving into the details of our algorithm, 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. Additionally the equivalence relations of our models are expressed as a set of labeled edges between pairs of states, with each edge's label denoting the set of agents which consider the two states indistinguishable, such that $Edges \subseteq States \times States \times 2^{Agents}$. Our reasons for these changes will be discussed in the implementation section of this thesis.

Algorithm 1 Recursive Bisimulation Check

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

Aside from the structure of our models, our algorithm is fairly similar to the logical definition in Definition 2.9, with the *props* check being equivalent to the *atoms* clause and the *knowledgeCheck* function replacing the *forth* and *back* clauses. We chose to split the pseudocode for checking the *back* and *forth* clauses of the original definition into its own function, which we then reuse by calling it twice with the state parameters flipped. Also note that *rbc* and *knowledgeCheck* are mutually recursive. The pseudocode for this *knowledgeCheck* function is shown in Algorithm 2.

Comparing our algorithm to the original definition of bisimilarity, the main

Algorithm 2 Knowledge Check

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

point of interest is how it is finite; 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. Examining the exit conditions of *rbc* we can see that in the case of comparing two bisimilar states, we end up with one of two outcomes; if the encompassing model has an even number of states, we sooner or later end up clearing all of our states such that $States = \emptyset$, otherwise we end up comparing the last remaining state to itself where $s = s'$.

We also made an auxiliary function for generating the sets of reachable states from a specific state, given a set of states, a set of agents and edges. This function, *neighbours*, builds a set of tuples for each state considered indistinguishable from the given state and the set of agents that consider them indistinguishable. We use this auxiliary function to iterate over the sets of states reachable from each starting state s and s', making sure that for each state $t \sim_a s$, that there also exists a bisimilar state $t' \sim_a s'$ as per the *back* and $forth$ clauses in Definition 2.9.

3.5 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. As bisimilarity is an equivalence relation, it also creates an equivalence class of bisimilar states for each state in our model. For each such equivalence class, $[\![s]\!]$, we choose a designated element and build a map of bisimilar states, linking each bisimilar state in this equivalence class to the state we chose as our designated element, such that $\forall s' \in [\![s]\!] : s \neq s' \Rightarrow (s', s) \in \mathit{bisimMap}$

Once we establish which states are bisimilar to other states in our model, we build our contracted set of states by subtracting the domain of our map of bisimilar states from our original set of states. Thus, assuming that s and s' are bisimilar, we end up with $(s', s) \in \mathit{bisimMap}$, leading to $s' \notin CS$. Note that we are mapping each bisimilar state to the state which they are bisimilar to.

Constraining the edges which make up our equivalence relations is done by limiting our original set of edges to edges where neither of its linked states have been filtered out. We do however also need to make sure that our edges are labeled with the full set of agents it is supposed to be, as in some cases we end up 'combining' multiple edges into one when contracting our models, such as the ones in Figure 2.2. This is done by our *contractEdges* function shown in Algorithm 4. This function iterates through the domain of our map of bisimilar states, 'remapping' any edges going to each bisimilar state so that they instead link to the state which that bisimilar state was bisimilar to. If an edge already exists between these states, we update it so that it is valid for the union of the sets of agents these edges were valid for through calling the *mergeAgents* function.

Skipping the states in the domain of our map of bisimilar states and set of 'visited' states is mostly just an optimization as there is no point in checking states we have already visited and the domain of *bisimMap* is already marked for removal and no longer matters as bisimilarity is transitive, so if $\{(s2, s1), (s3, s1)\} \subseteq \mathit{bisimMap}$, then we have no reason to check which states $s2$ and $s3$ are bisimilar to. Note however that we never check any state against itself, as this would lead to it being removed from the contracted model.

Algorithm 3 Bisimulation contraction

```
function $bisimContract(States, Agents, Edges)$
    $bisimMap \leftarrow \emptyset$
    $visited \leftarrow \emptyset$
    for all $(state)$ $in$ $States$ do
        $visited \leftarrow visited \cup state$
        for all $(otherState)$ $in$ $States \setminus (visited \cup \text{domain of } bisimMap)$ do
            if $rbc(state, otherState, States, Agents)$ then
                $bisimMap \leftarrow bisimMap \cup (otherState, state)$
            end if
        end for
    end for
    $CS \leftarrow States \ \setminus \ (domain\ of\ bisimMap)$
    $CE \leftarrow contractEdges(bisimMap, CS, Edges)$
    $contractedModel \leftarrow (CS, Agents, CE)$
    return $contractedModel$
end function
```

Algorithm 4 Bisimulation contraction of edges

```
function $contractEdges(bisimMap, CS, Edges)$
    $contractedEdges \leftarrow \{e \in Edges \mid e \in CS \times CS\}$
    for all $bisimState\ in\ domain\ of\ bisimMap$ do
        for all $edge\ connected\ to\ bisimState$ do
            $(connectedState, bisimState, ags) \leftarrow edge$
            $originalState \leftarrow bisimMap(bisimState)$
            $existingEdge \leftarrow findEdgeBetween(Edges, bisimState, originalState)$
            if $existingEdge = null$ then
                $existingEdge \leftarrow mkEdgeBtwn(originalState, connectedState, ags)$
            end if
            $mergeAgents(edge, existingEdge)$
            $contractedEdges \leftarrow contractedEdges \cup existingEdge$
        end for
    end for
    return $contractedEdges$
end function
```

3.6 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 3.4. While Ågotnes et al.'s definition in 3.5 is more compact, our definition more closely resembles its pseudocode translation.

Algorithm 5 Generating a coalition's set of announceable extensions

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

As can be seen, the algorithm for generating the set of announceable extensions boils down to 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 3.1. While the if-condition expressing the first rule in Definition 3.1 is fairly self-explanatory, the auxiliary function used to generate our equivalence classes is somewhat more interesting. The *genEqClass* function here works similarly to our previously described *neighbours* function, except it returns the list of states that all agents in the coalition are unable to distinguish from the given state.

Now that we not only have our bisimulation contracted model, but also a way to generate all of the announceable formula extensions for any coalition, we can describe the algorithm for checking group announcement formulas. As these extensions can be considered canonical representations of announcements made by our agents as per the original definition in Definition 2.3, 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 3.6, all our algorithm needs to do is 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 6. Note that our model checking function is split into eight simpler functions, one for each connective, for reasons which will be discussed in our implementation section.

Algorithm 6 Check function for group announcement operator

```
function $check_{[G]\varphi}(state, innerForm, model, coalition)$
    $contractedModel \leftarrow bisimContract(model)$
    $extensions \leftarrow genAnnExts(contractedModel, state, coalition)$
    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
```

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. The *check* function that gets called is implemented as an abstract function overridden by each operator in our system and Algorithm 6 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.

Now that we have presented our algorithm for checking group announcements, we will additionally present our checking algorithms for a few of the simpler operators as well. Starting out, we present Algorithm 7 for checking conjunctions. As all of the non-epistemic operators can be checked in similar fashion with only minor tweaks to the algorithm, we will be skipping the rest of them and instead move on to the knowledge operator.

Algorithm 7 Check function for the conjunction operator

```
function $check_{\varphi \wedge \psi}(state, leftForm, rightForm, model)$
    $leftSatisfied \leftarrow check(state, leftForm, model)$
    if $leftSatisfied$ then
        return $check(state, rightForm, model)$
    else
        return $false$
    end if
end function
```

Algorithm 8 describes how we check knowledge in our model checker. The *getIndishStates* function we use here returns the set of states an agent considers indistinguishable from the given state, which is then looped over as we check whether the 'inner' formula is satisfied in all these indistinguishable states.

Algorithm 8 Check function for knowledge operator

```
function $check_K(state, innerForm, agent, model)$
    $indistinguishableStates \leftarrow getIndishStates(state, agent)$
    for all $indishState$ in $indishtinguishableStates$ do
        if not $check(indishState, innerForm, model)$ then
            return $false$
        end if
    end for
    return $true$
end function
```

Finally, we also present Algorithm 9 for checking public announcements. Note that we also check if the announcement is truthful and immediately return true if this is not the case as false announcements would lead to contradictions per the semantics of public announcements.

Algorithm 9 Check function for public announcements

```
function $check_{[\varphi]\psi}(state, announcement, innerForm, model)$
    if not $check(state, announcement, model)$ then
        return $true$
    end if
    $updStates \leftarrow \emptyset$
    for all $currState$ in $states(model)$ do
        if $check(currState, announcement, model)$ then
            $updStates \leftarrow updStates \cup state$
        end if
    end for
    $updEdges \leftarrow \emptyset$
    for all $edge$ in $edges(model)$ do
        if $connectedStates(edge) \subseteq updStates$ then
            $updEdges \leftarrow updEdges \cup edge$
        end if
    end for
    $updModel \leftarrow updateModel(updStates, updEdges, agents)$
    return $check(state, innerForm, updModel)$
end function
```

4 GALMC

In this chapter we will present our own implementation of a model checker for Group Announcement Logic called GALMC and describe its inner workings. GALMC is intended to be an educational tool which aids students in learning epistemic logic in a more visual manner. Therefore, while most other model checking utilities such as DEMO [10] stick to answering the user's queries with simple yes and no answers, GALMC goes beyond that by showing them not just whether their formulas hold, but also visualizes why and provides 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 building GALMC was how 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[6] 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.

[6]www.jflap.org

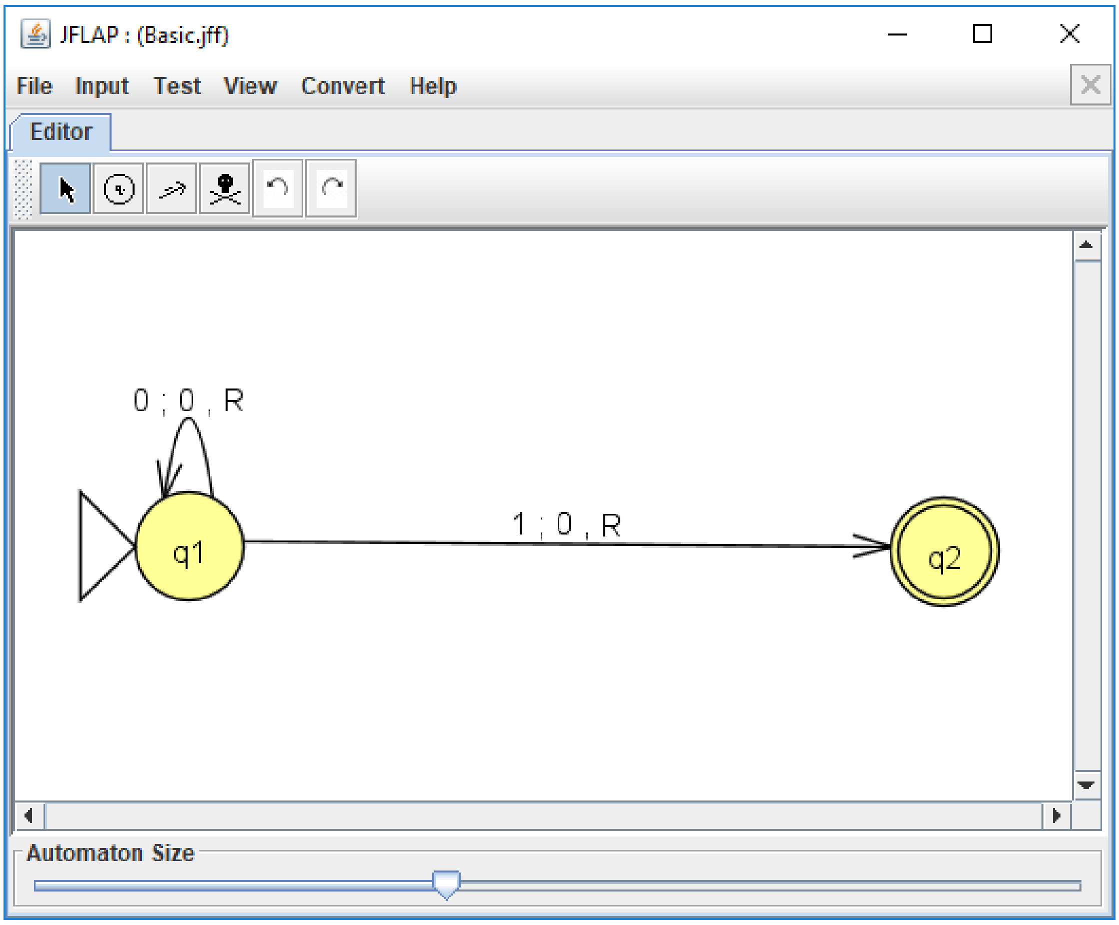


Figure 4.1: A basic two-state turing machine

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. 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.

4.1 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 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 4.2.

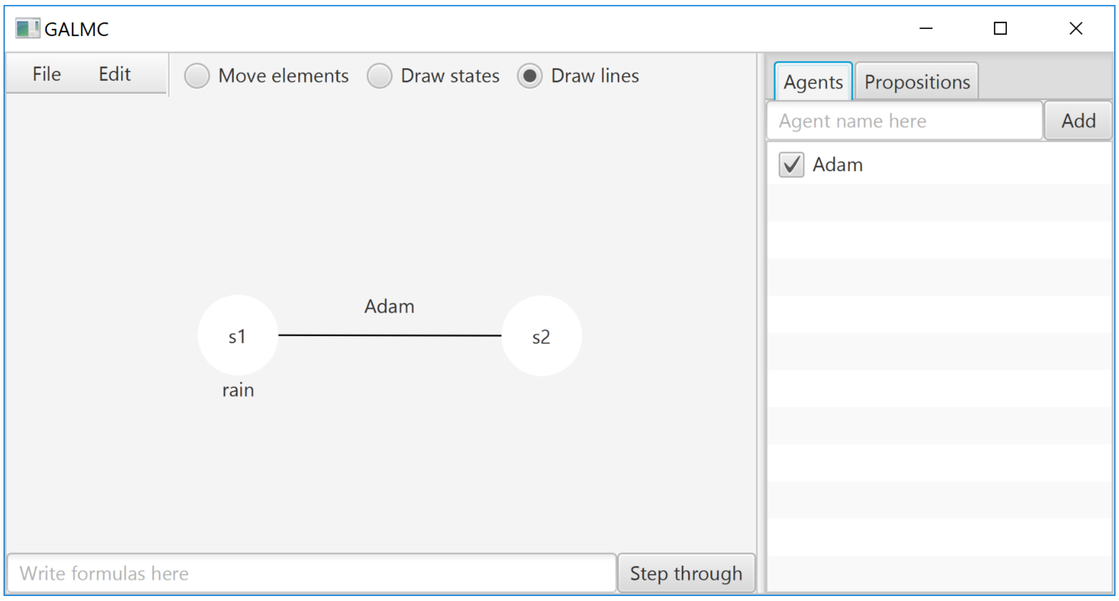


Figure 4.2: A basic model, visualized in GALMC

Figure 4.2 shows how GALMC visualizes the basic model we introduced in Figure 2.1 and as can be seen, the UI of GALMC is fairly close to that of JFLAP, in order to capture its simplicity. The reason for this being to enable the user to draw models in a fashion as close to how they would with pen and paper 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 clean 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; the proposition panel for selecting which propositions should hold in newly drawn states or to update existing ones, and the agent panel which allows the user to similarly determine which agents' equivalence relations should be updated when manipulating edges.

4.2 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 most of the logical symbols used to represent the various operators in GAL cannot be found on normal keyboards, GALMC replaces them with more easily accessible replacements. By the assumption that most of GALMC's userbase would be at least somewhat familiar with programming, these replacements were lifted 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 '(inclusive) or' and 'and'. A full list of operators and their replacement symbols are displayed in Table 1[7]. Note that while announcements are identical to how they are in GAL, the curly braces around the set of agents in group announcements were dropped 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). That said, the editor translates any formula the user types in back into proper legal formulas when displaying them, making it easier for the user to connect GALMC's representations with other material.

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	$[\{ann, bob\}]\varphi$	$[Ann, Bob]\varphi$

Table 1: Table of operators and their symbols in GALMC

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

[7]This list of replacements and a more complete user guide is also available from https://github.com/AndersKaareEide/MCGAL/wiki

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 satisfy the formula, GALMC also lets the user hover over parts of their original formula in order to check the valuation of its subformulas, which can be seen in Figure 4.3 (Note the mouse cursor over the 'clouds' proposition). This enables the user to quickly break formulas apart and see how their various subformulas change the valuation of their containing formula. It also helps visualize the semantics behind each operator in our language in a manner that makes the logic more enjoyable to learn.

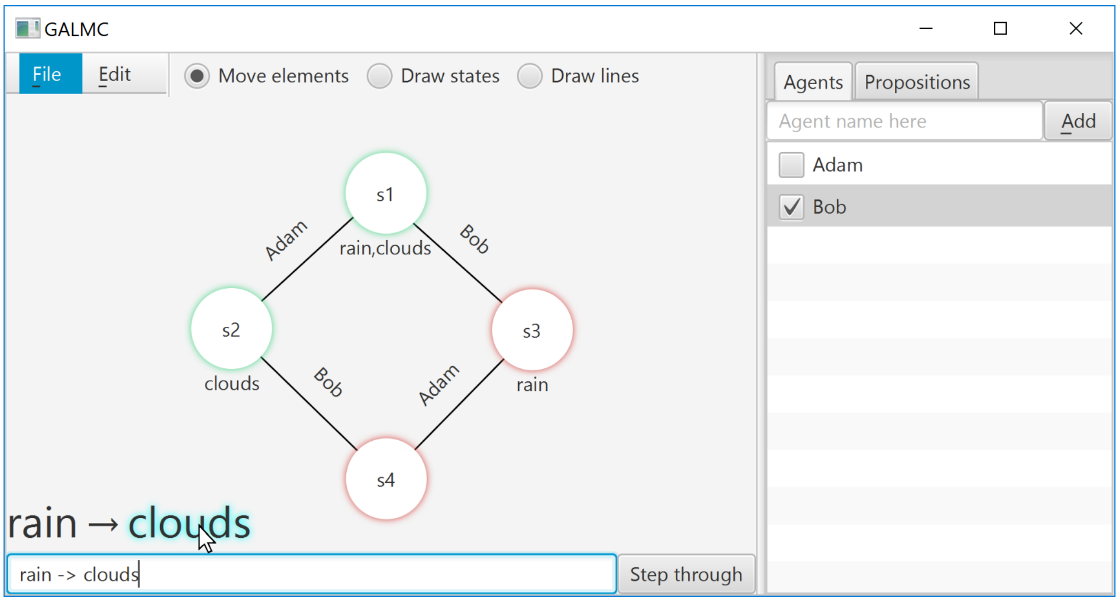


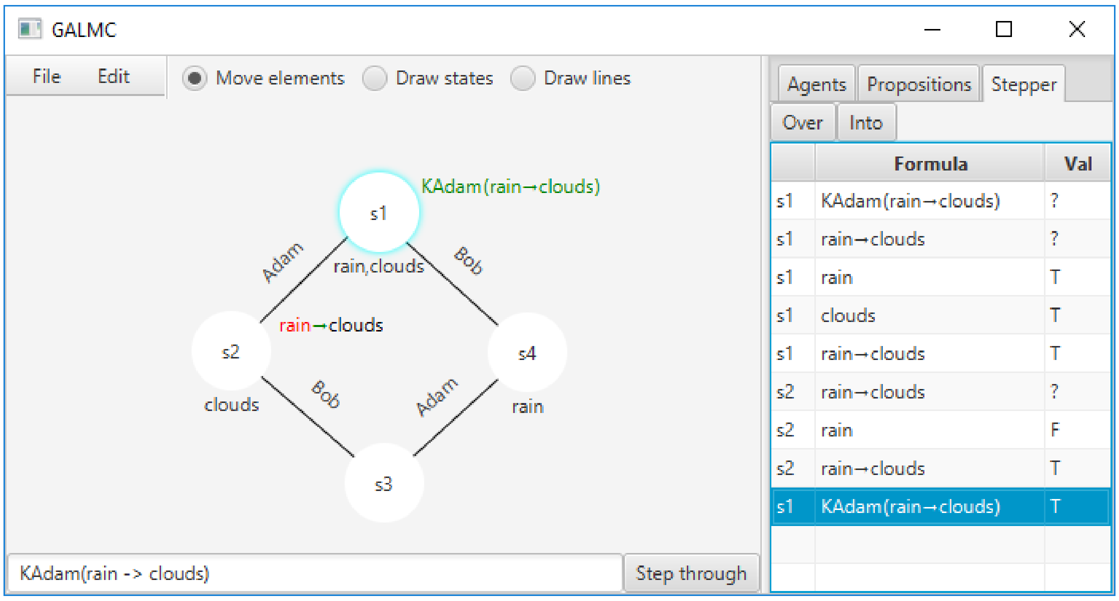
Figure 4.3: Illustration of mousing over interactive formula display

4.3 Visualizing the checking process

One of JFLAP's most powerful features however, is its ability to step through your automatons. As you present your Turing machines 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 tape and which state the machine is currently in. Obviously this feature is highly useful in showing how the user's machines operate, letting 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 makes Turing machines and FSAs much easier to grasp, but it also makes getting them to work correctly a far less frustrating process.

Naturally, GALMC having a similar feature, a way to 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 would greatly improve its usefulness. This is why the tool keeps 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 at the present point of the checking process, and which state it is being checked in. Continuing with the model from our previous figure, Figure 4.4 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 would 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.



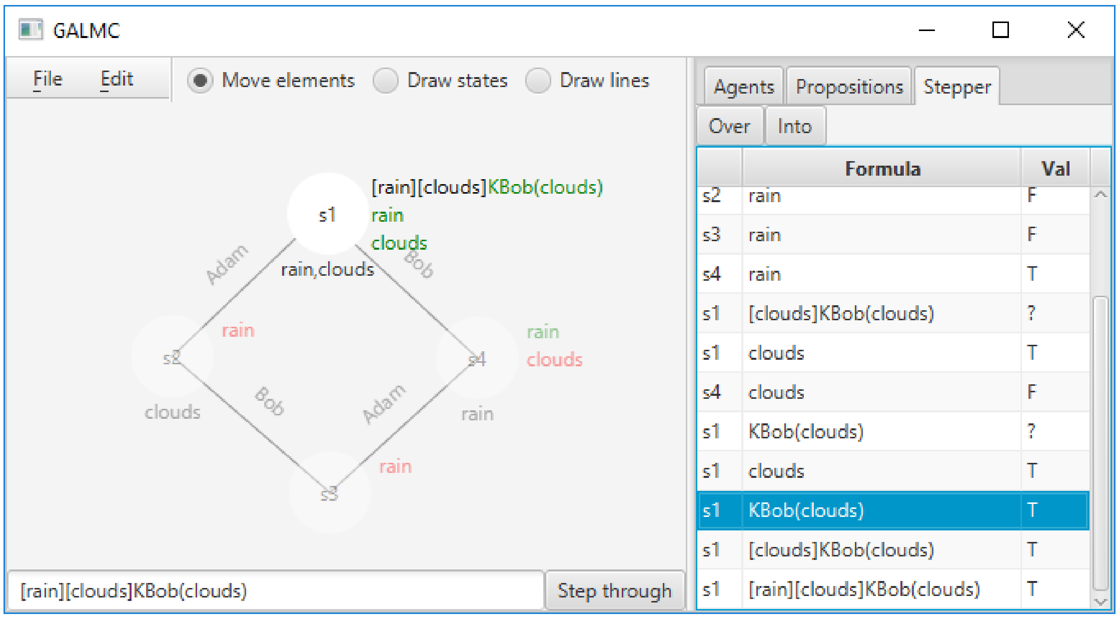
KAdam(rain -> clouds)

	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

Figure 4.4: Illustration of using stepping functionality in GALMC

Returning to our previous comparison between the stepper in JFLAP and our adaptation for GAL, 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. GALMC represents 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 4.5. Note that GALMC also generates formula labels for which subformulas 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.



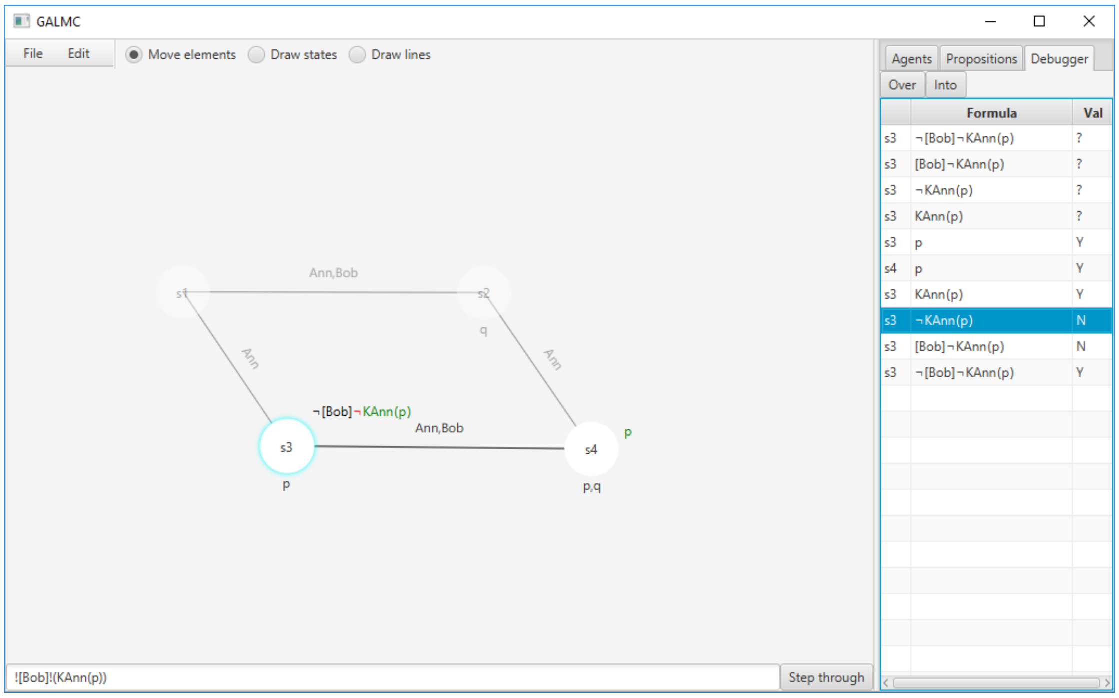
[rain][clouds]KBob(clouds)

	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

Figure 4.5: Visualization of the effects of chained public announcements in GALMC

4.4 Visualizing group announcements

Having presented most of GALMC's features 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 discuss group announcements. As the 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



```
![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

Figure 4.6: Checking a basic group announcement formula in MCGAL

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 subformulas 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, GALMC must 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.

As the tool provides a visualization of the model and keeps track of what the 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 4.6. 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 4.6 we can see from the log that since Bob is able to reduce the model to only states where *p* holds, Ann also knows that *p* holds in this updated model, satisfying our original formula. In this simple illustrative 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.

Last, but certainly not least, GALMC also facilitates saving and loading of models. As our tool is intended to be used in educational settings, making lecturers able to create exercises for their students is highly valuable. One example of such usage would be creating and handing out an incomplete model, where the students would have to ‘complete’ the model by either adding states or changing the indistinguishability relations for their agents in order to give this model some specific property. The lecturer could then load the models that their students have handed in to verify them.

5 Implementation details

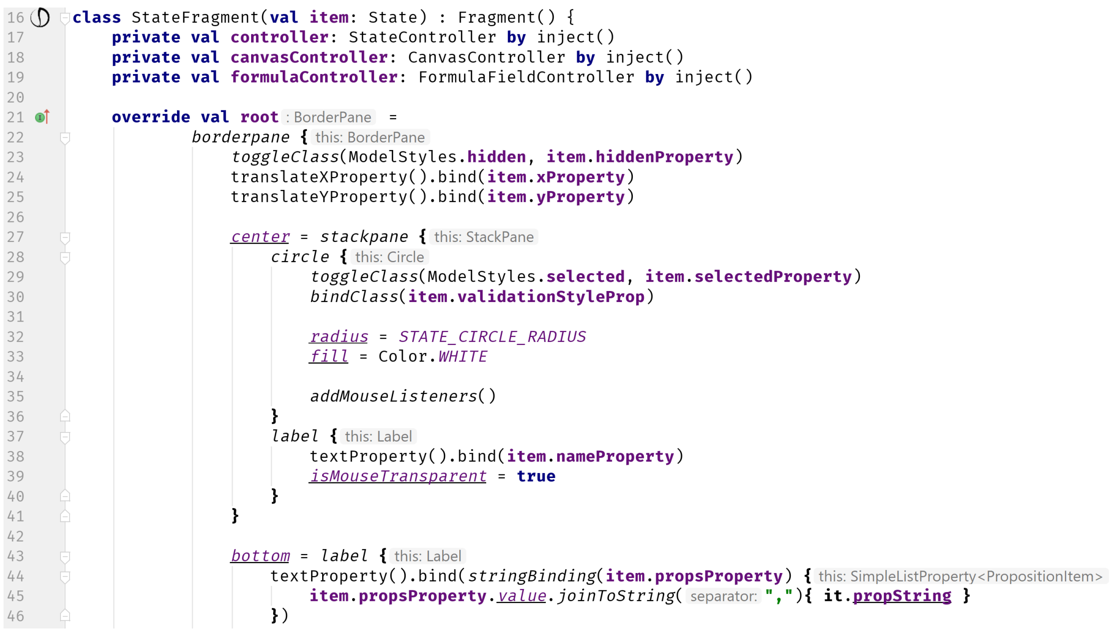
In this chapter we will discuss the various tools and technologies that were used to create our tool, including, but not limited to; choice of programming language, frameworks and libraries and discuss why these were selected.

Starting off with the programming language we went with, our choice was Kotlin. Kotlin is a relatively new language which runs on the Java virtual machine[8] (JVM) that has gotten a lot of attention in the recent years after Google announced they were making it an official language for Android development during their I/O conference in 2017 [11]. While the language has since 'branched out' with JavaScript transpilers and a native compiler, letting developers target different platforms, one of the main strengths of the language is that despite being relatively young, it is able to tap into the vast number of libraries and tools written for the JVM by compiling to Java-bytecode, allowing for full interoperability with Java. Among the other benefits of the language's relatively young age, is that it has been able to draw inspiration from other modern languages giving it features such as type inference, string interpolation and default values. This means that lends 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. As an added bonus, the language is also backed heavily by JetBrains, well known for their suite of IDEs and plugins such as IntelliJ, PyCharm and ReSharper, so the language also has first-class tooling support, making it easy to pick up.

For these reasons, GALMC is implemented in Kotlin, as prior to this project, most of my programming experience came from Java and its many libraries and as such, moving to a language with a far more concise syntax but still being able to lean on my previous experience made Kotlin an excellent tool for this complex system. The choice of library to construct our user interface in fell on the *de facto* standard Java GUI library, JavaFX, or more precisely; a Kotlin wrapper for it called TornadoFX [9]. The reason for choosing TornadoFX is that while JavaFX is a widely used and mature library with powerful features, its usage also tends to result in clunky and verbose code. JavaFX does admittedly provide a solution to this; dumping most of the layout, styling and positioning of components into specialized XML-files. While this helps clean up classes representing UI-components however, it also makes dynamic component generation clumsier and tends to abstract away the component hierarchy in ways that makes it harder to reason around its structure. TornadoFX on the other hand, being a Kotlin library is able to present a much cleaner API by utilizing Kotlin features such as lambda expressions attached

[8]https://www.java.com/en/download/

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



```
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 }
                    })
```

Figure 5.1: Code handling how the UI components representing states are built. Note the conciseness due to implicit contexts.

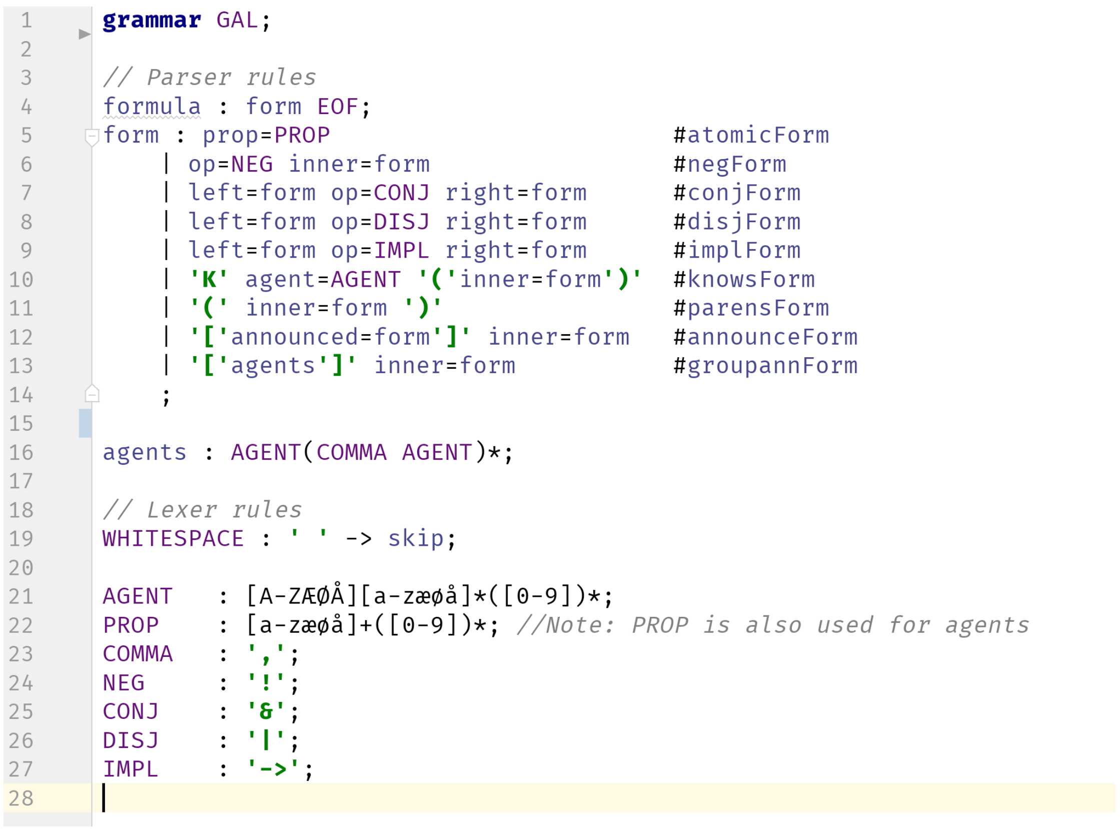
to receivers in order to create composable builder functions which generate your JavaFX component hierarchy in an imperative fashion. Additionally, the wrapper also has excellent and concise shorthands for creating dynamic bindings between UI components and observable sets of data, which we ended up using all throughout the application. However, the primary justification for choosing TornadoFX is how its composable builder functions allow 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 5.1.

5.1 Language and interpretation

Besides creating an understandable user interface another large challenge we needed to solve was how to convert the plain text the user types in into data structures representing formulas to check. For this, we used a tool called ANTLR (ANother Tool for Language Recognition)[10]. ANTLR is a powerful parser generator which, based on a set of grammatical rules, can be used to generate parsers which implement these rules. These grammatical rules can be broadly split into parser and lexer rules. Lexer rules are the low-level rules which define how the tool should convert individual characters or short strings of characters into lexical tokens which

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

are then used by the more high-level parser rules to define how ANTLR should assemble these tokens together again. For a more concrete example, we have the entirety of GALMC's grammatical rules pictured in Figure 5.2. Here we can see how this simple set of lexical rules handle converting symbols into tokens representing the various operators in our language, but also how our set of parser rules define all legal ways of combining these symbols into formulas.



```
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     : '->';

```

Figure 5.2: Grammatical rules for parsing formulas in GALMC with ANTLR.

For comparison, the language described by this ANTLR grammar can also be expressed as the following BNF (terminal symbols are underlined):

$$\begin{aligned}\phi &::= \pi \mid \underline{!}\,\phi \mid \phi\,\underline{\&}\,\phi \mid \phi\,\underline{|}\,\phi \mid \phi\,\underline{->}\,\phi \mid \underline{K}\,\alpha\,\underline{(}\,\phi\,\underline{)} \mid \underline{(}\,\phi\,\underline{)} \mid \underline{[}\,\phi\,\underline{]}\,\phi \mid \underline{[}\,C\,\underline{]}\,\phi \\ C &::= \alpha \mid \alpha\,\underline{,}\,C\end{aligned}$$

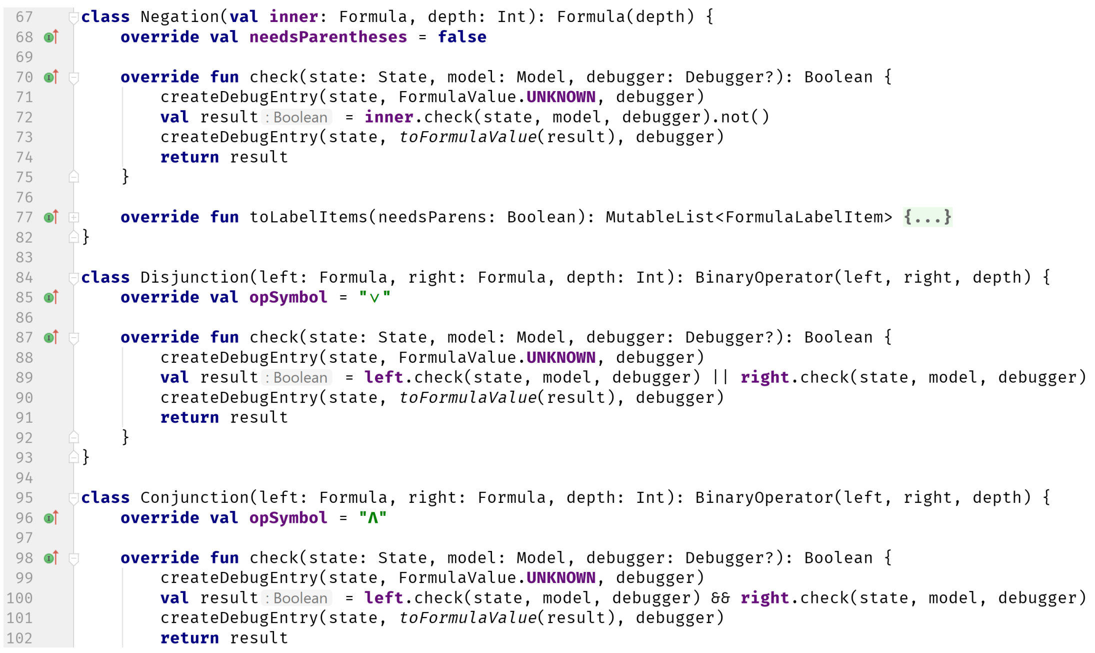
where π (propositions) and α (agents) are sets of terminal symbols characterized as follows: π contains all strings of at least one lower case letter and possibly followed by a string of digits, matching the following regular expression:

$$[\text{a-zæøå}]^{+}[0-9]^{*}$$

α is a single upper case letter followed by a (possibly empty) string of lower case letters, and subsequently a (possibly empty) string of digits, as described by the following regular expression:

$$[\text{A-ZÆØÅ}][\text{a-zæøå}]^*[0-9]^*$$

One thing to note in Figure 5.2 are the symbols used to represent negation, conjunction and disjunction. As previously mentioned, since 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 accessible surrogate symbols. Another small concession I had to make in order to differentiate between agents and propositions, was to require that agent names be capitalized, as I would otherwise have to resort to using different symbols to differentiate between regular public announcements and group announcements.



```kotlin
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
```

Figure 5.3: Snippet showing how various operators are implemented

The parser that ANTLR generates from this simple grammar is then used to convert the user's plain text input into instances of our Kotlin classes which represent the various operators in our language. An interesting thing to note here however is that not only does it create instances of our operators, but it also instantiates them in order of traversal as it builds the formula tree, conserving the structure of this tree as it is rebuilt using our Kotlin-implementations of the operators. One of the more elegant aspects of our design is how these operators were implemented; with each operator extending an abstract formula class and simply overriding it's 'check'-function with their own semantics, as can be seen in Figure 5.3. Going back to our pseudocode algorithms for the various check functions, we can also see the implementation of Algorithm 7 on line 100, note that the '&&' operator is short-circuiting. Also note how the constructors of each operator also takes other formulas as their parameters, allowing us to directly implement the BNF definition of GAL presented earlier through constructor typing alone.

5.2 Kripke structures and UI components

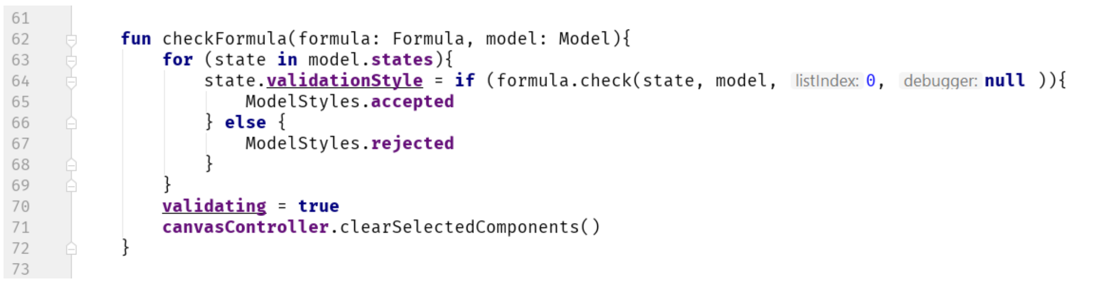
As the focus when making the application was mainly on making it as easy to use as possible while presenting information in a visual manner that is easy to grasp, we ended up making several deviations from the more commonly seen logical definitions of epistemic models. In Definition 2.1 we present $\sim$ as a function from each agent to their respective equivalence relation for every state in the model. In GALMC however, we instead chose to represent these equivalence relations as a set of edges represented as objects consisting of a pair of states and the set of agents that consider this pair of states indistinguishable. The reasons for this ties back into how we wanted to present an interactive view of the models in our application, as TornadoFX, our GUI library makes it far easier to represent each component of our models as concrete objects, which we can then bind UI-components to. Since each edge also has a reference to the set of agents it applies to, this makes it trivial for us to visualize this information as well.

Building on these changes, we additionally made each state aware of which edges it is connected to, simplifying the process of finding indistinguishable states for a given agent, by enabling us to filter the set of edges our state is connected to based on our given agent and return the list of states these edges lead to. While these are deviations from how Kripke structures are more commonly defined, we argue that they make for a much cleaner programmatic representation as they allowed us to simplify both visualization as well as our implementation of the semantics.

We also chose to flip the valuation function by letting states hold a reference to a set of propositions which are satisfied in them, rather than each proposition being linked to the set of states they are satisfied in. Our reasoning here is much the same as for changing how we handle indistinguishability; it allowed us to simplify how we present which propositions are true in each state. We do so by applying a mapping function to these sets of propositions to generate labels in our UI which can then be automatically updated whenever this underlying set of propositions updates. This is far simpler than having to go through the entire set of propositions each time the user updates which propositions a given state satisfies. The code responsible for binding this set of propositions to labels can be seen at the bottom of Figure 5.1. It is these two-way bindings that that are used throughout our implementation which enables the user to directly manipulate their models, as any changes done through the UI are immediately reflected in the internal data structures and vice versa when the tool is checking formulas.

5.3 Model checking

Going back to our presentation of GALMC in the previous chapter, we described that the tool provides two main modes of checking formulas, which we will now discuss the implementation of. The first and simplest of these modes is simply checking which of the states in the user's model satisfy the input formula. With our implementation of formulas, we can simply do this by invoking the formula's check-function on each state in our model, styling the state-component based on the outcome of this checking, as can be seen in Figure 5.4. We also previously described how GALMC allows its user to hover over the various subformulas in order to check how they affect the valuation of their containing formula. We implemented this by tying each symbol in the displayed formula to the subformula it represents and then calling the same check-formula with this subformula.

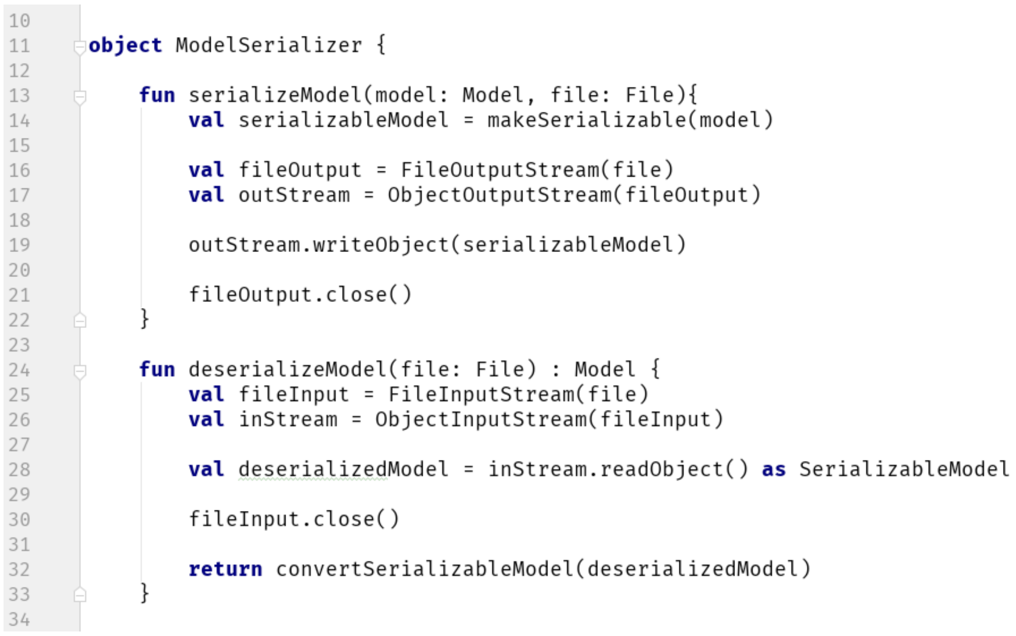


```
    fun checkFormula(formula: Formula, model: Model){
        for (state in model.states){
            state.validationStyle = if (formula.check(state, model, listIndex: 0, debugger: null )){
                ModelStyles.accepted
            } else {
                ModelStyles.rejected
            }
        }
        validating = true
        canvasController.clearSelectedComponents()
    }
```

Figure 5.4: Function responsible for highlighting which states in our model satisfy the input formula

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. In Figure 5.3, this can be seen through our calls to *createDebugEntry*, which is responsible for logging the valuation of each operator in the user's formula before and after every call to this *check* function.

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.



```kotlin

object ModelSerializer {

    fun serializeModel(model: Model, file: File){
        val serializableModel = makeSerializable(model)

        val fileOutput = FileOutputStream(file)
        val outStream = ObjectOutputStream(fileOutput)

        outStream.writeObject(serializableModel)

        fileOutput.close()
    }

    fun deserializeModel(file: File) : Model {
        val fileInput = FileInputStream(file)
        val inStream = ObjectInputStream(fileInput)

        val deserializedModel = inStream.readObject() as SerializableModel

        fileInput.close()

        return convertSerializableModel(deserializedModel)
    }

```

Figure 5.5: Code behind reading and writing models to file

Briefly touching upon saving and loading of models as mentioned at the end of the last chapter, as our models are plain Kotlin (and by extension Java, as Kotlin is fully interoperable with Java) objects, reading and writing them to files is simply a matter of using the built-in tools in the Java library to serialize them and write them to file, as seen in Figure 5.5. While using the JVM library for writing and reading models from files saved us a lot of time, it does however also mean that the models are written in a plain binary format. While it would have been nice to store our models in a more widely-used format so that more complex models could be visualized and edited in more powerful graph editing tools such as Gephi[11], this was left as future work due to time restrictions.

[11]https://gephi.org/

6 Summary

Now that we have covered all of our topics and presented GALMC, a novel educational tool for teaching Group Announcement Logic, it is time to summarize our contributions.

In this thesis we have presented a concrete set of algorithms for not just model checking group announcements, but also for computing bisimulation and bisimulation contracted models. Additionally we presented our own revised definitions for the group announcement operator, with clearer semantics, making them easier to formulate model checking algorithms for, as well as a set of algorithms for expressing these semantics through pseudocode.

Finally, we presented the implementation of these algorithms in our fully realized model checker, GALMC. This model checker is not only capable of checking GAL formulas, but also features a powerful, yet easy to learn graphical user interface, allowing the user to directly manipulate their models in an intuitive manner. Additionally, it also incorporates a variety of useful features for visualizing how these formulas are checked, such as breaking the process down into its individual steps, which can be navigated freely, highlighting the semantics behind each operator in the language.

Although GALMC is fully functional educational aid in its current state, there are also a few features we simply did not have time to implement. One of the simpler additions to our model checker 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.

Other additions include separately displaying the set of formula extensions a coalition can announce, and providing 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 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 missing parentheses and the like might be more challenging.

There were also plans for generalizing GALMC's model serializer in order to be able to export the models as formats beyond its current basic binary format such as GEXF[12] in order to be able to view these models in other tools such as Gephi[13]. 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.

While both user testing and educational impact studies go beyond the scope of this thesis, we believe we have developed a highly useful educational tool that will make it much easier to learn these logics in the future. GALMC is a graphical, user friendly model checker that allows the user to work directly with models and formulas in both single- and multi-agent S5 epistemic logics, PAL and GAL. It allows students to explore increasingly sophisticated multi-agent logics including GAL, which semantics are defined through quantifying over an (infinite) set of possible collaborative announcements. We hope that the development of this model checker for GAL and the concretization of the semantics behind its group announcement operator can also help spark continued interest in the logic itself.

[12]`https://gephi.org/gexf/format/`

[13]`https://gephi.org/`

References

[1] Thomas Ågotnes, Philippe Balbiani, Hans van Ditmarsch, and Pablo Seban. Group announcement logic. *Journal of Applied Logic*, 8(1):62–81, mar 2010.

[2] R. Alur, T. A. Henzinger, and O. Kupferman. Alternating-time temporal logic. In *Proceedings 38th Annual Symposium on Foundations of Computer Science*, pages 100–109, Oct 1997.

[3] Randal E Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys (CSUR)*, 24(3):293–318, 1992.

[4] Edmund M Clarke and E Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logic of Programs*, pages 52–71. Springer, 1981.

[5] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.

[6] Malvin Gattinger. Dynamic epistemic logic for guessing games and cryptographic protocols. *Master's thesis, University of Amsterdam*, 2014.

[7] Malvin Gattinger. *New directions in model checking dynamic epistemic logic*. 9789402810257, 2018.

[8] Thomas A Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Software verification with blast. In *International SPIN Workshop on Model Checking of Software*, pages 235–239. Springer, 2003.

[9] Gerard J. Holzmann. The model checker spin. *IEEE Transactions on software engineering*, 23(5):279–295, 1997.

[10] Jan van Eijck. DEMO-S5.

[11] Paul Miller. https://www.theverge.com/2017/5/17/15654988/google-jet-brains-kotlin-programming-language-android-development-io-2017. *The Verge*, May 2017.

[12] Jan Plaza. Logics of public communications. *Synthese*, 158(2):165–179, Sep 2007.

[13] A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 46–57, Oct 1977.

[14] Wiebe Van der Hoek and Michael Wooldridge. Cooperation, knowledge, and time: Alternating-time temporal epistemic logic and its applications. *Studia logica*, 75(1):125–157, 2003.

[15] Hans Van Ditmarsch, Wiebe van Der Hoek, and Barteld Kooi. *Dynamic epistemic logic*, volume 337. Springer Science & Business Media, 2007.