Creating a sandbox for multi-relation Modal Logic

Børge Eilif Mastberg
Supervisor: Fernando Raymundo Velazquez Quesada

Master's Thesis

Department of Information Science and Media Studies

University of Bergen

May 31, 2023

Acknowledgements

First and foremost I would like to thank my brilliant supervisor Fernando for going above and beyond to facilitate my work on this thesis, and for making Logics such a fascinating and rewarding field to work with.

Next, I need to thank everyone on Lesesal 645 for making the past year tolerable and even at times enjoyable.

Finally, maybe the biggest thanks to my mother, for making me who I am, and my girlfriend Ina, for making every day a better day.

Børge Eilif Mastberg
Nygårdshøyden, May 31, 2023

Abstract

Modal Logic is a framework used to reason about relational structures, with these structures being widely used for modelling various types of phenomena in many different disciplines, including but not limited to computer science, linguistics and philosophy. In this thesis, we aim to create a user-friendly visual web tool for modelling and reasoning about relational structures used within Modal Logic, as well as enabling the user to execute model-changing actions. While existing online tools allows for reasoning about multi-relation models to a certain extent, we have extended the modal language available, as well as increased focus on notions and dynamics in models with multiple relations. We hope that this can be a useful program for both students, teachers and researchers within Modal Logic. More precisely, we expect that this tool can be used by students (e.g to understand the semantic interpretation of formulas), by teachers (e.g., to design interesting and visually compelling examples) and researchers (e.g., to explore and experiment with new concepts within Modal Logic).

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Modal Logic is a tool used within a variety of disciplines, such as computer science, linguistics, and philosophy, to model and reason about different types of phenomena. The models used are relatively simple relational structures, which can be visualized with nodes and edges in a directed graph system. The models can be quite cumbersome and time consuming to visualize (either by hand or using digital tools) and perform calculations on, especially if the graphs contain multiple atomic variables and/or relations (making the graph more extensive and complex). While evaluating simple formulas can be relatively easy, working with more complex formulas may also be a complicated exercise - especially if, in addition, the graphs are large. We hope that a user friendly visual tool for creating and reasoning on models can be of good help both for students, teachers and researchers within various disciplines of Modal Logic.

1.2 Problem Statement

Consider figure 1.1, which is a representation of a common puzzle within Epistemic Logic, the Muddy Children puzzle (*Baltag and Renne*, 2016). While we will not go into detail on the puzzle, the three atoms *ma*, *mb* and *mc* represent each child (a, b and c) and whether they are muddy, while the edges represent epistemic uncertainty of each child. The model is an illustration of how a multi-relation model may look. This type of model can be time consuming to draw and double check, as every atom and every edge has to be carefully placed, and even more complex to perform calculations and operations on. The objective with this puzzle is to examine what each child knows about which one(s) are muddy, and how their epistemic ranges change as they receive more information. We believe that a visual web tool that can be used to create such models, as well as evaluate formulas and perform model transformations on, can be a useful and time-saving supplement for people who study the field of Modal Logic. We can imagine that a teacher preparing lectures or exercises could prepare more examples, and more interesting ones if they had such a tool (or calculator, put a bit crudely) to use for double checking and quality control their calculations. Similarly, researchers and students could benefit from a tool that makes some of the "grunt work" easier and less time-consuming.

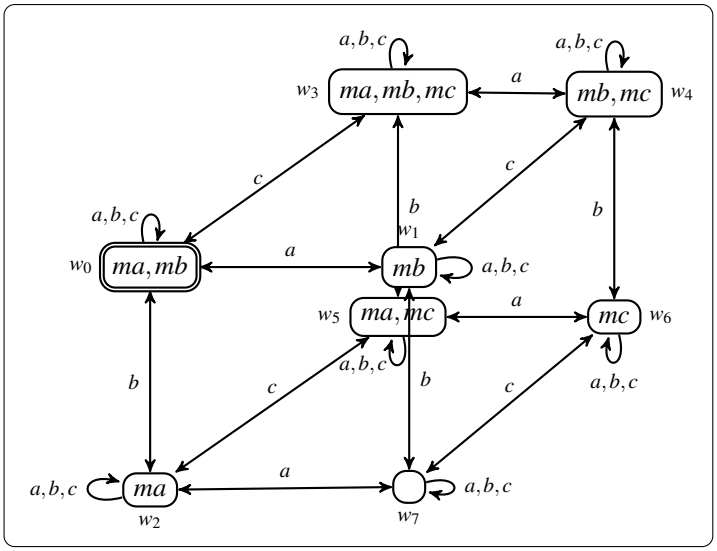

Figure 1.1: The Muddy Children puzzle

Further, we hope to make a tool that can easily be expanded by logicians with some programming knowledge, by adding new modalities, operations or properties they would want to be included. This way, it could become a useful tool for researchers who want to investigate and explore how new concepts may work within the field of Modal Logic.

1.3 Objectives

The main objectives of this project is to write code which can be used to create graphs with multiple (binary) relations, evaluate formulas and perform model change operations within multi-relation Modal Logic, and to create a web tool where this program can be easily accessed by those interested (presumably primarily students, teachers and researchers of Modal Logic). The main objectives which will motivate and guide this process are:

- We want to explore how we can write a program which can be used to create and evaluate models of multi-relation Modal Logic.
- We want to present this program as a web tool with a user friendly interface.

More detailed, the functional objectives are the creation of a program where the user can

1. Build relational models with multiple relations and a number of propositional variables
2. Evaluate formulas of various modal languages
3. Observe the way a given model change action transforms the current model (either one created by the user or standard built-in models)
4. Evaluate formulas that describe the effect of the given change action

In addition, we want to structure the program in such a way that it can be extended with new concepts by any interested programming proficient person.

1.4 Contribution

The aim of this project is to yield a tool which can benefit students and teachers of Modal Logic. As previously mentioned, working with Modal Logic can be time consuming, and a teacher preparing classes could use this tool either to build their models, subsequently update these models, or to double check that their work is correct. Similarly, students can not only evaluate their answers to proposed exercises, but also explore how different modal languages can be used to describe relational structures.

On a more technical level, the main contribution will be software that can be used to build models on which formulas can be evaluated and modal changes can be applied. The aim is to write reusable code that can be extended or modified for anyone who may find it useful.

On yet another ambitious level, if this project is successful in yielding a usable tool for creating and editing Modal Logic models, the result may function as a sandbox for experimenting with these models and perhaps discovering new modalities or interactions which there currently do not exist languages for. If we are successful in making a functional and usable tool, it can become a useful instrument for researchers of Modal Logic.

1.5 Thesis outline

Basics of Modal Logic A discussion of models, language and properties of Modal Logic, as well as a description of related work.

Software A description of the programming language and libraries used to build the program, choices taken when it comes to data structures and representation of models, and how the program works.

Discussion A presentation of what we found working on this project, reflections on choices we made and potential application of the program, contributions, conclusions and potential further work.

Chapter 2

Basics of Modal Logic

2.1 Modal Logic

Modal Logic is a powerful tool for using the framework of relational structures to reason about a variety of phenomena. A relational structure consists of a set of worlds (or states, nodes, points), valuation of these worlds (i.e., a description of their local properties), and relations between them. Although the relations can be of different arity, binary relations are already useful for representing interesting phenomena. This structure can be used, for example, to reason about attitudes towards information within a multi-agent system, where the worlds represent epistemic possibilities and the relations are interpreted as uncertainty (*Epistemic Logic*). We can also understand the worlds as points in time and the relation as temporal precedence (*Temporal Logic*), or we can understand the worlds as objects or events (or anything else that can be subject to choice) and the relation to represent preference ordering (*Preference Logic*), and so on. In this section, we will provide a basic outline of these relational structures that can be described with modal languages, then continue to define a modal language to describe and reason about these structures. Our presentation of Modal Logic is based on *Blackburn et al.* (2001).

2.1.1 Models

We define a one-relation model as a tuple M consisting of a domain W, a single relation R and a valuation V (*Blackburn et al.*, 2001): The domain W is a non-empty set of worlds (also named states or nodes among others - in this thesis we will refer to them as worlds or nodes), while relation R is a single binary relation on W. The valuation function V maps truth values of propositional atoms to each world in W.

$$M = \langle W, R, V \rangle$$

Figure 2.1 shows a simple example of a model, where the domain consists of two worlds $\{w_0, w_1\}$, the relation consists of a single transition (w_0, w_1) and p is true in w_1.

A *pointed model* is a model together with a world, which denotes the model's *evaluation point*:

$$(M, w) = (\langle W, R, V \rangle, w)$$

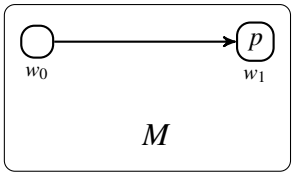

Figure 2.1: A model

Figure 2.2 is the same model as before, with the evaluation point (double circled) set to w_0. In the next section, we will define a modal language, and describe how we use this language to evaluate formulas from this evaluation point.

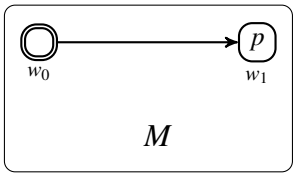

Figure 2.2: Pointed model

2.1.2 Language

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid \Diamond\varphi \mid \Box\varphi$$

In addition to the basic propositional language (atom, negation and disjunction), we define existential - $\Diamond$ - and universal - $\Box$ - quantifiers. The "diamond" (possibility) operator, along with any formula φ, is intuitively understood as "there is at least one world that can be reached from the evaluation point and in which φ is true". The "box" (necessity) is understood as "φ is true in every world that can be reached from the evaluation point".

Formally, we define them as follows:

- $(M,w) \Vdash \Diamond\varphi$ if and only if there is $u \in W$ such that Rwu and $(M,u) \Vdash \varphi$
- $(M,w) \Vdash \Box\varphi$ if and only if every $u \in W$ is such that if Rwu then $(M,u) \Vdash \varphi$

As we can infer from these definitions, although the $\Diamond$ and $\Box$ modalities are existential and universal quantifiers, they do not quantify over the whole domain; rather, they work only on the set of worlds that can be reached from the current evaluation point.

To illustrate, we use the pointed model in figure 2.3

$$(M,w_2) \Vdash \Diamond p$$
$$(M,w_2) \Vdash \Diamond q$$

From the evaluation point, there is at least one world where p is true, and there is at least one world where q is true.

$$(M,w_2) \Vdash \Box p$$
$$(M,w_2) \Vdash \neg\Box q$$

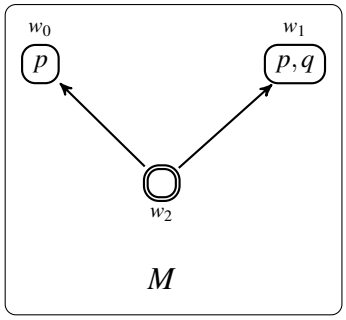

Figure 2.3: Pointed model with three worlds and two variables

p is true in every world that can be reached from the evaluation point, but it is not the case that q is true in every world that can be reached from the evaluation point.

$$(M,w_2) \Vdash \Box\Box p$$
$$(M,w_2) \Vdash \Box\Box q$$
$$(M,w_2) \Vdash \Diamond\Box(p \vee q)$$

All the above formulas are true, as the $\Box$-operator always will be true in a world with no successors (the box definition is formulated like an implication: "if... then", while the diamond definition is a conjunction).

Extended language

For reasoning about certain phenomena, the basic modal language that we have presented in this section may not be sufficient. In the literature, several further modalities have been proposed to enrich the language. Here we mention two of them: the *global* modality and the *difference* modality. These modalities (and especially the global modality) is motivated in *Blackburn et al.* (2001) by using an example from reasoning about computer networks. We take φ to mean "server 1 is active" and ψ to mean "server 2 is active". Thus, we can check whether one of or both or none of the servers are active in one state (and in the following state, and in all the following states, and so on). However, if we need to check if, for example, server 2 is active whenever server 1 is active (i.e. $\varphi \rightarrow \psi$) in *every* state, the basic modal language falls short. Thus the need for a global modality, which quantifies over every state (or world) in the domain.

The global modality E evaluates whether there is a world in the domain where a given formula is true (regardless of relations), while the difference modality D evaluates whether there is a world in a domain *which is not the evaluation point* where a given formula is true. Formally,

$$(M,w) \Vdash E\varphi \text{ if and only if there is } u \in W \text{ such that } (M,u) \Vdash \varphi$$
$$(M,w) \Vdash D\varphi \text{ if and only if there is } u \in W \text{ such that } u \neq w \text{ and } (M,u) \Vdash \varphi$$

These two modalities, E and D are still existential quantifiers. However, they are not restricted to the worlds reachable from the evaluation point. The first, E, quantifies over the whole domain, while the second, D, quantifies over the whole domain minus the current evaluation point.

This provides an extended modal language:

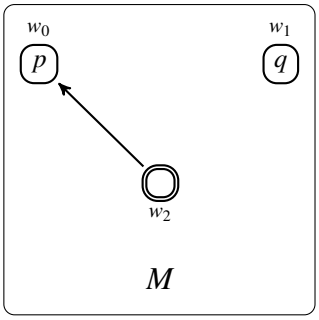

Figure 2.4: Pointed model with three worlds and two variables

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid \Diamond\varphi \mid \Box\varphi \mid E\varphi \mid D\varphi$$

In the model in figure 2.4, our extended language can express more than the basic modal language. For example,

$$(M, w_2) \Vdash Eq$$

tells us that there is in fact a world in the domain where q is true, which cannot be expressed by the basic modal language that depends on transitions. Further,

$$(M, w_2) \Vdash D\neg p$$

expresses that there is a world in the model, which is not the evaluation point w_2, where p is false. This is also something that that is outside the capabilities of the basic modal language.

There is a formal way to prove that operators E and D provides more than the basic modal language can, namely using notions of invariance (e.g. bisimulation), but this will not be discussed here.

2.2 Modal Logic with multiple relations

In many cases, it makes sense for the model to contain more than one relation. For example, within Epistemic Logic, which attempts to reason about information and attitudes towards information among a group of agents, we need a single relation for each agent to express the differences of the respective agents' epistemic range.

2.2.1 Models

The models for multiple-relation Modal Logic are quite similar to the already defined models, the difference is that we deal with multiple binary relations. The domain W and the valuation V are exactly the same, and the relation is now a set of relations $\{R_i | i \in G\}$, where G is the set of relation labels (under some interpretations named agents).

$$M = \langle W, \{R_i | i \in G\}, V \rangle$$

We use pointed models to evaluate formulas, which are defined in the same manner as before:

$$(M,w) = (\langle W, \{R_i | i \in G\}, V\rangle, w)$$

In the pointed model in figure 2.5, we have two relations, R_a and R_b. R_a consist of one single reflexive transition (w_0, w_0), while R_b is comprised of two symmetric transitions (w_0, w_1) and (w_1, w_0) and a reflexive relation (w_0, w_0).

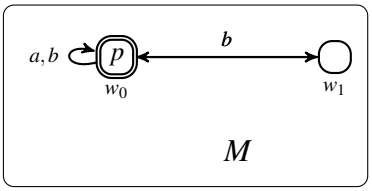

Figure 2.5: Pointed model with two relations

2.2.2 Language

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid \Diamond_i \varphi \mid \Box_i \varphi$$
$$i \in G$$

The multi-relation modal language is extended by adding sub-indexes to the box and the diamond, representing the individual relations they follow. In figure 2.6, the following (among others, of course) formulas are true:

$$(M, w_0) \Vdash \Diamond_b p$$
$$(M, w_0) \Vdash \Diamond_b \neg p$$

Using relation b, from the evaluation point one can reach a world where p is true, and relation b can reach a world where p is not true.

$$(M, w_0) \Vdash \neg\Box_b p$$
$$(M, w_0) \Vdash \Diamond_b \Box_b p$$

p is not true for every world that can be reached by relation b, but there is a world that relation b can reach where p is true for every successor.

$$(M, w_0) \Vdash \Diamond_a p$$
$$(M, w_0) \Vdash \neg\Diamond_a \neg p$$

Relation a can reach a world where p is true, and relation a can not reach a world where p is false (which can be rewritten as $\Box_a p$).

$$(M, w_0) \Vdash \Box_a p$$
$$(M, w_0) \Vdash \Box_a \Diamond_a p$$
$$(M, w_0) \Vdash \Diamond_a \Box_a p$$

p is true for every world a can see, every successor of the evaluation point for relation a has at least one successor where p is true, there is a successor where p is true for every world that can be reached (and so on and so on - we could go on for ever).

Extended language

Once a relation is present in a model, one can use use it to define further relations. For example, given a relation R, one can define its complement (all pairs in $W \times W$ that are not in R) and also its converse (all pairs in R, but with their order reversed). With these newly defined relations, we can use existential or universal quantifiers in the same way we can on the relations the model provides explicitly.

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid \Diamond_i\varphi \mid \Box_i\varphi \mid \not\Diamond_i\varphi \mid \overleftarrow{\Diamond}_i \quad i \in G$$

Humberstone (1983) proposes a modal inaccessibility operator which describes which worlds in the domain of a pointed model *can not be reached* from the evaluation point. Humberstone does not discuss in detail the motivation of this modality, apart from a purely technical perspective, but mentions that it could be an interesting modality in *Deontic Logic*. We define this as an existential quantifier along a "Cartesian relation" - $W \times W$ minus agent i's relation R_i: $(W \times W) \setminus R_i$. The semantic interpretation of the modality is

$(M,w) \Vdash \not\Diamond_i\varphi$ if and only if there is $u \in W$ such that not R_iwu and $(M,u) \Vdash \varphi$

This inaccessibility modality is a (existential) quantifier working over the complement of relation R_i.

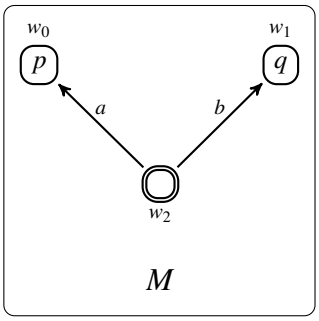

Figure 2.6: Pointed multi-agent model with three worlds and two variables

In the model in figure 2.6, we have $(M,w_2) \Vdash \not\Diamond_a q$ and $(M,w_2) \Vdash \not\Diamond_b p$: There is a world which is inaccessible to a where q is true, and there is a world which is inaccessible for relation b where p is true.

Within *Temporal Logic* (*Goranko and Rumberg*, 2022), which aims to reason about time, and where transitions describes that time moves forward, it makes sense to define a "backwards" modality, i.e. a modality that can describe events in the past. In our context, this would translate to a modality that reverse transitions, and has an intuitive meaning of "there is a world of which the evaluation point is a successor where [something] is true". In Temporal Logic, this diamond version "It has at some time been the case that..." is denoted as P, while H is used for the dual "It has always been the case that...". For our "multi-purpose" multi-agent modal language, we will use a diamond with an arrow going "backwards" - $\overleftarrow{\Diamond}$ - for this *converse* modality. We define it as

$(M,w) \Vdash \overleftarrow{\Diamond}_i\varphi$ if and only if there is $u \in W$ such that R_iuw and $(M,u) \Vdash \varphi$

The converse modality is a (existential) quantifier working over a "reversed" R_i; every (source, target) with a reversed order.

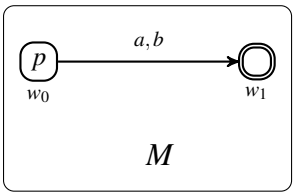

Figure 2.7: Pointed model with two agents

In the model in figure 2.7, we have $(M, w_1) \Vdash \overleftarrow{\Diamond}_a p$ and $(M, w_1) \Vdash \overleftarrow{\Diamond}_b p$, as there for both relation a and relation b is a previous world from the evaluation point (or a world of which the evaluation point is a successor) where p is true.

Interpretations of modalities

Even though we have not mentioned it explicitly when defining them, these modalities may have a variety of different interpretations, depending on what one wants to reason about (it can be information, time, preferences, computer programs etc). The sub-discipline of Modal Logic we have the most experience with is Epistemic Logic, which has been one of the biggest motivations for this project.

Within Epistemic Logic, where the worlds represents possible states of information (e.g. the sun is shining) and the relations represent uncertainty of agent (e.g. agent a considers both a world where the sun is shining and one where it does not shine), $\Box_i \varphi$ is understood as "agent i knows φ". This is because, in a situation where $\Box_i \varphi$ is true, φ is true in every world agent i consider. In Epistemic Logic, one typically changes notation from $\Box_i$ to K_i (to emphasize the knowledge interpretation).

2.2.3 Combining relations

The just presented "i-inaccessible" and "i-backwards" modalities are (existential) quantifiers that work over a relation defined from R_i. Once multiple relations are present, one can also define further relations by combining them. This is particularly useful in fields such as Epistemic Logic, where these newly defined relations *built from several relations* can be used to express some intuitively useful epistemic notions for *groups of agents*.

Literature within Epistemic Logic (*Rendsvig and Symons*, 2021) define some group notions of knowledge within our context of multi-agent Modal Logic. In this section, we will provide three mathematical definitions of such notions, as well as give a more "intuitive" explanation of these concepts (as interpreted within Epistemic Logic).

Everybody knows

Take a set of labels for relations G, and a subset of labels $H \subseteq G$. The relation R_H^{EK} is defined as the union of the relations within H:

$$R_H^{EK} := \bigcup_{i \in H} R_i$$

Using this combined relation, we can introduce modality EK_H with two equivalent definitions:

$$(M,w) \Vdash EK_H\varphi \text{ iff every } u \in W \text{ is such that if } R_H^E wu \text{ then } (M,u) \Vdash \varphi$$
$$(M,w) \Vdash EK_H\varphi \text{ iff every } i \in H \text{ is such that } (M,w) \Vdash \Box_i\varphi$$

This is a universal quantifier (like the $\Box$-modality), with the added condition that φ is universally true in the union of relations, and not only on a single relation.

Interpreted epistemically, we call this modality *everybody knows*: $EK_H\varphi$ holds whenever every agent in H knows φ (as we mentioned earlier, the epistemic interpretation of $\Box_i\varphi$ is that agent i knows φ - φ is true in every world i consider to be possible.)

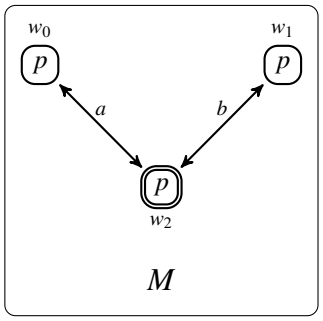

Figure 2.8: Pointed model with three worlds and two agents

In model M in figure 2.8, agents a and b both know p to be true. We can probably deduce that everybody knows p without performing any mathematical operations, but a union of the relations will show that every Rwu is such that p is true. Thus, everybody knows p, or $EK_{\{a,b\}}p$.

Common knowledge

Take a set of labels for relations G, and a subset of labels $H \subseteq G$. The relation R_H^{CK} is defined as the transitive closure of the union of the relations within H (or, the transitive closure of R_H^{EK}):

$$R_H^{CK} := \Big(\bigcup_{i \in H} R_i\Big)^+$$
$$R_H^{CK} := (R_H^{EK})^+$$

Using this relation, we can introduce modality CK_H:

$$(M,w) \Vdash CK_H\varphi \text{ iff every } u \in W \text{ is such that if } R_H^{CK} wu \text{ then } (M,u) \Vdash \varphi$$

As with everybody knows, this is also an universal quantifier, but on a combined relation which is the transitively closed union of relations.

Interpreted epistemically, $CK_H\varphi$ describes a situation where φ is *common knowledge*, i.e. not only does everybody know φ, but everybody knows that everybody knows φ, and everybody knows that everybody knows that everybody knows φ and so on:

$$CK_H\varphi\text{: } EK_H\varphi \text{ and } EK_HEK_H\varphi \text{ and } EK_HEK_HEK_H\varphi \text{ and...}$$

The notion of *common knowledge* describes the situation where not only does everybody in the group know that the sun is shining, but everybody also knows that everybody else knows it (and everybody knows that everybody knows that everybody knows it and so on...). In a situation where a group of people is outside on a hot summer day, the fact that the sun is shining is common knowledge because everybody can see that the others are also outside enjoying the nice weather.

Distributed knowledge

Take, again, a set of labels for relations G, and a subset of labels $H \subseteq G$. The relation R_H^{DK} is defined as the intersection of the relations labeled in H:

$$R_H^{DK} := \bigcap_{i \in H} R_i$$

Using this combined relation, we can introduce modality $DK_H\varphi$:

$$(M,w) \Vdash DK_H\varphi \text{ iff every } u \in W \text{ is such that if } R_H^{DK}wu \text{ then } (M,u) \Vdash \varphi$$

This is also a universal quantifier, but on the intersection of relations in H.

Interpreted epistemically, $DK_H\varphi$ describes a situation where, if a group of agents combine their information, they can potentially know φ. This is called *distributed knowledge*: DK_H holds whenever every agent in H has the potential to know φ if every agent in H share their information with the rest of the group.

For example, for some reason, one agent may know that if it is raining, the streets will flood, and another may know only that it is raining. If they pool their knowledge, together they can get to know something they do not know individually: the streets will be flooded.

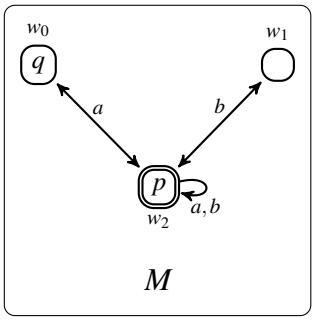

Figure 2.9: Pointed model with three worlds and two agents

The model above illustrates a situation where agent a considers $(p \wedge \neg q) \vee (\neg p \wedge q)$, i.e., she knows that exactly one of the atoms is true, while agent b considers $(p \wedge \neg q) \vee (\neg p \wedge \neg q)$, i.e., she knows that q is false. The intersection of agent a and agent b's relations results in a relation where only $(p \wedge \neg q)$ can be true: $DK_{\{a,b\}}(p \wedge \neg q)$

2.3 Model change

Within many sub-disciplines of Modal Logic, it can be useful to inspect how model change affects not only a given model, but also the one that results after a certain transformation takes place. In this way, if the model is representing a certain situation, one can reason about consequences of *actions* that affect the situation (therefore transforming the model). In this section, we will describe two such dynamics, and focus on an epistemic interpretation of these.

2.3.1 Public announcements

Take any model

$$M = \langle W, \{R_i | i \in G\}, V \rangle,$$

then introduce any formula with which to transform the model:

$$\chi!$$

The new and transformed model $M_{\chi!}$ with a new domain, a new set of relations and a new set of valuations is defined as follows:

$$\begin{gathered} M_{\chi!} = M' = \langle W', \{R'_i | i \in G\}, V' \rangle \\ W' := \{w \in W | (M, w) \Vdash \chi\} \\ R'_i = R_i \cap (W' \times W') \\ V'(p) := V(p) \cap W' \text{ for } p \in P \end{gathered}$$

Intuitively, this operation removes every world in which χ is false, and removing also every transition connected to the removed worlds. More technically, the new domain consist of every world in which χ is true, the new set of relations is the intersection of the old set of relations and the Cartesian product of the new domain, and the new valuation $V'(p)$ is the intersection of the original V and W' for every p in the set of atoms P.

We can use this operation to create two new modal operators:

$$\begin{gathered} (M, w) \Vdash \langle \chi \rangle \varphi \text{ if and only if } (M, w) \Vdash \chi \text{ and } (M_{\chi!}) \Vdash \varphi \\ (M, w) \Vdash [\chi] \varphi \text{ if and only if } (M, w) \Vdash \chi \text{ implies } (M_{\chi!}) \Vdash \varphi \end{gathered}$$

Interpreted epistemically, this operation is called a *public announcement*, in which the agents receive truthful information publicly. After the announcement of χ, every agent considers possible only worlds where χ was true in the initial model M, intuitively causing every agent to know χ.[1]

In the example in figure 2.10, we show the transformation of a pointed model after formula p is publicly announced.

[1] Still, note that the intuition is not fully correct: when the formula χ involves modalities, it is possible for the formula to be false in some world of the new model. The classic example is the announcement of a formula $p \wedge \neg K_a p$, indicating that p is true and agent a does not know it. After its announcement, the agent will know p, but it cannot know $\neg K_a p$, as now it knows p. Thus she cannot know the whole $p \wedge \neg K_a p$.

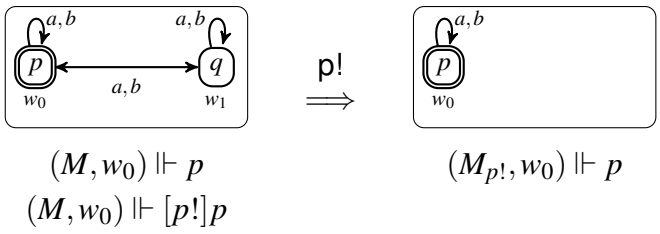

Figure 2.10: Initial pointed model (M,w) and pointed model $(M_{p!},w)$ after announcement of p

2.3.2 Communication

Take any model

$$M = \langle W, \{R_i | i \in G\}, V \rangle$$

Then perform an intersection action $C!$[2] from relation label subset $H \in G$

$$C!_H$$

The original model and transformed model has identical domains and valuations, only the set of relations are affected by the intersection.

$$M^! = \langle W, \{R_i^! | i \in G\}, V \rangle$$
$$R_i^! := R_i \cap \bigcap_{j \in H} R_j$$

The relations of H are intersected, and then every relation $i \in G$ is intersected with the result.

We can use this operation to create a new modal operator:

$$(M,w) \Vdash [C!]_H \varphi \text{ if and only if } (M^!,w) \Vdash \varphi$$

Interpreted epistemically, we call this operation *communication*. Every agent in H share everything they know with the rest of the group, eliminating uncertainty (transitions) that can be ruled out.

The attentive reader may have noticed a connection between the previously described notion of distributed knowledge and communication; agent communicating will, intuitively, turn distributed knowledge into explicit knowledge: everybody in the group will know what they knew distributively before.[3]

$$R_i^! := R_{H \cup i}^{DK}$$

If we revisit our example from distributed knowledge and figure 2.9, we can illustrate the results of the communication operation on a model where p is distributed knowledge. In our example, the whole group (agents a and b) now communicates.

As we can see in figure 2.11, the only transitions preserved by the communications are the reflexive transitions of both agents at the evaluation point, making this the only point in the model each of them consider to be possible.

[2] We use $C!$ as a symbol for the model change action, not a formula, relation or set of agents

[3] Again, this intuition is not fully correct, due to situations like the previously mentioned formula $p \wedge \neg K_a p$.

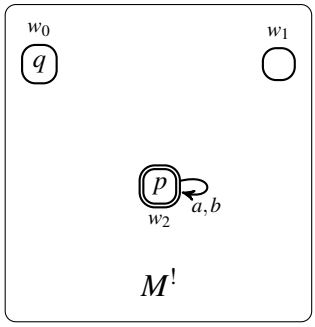

Figure 2.11: Pointed model after communication

2.4 Models with particular properties

As we discussed earlier, relational structures can be used to represent several different phenomena. In order to let these representations model the phenomena appropriately, one may want to require the relations to satisfy certain properties. For example, when worlds in a model are understood as points in time and the relation is interpreted as temporal precedence (Temporal Logic), it makes sense for the relation to be transitive. Similarly, if the worlds are understood as epistemic possibilities and the relation is interpreted as representing uncertainty (Epistemic Logic), it makes sense for the relation to be reflexive and symmetric.

We use the following (standard) definitions of reflexivity, symmetry and transitivity:

A relation R_i is reflexive if and only if for every $w \in W$ there is R_iww

A relation R_i is symmetric if and only if for every $w,u \in W$, if there is R_iwu then there is R_iuw

A relation R_i is transitive if and only if for every $w,u,v \in W$, if there is R_iwu and R_iuv then there is R_iwv

2.5 Related work

We are working along the lines of, as well as inspired by, a couple of existing web tools for creating and evaluating models of Modal Logic.

Modal Logic Playground (MLP - http://rkirsling.github.io/modallogic/) (*Kirsling*, 2021a), is "A graphical semantic calculator for modal propositional logic", where the user can create single-relation models by adding worlds and transitions, and use up to five propositional variables. It has a clean and simplistic design(see figure 2.12), and lets the user save their model by generating a link which the user can use later to return to their work later.

Modal Logic Playground is written using JavaScript along with libraries D3, MathJax and Bootstrap. For parsing formulas, is uses FormulaParser (*Kirsling*, 2017). Its functionality allows the user to create models with one relation, and evaluate formulas with the basic modal language (as described in section 2.1, excluding modalities E and D from the extended language).

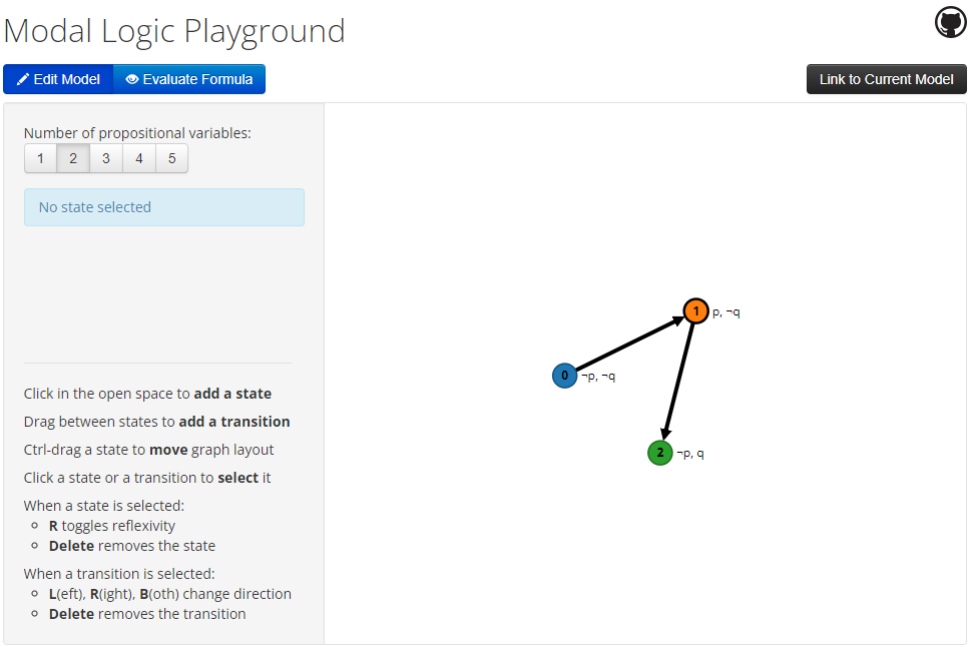

Figure 2.12: Modal Logic Playground

Dynamic Epistemic Logic Playground (DELP - https://vezwork.github.io/modallogic/) (*Evans*, 2022) is the second web tool, which is based on Modal Logic Playground. DELP has extended the functionality from MLP by adding an option for multiple relations (up to five), as well as modalities $\Box_i$ and $[\varphi!]\psi$ (from our language definition in section 2.2.2). Further, the user can "announce" a formula, which then transforms the model according to the modal change operation we describe in section 2.3.1.

When we started this project, we were only aware of Modal Logic Playground, and found out about Dynamic Epistemic Logic Playground in the later stages of development.

Chapter 3

Software

3.1 Language and libraries

3.1.1 JavaScript

For the purpose of dealing with Modal Logic and set theory, it might have been easier to write the project in Python programming language. Python has both built-in functions for working with set theory (*python.org*, 2023) and a library for creating, manipulating and making calculations on graphs (*Hagberg et al.*, 2008). However, we would like the application to be web based, as we believe this will make it far more accessible for anyone who would like to try it. Furthermore, a web application is much more practical to maintain and update than a desktop application the user needs to download and install locally.

For client-side programming, JavaScript is by far the most used language (*stackoverflow.com*, 2021), and is "the programming language of the web" (*w3schools.com*, 2023). For creating a visual web tool, JavaScript is practically the only option. It does also offer the same functionalities and data structures needed for this project, although with less built-in functions.

There are also a variety of JavaScript libraries suitable for our needs. Modern web development with JavaScript is often accompanied with an additional framework (e.g React, Angular and Vue). Although it is tempting to use an in-fashion framework, "vanilla" JavaScript, accompanied with libraries for parsing and visualization, seems to offer everything we need for the purpose of this project.

3.1.2 FormulaParser

FormulaParser (*Kirsling*, 2017) is a JavaScript parser for mathematical formulas (such as formulas of Propositional Logic), which takes a user input formula and transforms it to an Abstract Syntax Tree (AST), using a set of predefined rules where connectives and operators are defined, long with their respective precedence. The parser is implemented in both of the previously mentioned Modal Logic Playground and Dynamic Epistemic Logic Playground. We will describe this in more detail in section 3.3.

3.1.3 D3.js

D3.js is a JavaScript library for manipulating the Document Object Model (DOM)-tree based on mutable data (*d3js.org*). For visualization, the library uses Scalable Vector Graphics (SVG)-

elements attached to the DOM tree. Using some built-in methods of both JavaScript and D3.js, these elements can be added, removed or otherwise manipulated by the user. There are also mechanics that can be used to automatically adjust the position of each element according to the position of others (for example, we do not want nodes to be too close to or too far from each other). We use D3.js to draw nodes and edges (where the data is arrays of these nodes and edges), and to allow the user to modify and play with the models created. We will discuss D3.js further in section 3.8

3.2 Representation of models

We chose to represent models with two JavaScript Objects (Objects in JavaScript are similar to dictionaries, hashmaps or maps in other languages): one object for worlds and their valuations, and one separate object for the relations of the model.

Worlds

The domain and valuations are represented in a single, simple key:value object, where the worlds are represented by the keys, and the values contain a string representing true atomic variables for the corresponding world. The example below shows a set of four worlds where two propositional variables may or may not be true.

```
let worlds = {
           0:'pq',
           1:'p',
           2:'q',
           3:''
               }
```

The object structure makes accessing and evaluating truth values of each world both convenient and quite readable:

```
    console.log(worlds[0]) // propositions true in world 0
    >> 'pq'
    worlds[1].includes('p') // 'p' is true at world 1 / 'p' is in world 1
    >> true
    worlds[4] = 'r' // add a world where 'r' is true
    delete world[4] // remove world 4
    Object.keys(worlds).length // get number of worlds
    >> 4
```

Relations

The set of relations are represented by a nested object where each relation is an object with a key:value structure representing each world and its successors.

```
let relations = {
              'a': {
              0:[0,1,2],
              1:[0,1],
              2:[0,2,3],
```

```
3:[2,3]
},
'b': {
0:[0,2,3],
1:[1,2],
2:[0,1,2],
3:[0,3]
}
}
```

This lets us both access and manipulate successors for each world, evaluate formulas on the relations and perform set operations on the set of successors.

```
relations['a'][0] // get successors of world 0 for agent a
>> [0, 1, 2]

relations['a'][0].push(3) // add w3 to successors of w0
>> [0, 1, 2, 3]

relations['a'][0].filter(w => w !== 3) // remove w3 from successors of w0
>> [0, 1, 2]

// 'q' is true for least one of the successors from w0 for agent a
relations['a'][0].some(function(succState){return
    worlds[succState].includes('q')})
>> true

// 'p' is true for all successor from w1 for agent a
relations['a'][1].every(function(succState){return
    worlds[succState].includes('p')})
>> true

// intersection of w0 for agents a and b
[relations['a'][0], relations['b'][0]].reduce((a, b) => a.filter(c =>
    b.includes(c)))
>> [0, 2]

// union of w1 for agents a and b
[new Set([...relations['a'][1], ...relations['b'][1]])]
>> [0,1,2]
```

The attentive reader may have noticed that we use arrays to represent sets. The reason for this is that while JavaScript does have a set structure (as shown in the last example), the number of built in functions to perform on sets are limited (as opposed to arrays). While we could use the set structure and convert to arrays when we need to perform operations (and then convert back), this seemed to be an inefficient solution. Thus we decided that a more pragmatic approach was more convenient. The set structure does appear in the code, but this is just a nice way to remove duplicates (e.g. as shown above, to create unions).

3.3 The parser

The parser used is from a library called FormulaParser, "A parser class for simple formulae, like those of algebra and propositional logic." (*Kirsling*, 2017). The parser returns an abstract syntax tree, which we use to evaluate the formula. This allows us to define the unary and binary operators of our choice. Both unary and binary operators contain attributes for "symbol", "key" and "precedence", and the binaries also require an associativity ("left" or "right"). Our definitions look like this:

```
const unaries = [
    { symbol: '~', key: 'neg', precedence: 4 },
    { symbol: `K`, key: `nec`, precedence: 4 },
    { symbol: '<>', key: 'poss', precedence: 4 },
    { symbol: 'D', key: 'diff', precedence: 4 },
    { symbol: 'E', key: 'glob', precedence: 4 },
    { symbol: '[C!]', key: 'comm', precedence: 4 },
    { symbol: 'EK', key: 'ekno', precedence: 4 },
    { symbol: 'DK', key: 'dist', precedence: 4 },
    { symbol: 'CK', key: 'cokn', precedence: 4 },
    { symbol: '[|]', key: 'inac', precedence: 4 },
    { symbol: `[<-]`, key: `inve`, precedence: 4 }
    ];
const binaries = [
    { symbol: '&', key: 'conj', precedence: 3, associativity: 'right' },
    { symbol: '|', key: 'disj', precedence: 2, associativity: 'right' },
    { symbol: '->', key: 'impl', precedence: 1, associativity: 'right' },
    { symbol: '<->', key: 'equi', precedence: 0, associativity: 'right' },
    { symbol: '[!]', key: 'ann', precedence: 4, associativity: 'right' }
    ];
```

The "symbol" attribute is used to identify the operator in the formula string input by the user. The "key" attribute defines the name of the operator in the abstract syntax tree, and the precedence and associativity (for binaries) defines operator precedence and associativity.

Some of these (the first three unaries and all binaries except from "[!]") are originally defined in Kirsling's Modal Logic Playground (although we have renamed the box operator "[]" to "K"). The reason why we have defined [!] - public announcement - as a binary operator is a pragmatic one. This allows us to input any variable or formula on the left side of the operator, and the formula to be evaluated after the announcement on the right side (for example p[!]Kap - after an announcement of p, a knows P). Thus we to take the announced formula, perform the operation and return a new model on which the right side formula would be announced.

Dynamic Epistemic Logic Playground solves this by splitting the announcement operator into two operators - one unary "[" and one binary "]". This works similarly to our solution, but with the advantage that the user can put the formula inside the brackets. As we mentioned in section 2.5, we only became aware of DELP in the later stages of our project.

3.3.1 Abstract Syntax Tree

As mentioned above, the parser breaks a (well-formed) given formula down to an abstract syntax tree (AST) using the precedence and key attributes. The AST is a tree-structure ordered

by precedence rule, where the root node is the operator with highest precedence, followed by operators in descending order, and atomic propositional variables as the leaf nodes.

```
Input: ((p & q) | r) -> ~s
Result:
                    impl
                     /\
                    /  \
                  disj  neg
                   / \   |
                  /   \  s
                conj   r
                 /\
                /  \
               p    q
```

This structure allows for an efficient approach to evaluate formulas by providing a systematic way to evaluate sub-formulas and propositions, and apply operators in the correct order. As we will explore further a little bit later, the truth value is evaluated by recursively traversing the AST, evaluating each subformula.

3.4 Operators and modalities

Propositional connectives

The "standard" connectives of Propositional Logic (negation, disjunction, conjunction, implication and equivalence) are implemented by exploiting the built-in logic operators of the programming language. If the sub-formula to be evaluated is a negation, we check if !subFormula is true. If the connective to be evaluated is a binary connective, the parser returns an array containing the two sub-formulas, and we look at the relationship between these.

```
    subFormula[0] && subFormula[1] // conjunction
    subFormula[0] || subFormula[1] // disjunction
    !subFormula[0] || subFormula[1] // implication
    subFormula[0] === subFormula[1] // equivalence
```

Modal operators

For the existential and universal modal operators we use the built-in JavaScript functions `.some` and `.every` (as demonstrated briefly earlier). These are used to evaluate whether a sub-formula is true in respectively at least one and every world in the set of successor worlds.

```
// existential
    setOfSuccessors.some(function(successor) {
    return truth(successor, subFormula)
    })
// universal
    setOfSuccessors.every(function(successor) {
    return truth(successor, subFormula)
    })
```

The anonymous functions defines a condition that the built-in functions require to be true for at least one or every successor. In addition, the .every function is automatically true if the array is empty, while .some is not. This fulfills the semantic interpretation of the diamond and box, where the diamond is a conjunction of conditions (there is $u \in W$ such that Rwu and $(M,u) \Vdash \varphi$) and the box is an implication (every $u \in W$ is such that if Rwu then $(M,u) \Vdash \varphi$).

Further modal operators

Our developed tool allows the use of some further modal operators that can "see" beyond the standard modal operators, and can thus enhance and enrich the modal language. These do not follow the established relations, so we create new relations and define new modalities which follow similar patterns as other modalities, but on these new relations.

The global operator E can see every world in a model, and is defined as a diamond on a relation that is the Cartesian product of the domain. In other words, each world has every world in the model in its set of successors.

The difference operator D can see almost as much as E, except from the evaluation point itself. In other words, we need a Cartesian relation without the reflexive transitions (as the operator can not see the evaluation point).

We create a function that takes one argument, a Boolean that checks if the Cartesian relation should be complete or without reflexivity (allowing us to use the same function for both E and D). Then we add the worlds of the domain to each world (with or without the world itself).

```
function cartesian(reflexive) {
    let cartesianRelations = new Object();
    for (const world of Object.keys(worlds)) {
        cartesianRelations[world] = new Array();
    }
    if (reflexive === true) {
        for (const world of Object.keys(worlds)) {
            cartesianRelations[world] = Object.keys(worlds).map(w =>
                parseInt(w));
        }
    } else {
        for (const world of Object.keys(worlds)) {
            cartesianRelations[world] = Object.keys(worlds).filter(w => w !==
                world).map(w => parseInt(w));
        }
    }
    return cartesianRelations;
}
```

Model changing operators

We have defined two model changing operators: one that removes worlds from the domain (public announcement) and another that intersects relations (public communication).

Public announcement is an operation where we announce a formula to be true, and then remove every world (and their connected transitions) where the formula is not true. Then we evaluate the next sub-formula on the remainder of the model. To create the announced relation,

we add the worlds where the announced formula holds to a new set of worlds, and then add the relations not affected by announcement (formally the intersection of newWorlds $\times$ newWorlds and the original relations).

```
function publicAnnouncement(formula, worlds, relations) {
    let newRelation = new Object();
    let newWorlds = new Object();
    // Add worlds that "survive" the public announcement to the new domain
    for (const world of Object.keys(worlds)) {
        if (truth(world, worlds, relations, formula)) {
            newWorlds[world] = worlds[world];
        }
    }
    // fill the new relations with surviving worlds
    for (const agent of Object.keys(relations)) {
        newRelation[agent] = new Object();
        for (const world of Object.keys(relations[agent])) {
            if (Object.keys(newWorlds).includes(world)) {
                newRelation[agent][world] = new Array();

                for (const successor of relations[agent][world]) {
                    if (Object.keys(newWorlds).includes(String(successor))) {
                        newRelation[agent][world].push(successor);
                    }
                }
            };
        }
    }
    return [newWorlds, newRelation];
```

The communication operation only involves the relations, and does not affect the worlds. Specifically, it takes the intersected relation between each communicating agent and intersects it with the relations of each agent in the group. (If there is only one communicating agent, we skip the first intersection, and jump right to the final one.)

```
    function publicCommunication(agents, communicatingAgents, worlds,
        relations) {
    const notCommunicatingAgents = agents.filter(agent =>
        !communicatingAgents.includes(agent))
    let communicatingAgentsRelation = new Object();
    let newRelation = new Object();
    for (const agent of agents) {
        newRelation[agent] = new Object();
    }
    if (communicatingAgents.length > 1)
    // intersect relations of communication agents if more than one agent
    {
        for (const world of Object.keys(worlds)) {
            let toBeIntersected = new Array();
            for (const agent of communicatingAgents) {
                toBeIntersected.push([...relations[agent][world]]);
            }
```

```
            if (toBeIntersected.length > 0) {
                var intersectedWorld = new Array(toBeIntersected.reduce((a, b)
                    => a.filter(c => b.includes(c))));
            } else { var intersectedWorld = new Array() }
            communicatingAgentsRelation[world] = intersectedWorld;
        };
    } else { communicatingAgentsRelation = relations[communicatingAgents]; }
    // intersect remaining agent's relations with communicating agents
    for (const world of Object.keys(worlds)) {
        for (const agent of communicatingAgents) {
            newRelation[agent][world] = communicatingAgentsRelation[world];
        }
        for (const agent of notCommunicatingAgents) {
            newRelation[agent][world] = new Array(relations[agent][world],
                communicatingAgentsRelation[world]).reduce((a, b) =>
                a.filter(c => b.includes(c)))
        }
    }
    return newRelation;
}
```

Group operators

We have defined some group notions: Everybody Knows, Common Knowledge and Distributed Knowledge. These are standard modalities interpreted over relations built from a collection of relations, and they have an intuitively appealing epistemic interpretation. By using these as operators, we can explore what the current information state amongst a group of agent is; when combining them with "model changing" modalities, we can also explore how the information of groups is affected by different information dynamics.

Everybody Knows (EK) and Common Knowledge (CK) are quite similar. EK is a union of the relations within a group of agents, while CK is the transitive closure of EK.

```
function everybodyKnows(agents, worlds, relations) {
    let newRelation = new Object();
    for (const world of Object.keys(worlds)) {
        let toBeUnionized = new Array();
        for (const agent of agents) {
            toBeUnionized.push(...relations[agent][world]);
        }
        newRelation[world] = [...new Set(toBeUnionized)];
    }
    return newRelation;
}
function commonKnowledge(agents, worlds, relations) {
    let relation = everybodyKnows(agents, worlds, relations)
    let is_transitive = false;
    do {
        let changes = 0
        for (const world of Object.keys(relation)) {
            for (const successor of relation[world]) {
```

```
                const toBeAdded = new Array([...relation[successor]].filter(x
                    => !relation[world].includes(x)));
                if (toBeAdded.size > 0) {
                    relation[world].add(...toBeAdded);
                    changes++;
                }
            }
        }
        if (changes === 0) { is_transitive = true }
    }
    while (!is_transitive)
    return relation
}
```

Distributed Knowledge returns the intersected relation of a group of agents. For each world and each agent, we add the array of successors to an array, and then intersects the inner arrays to create the new set of successor for the world.

```
function distributedKnowledge(agents, worlds, relations) {
    let newRelation = new Object();
    for (const world of Object.keys(worlds)) {
        let toBeIntersected = new Array();
        for (const agent of agents) {
            toBeIntersected.push([...relations[agent][world]]);
        }
        if (toBeIntersected.length > 0) {
            var intersectedWorld = new Array(toBeIntersected.reduce((a, b) =>
                a.filter(c => b.includes(c))));
        } else { continue }
        newRelation[world] = intersectedWorld;
    }
    return newRelation;
}
```

Modal operators with agents

For multi-agent systems, we need agent-specific operators. To do this, we add a new operator for each agent to unaries. For example, for agent a we add an operator whose symbol is "Ka" and key is "neca". Then we use the last letter of the key to separate between agents while evaluating (which we will look deeper into later).

```
    for (const agent of agents) {
        unaries.push(
            { symbol: `K${agent}`, key: `nec${agent}`, precedence: 4 },
            { symbol: `<>${agent}`, key: `poss${agent}`, precedence: 4 }
            )}
```

Parser trouble

The above approach to modalities with agents works fine with single agent operators, but it has its drawbacks when it comes to operators that allows for multiple agents. For example, any number of agents can be included in the communication operator. Thus we need to add an operator for every possible combination of agents. If we have three agents we need seven operators, if we have four we need 15 (and so on). This is, to say the least, not ideal efficiency wise. We do however believe that in practice it will not be critical, as there is a limit to how many agents we use in these types of models.

3.5 Model-changing operations

We have also included functionality where the user can perform the dynamic operations (public announcement and communication) and see how the model changes. The functions used for this are the same as the ones we use for the modalities, but instead of evaluating a formula, we apply the changes to the visualisation of the model. For example, if the user wants to perform a communication action, we remove edges according to the definition of communication. This is in contrast to the corresponding *operators*, where we create a temporary model "behind the scenes" to evaluate formulas. With the dynamic *operations* we change the actual model (or, more precisely, we create a new model which replaces the original).

```
    for (const link of linksToBeRemoved) {
      svg.select(`#${link}`)
      .remove();
}
```

In the above code, we iterate over the array of links (edges) we want to remove, and then we use a D3.js method (`.select().remove()`) to remove said links. The links, which are elements in the DOM-tree, are identified with unique HTML id-properties. These properties are a concatenation of the relation label (agent name), source node and target node. A link for relation a from node 0 to node 1 will have `id = "a01"`. For the purpose of identifying links to be removed by a public announcement (which, remember, removes any edge connected to a world where the announced formula is not true), we also assign html classes to the links, one class for each node it is connected to. The same node as mentioned before will, in addition to its unique id, have `class = "w0 w1"` (HTML elements can have multiple classes assigned to them, separated by whitespace).

3.6 Forced properties

We want the user to be able to impose some properties on the models, because, as we discussed in 2.4, it makes sense within some sub-disciplines of Modal Logic to require that models should be e.g reflexive. The properties we have implemented are reflexivity, transitivity and symmetry.

For the reflexivity function, we simply check if each world is a successor of itself. If it is not, we add it. For symmetry, we iterate over every successor of each world for each agent, and add the opposite transition where it does not exist.

For transitivity, we add a new transition every time we encounter worlds that are connected by two steps and not by one, and we repeat this until no more transitions are added (as one

added transition may cause another one to be needed for the relation to be transitive).

We store any forced transitions, which the user may want to remove later, in an object. To remove a selected transition, we utilize a separate function that accesses this object.

3.7 Evaluating formulas

The function that does the actual evaluation of formulas is modelled after Kirsling's Modal Logic Playground, but with some modifications and quite a bit of extensions. Our function takes four arguments: worlds (the object that stores the domain and valuations), world (current world), relations (the relations object) and parsedFormula (the AST returned by the parser). The evaluator traverses through the AST recursively, starting at the root node and working itself down the tree.

The atomic proposition is evaluated simply by checking if the string of valuations for the evaluated world contains the atom we want to check.

```
function truth(world, worlds, relations, parsedFormula) {
    if (parsedFormula.prop) {
        return (worlds[world].includes(parsedFormula.prop))
    }
```

As mentioned above, for the standard propositional connectives we simply evaluate the truth value of each side of the connective, and return the "combined" truth value:

```
    else if (parsedFormula.conj) { // conjuction
        return (truth(world, worlds, relations, parsedFormula.conj[0]) &&
            truth(world, worlds, relations, parsedFormula.conj[1]))
    }
```

If the sub-formulas `parsedFormula.conj[0]` and [1] are atoms P and Q, the function evaluates $P \wedge Q$. If they are not, the functions continue to evaluate sub-formulas until they have reached the atom, then evaluate whether (in this case) both sub-formulas are true.

For the box (or K) and diamond operators with specified relations, we find the agent that we want to evaluate by looking at the last character of the operator key. Then, we can use the relations object to locate which world and successors we need to evaluate with the `.some` or `.every` function.

```
else if (Object.keys(parsedFormula)[0].slice(0, 3) === 'nec')
{
return
    relations[Object.keys(parsedFormula)[0].slice(3)][world].every(function
    (succState) { return truth(succState, worlds, relations,
    parsedFormula[Object.keys(parsedFormula)[0]]); })
        }
```

`Object.keys(parsedFormula)[0].slice(3)` provides the specific relation, which allows us to evaluate its belonging worlds. `.every` combined with the anonymous function and the recursive function call evaluate whether the next sub-formula holds with every successor (i.e. whether the box is true). Similarly, `.some` is used to evaluate whether the next sub-formula is true in at least one of the succeeding worlds (i.e. the diamond).

Except for the two dynamic operators, public announcement and communication (which

simply changes the model and evaluates whether the next sub-formula is true after model change), the rest of our modalities are variations of the box and the diamond. More precisely, they are boxes or diamonds along a new set of relations.

Take the global operator E, for example. As we have discussed earlier, this modalities can see every world regardless of transitions.

```
// Global modality
    else if (parsedFormula.glob) {
        const globalModel = cartesian(true);
        return (globalModel[world]).some(function (succState) { return
            truth(succState, worlds, relations, parsedFormula.glob) })
    }
```

To evaluate this modality, we create a new model with only one relation, which is the Cartesian product of the set of worlds. (The argument "true" indicates that we should include reflexive transitions, to separate the global modality from the difference modality). Further, we use the diamond on this new relation to see if the next sub-formula is true in any world. Similarly, for the inverse modality and the everybody knows modalities, we create new relations and evaluate formulas on these.

```
\\ Inverse modality
        else if (Object.keys(parsedFormula)[0].slice(0, 4) === 'inve') {
        const inverseModel = inverse(worlds, relations);
        return
            (inverseModel[Object.keys(parsedFormula)[0].slice(4)][world]).every(function
            (succState) { return truth(succState, worlds, relations,
            parsedFormula[Object.keys(parsedFormula)[0]]); })
    }
\\ Everybody knows modality
    else if (Object.keys(parsedFormula)[0].slice(0, 4) === 'ekno') {
        const unionizedModel =
            everybodyKnows(Object.keys(parsedFormula)[0].slice(4), worlds,
            relations)
        return (unionizedModel[world]).every(function (succState) { return
            truth(succState, worlds, relations,
            parsedFormula[Object.keys(parsedFormula)[0]]); })
    }
```

As mentioned before, the dynamic operators behave slightly differently. They perform changes on the model (remove world(s) and/or transitions) and return the manipulated model to be evaluated by the next sub-formula in the AST.

```
        else if (Object.keys(parsedFormula)[0].slice(0, 3) === 'nec') {
        return
            (relations[Object.keys(parsedFormula)[0].slice(3)][world].every(function
            (succState) { return truth(succState, worlds, relations,
            parsedFormula[Object.keys(parsedFormula)[0]]); }))
    }
    else if (Object.keys(parsedFormula)[0].slice(0, 4) === 'poss') {
        return
            (relations[Object.keys(parsedFormula)[0].slice(4)][world].some(function
            (succState) { return truth(succState, worlds, relations,
```

```
            parsedFormula[Object.keys(parsedFormula)[0]]); }))
    }
```

3.8 The visuals

As we barely touched in section 3.5, we use D3.js to add and manipulate SVG elements to the DOM tree. Our graph editor is based on code made by Ross Kirsling for the Modal Logic Playground (*Kirsling*, 2021b), but we have modified it to accept multiple relations, allow for assumed/forced properties, and removing nodes and/or edges as a result of dynamic operations.

The data structures used by D3.js to create nodes and edges are different from the ones we use to represent models. Both nodes and edges (or links) are defined as arrays of objects, where each object represent one node or link.

```
let nodes = [
  { id: 0, vals: 'p', reflexive: a },
  { id: 1, vals: 'q', reflexive: ab },
  { id: 2, vals: '', reflexive: b }
]
let links = [
  { source: nodes[0], target: nodes[1], left: false, right: true, id: "a-0-1"
     },
  { source: nodes[1], target: nodes[2], left: false, right: true, id: "b-12" }
];
```

Our modifications here are the "vals" property of the node objects, which is the valuation of each node, the "reflexive" property, (which originally was a Boolean to indicate whether there is a reflexive edge in the single-relation models of Modal Logic Playground) to a string that indicates which (if any) relations have a reflexive transition at that node, the "id" property of the link object, which is, as previously mentioned, a concatenation of agent, source and target.

The use of the valuation property should be self explanatory. The id-property has two uses. The first, which we mentioned in section 3.5, is to be able to identify which links to remove when performing the communication operation. The second, which is probably the most important, is to separate edges that are otherwise identical from each other by specifying the relation they belong to: "a01" belongs to relation a, while "b01" would belong to relation b.

```
    const g = circle.enter().append('svg:g'); // initialize a variable g that
       holds circle elements
    g.append('svg:text')  // add node ID to the circle elements
    .attr('x', 0)
    .attr('y', 4)
    .attr('class', 'id')
    .attr('id', (d) => `w${d.id}`)
    .text((d) => d.id); // set text to node id

    g.append('svg:text') // show node valuations
    .attr('x', 0)
    .attr('y', 20)
    .attr('class', 'id')
```

```
    .text((d) => d.vals) // set text to node vals

  for (const ID of linkIDs) { // Add relation label to link
    text.append("textPath")
      .attr("xlink:href", `#${ID}`) // find link by link id
      .style("text-anchor", "middle")
      .style("fill", "red")
      .attr("startOffset", "50%")
      .text(`${ID[0]}`);    // Add first character of link id as label
  }
```

In the code above, we show how we add node id, node valuations and relation label to our graph. Below, figure 3.1 show a simple graph with two relations and three worlds.

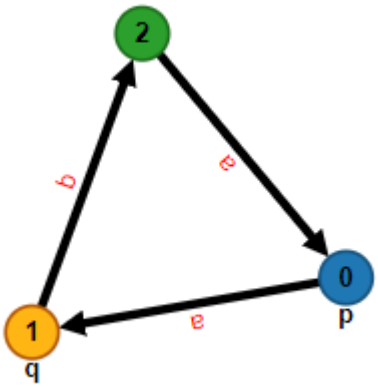

Figure 3.1: A simple graph created by our editor

3.9 Saving and uploading models

JSON

In order to upload and download models, we use the JSON (JavaScript Object Notation) file format, in which data is stored in object-like structures (as one may suspect from the name). We put the `links` and `nodes` arrays (as presented in section 3.8) in a single object, and create a JSON file using built-in JavaScript function `JSON.stringify`, and lets the user download this file

```
window.downloadJSON = async function (args) {
    const model = { 'links': links, 'nodes': nodes }
    let data, filename, link;
    let json = 'data:text/json;charset=utf-8,' + JSON.stringify(model);
    filename = args.filename || 'export.json';
    data = encodeURI(json);
    link = document.createElement('a');
    link.setAttribute('href', data);
    link.setAttribute('download', filename);
    link.click();
```

```
}
```

Chapter 4

Discussion

4.1 Results

Our main result is software[1] that can be used to represent models of Modal Logic, including multiple relations, where we have implemented both basic and more extended modal languages, as well as the ability to transform these model based on two model operation that have interesting (in particular, epistemic) interpretations. The software produced can be used to further implement modal operators or change operations, and it can be used as a sandbox for people who study the field.

Example: The Muddy Children Puzzle We quote *Baltag and Renne* (2016):

> Three children are playing in the mud. Father calls the children to the house, arranging them in a semicircle so that each child can clearly see every other child. At least one of you has mud on your forehead, says Father. The children look around, each examining every other childs forehead. Of course, no child can examine his or her own. Father continues, If you know whether your forehead is dirty, then step forward now. No child steps forward. Father repeats himself a second time, If you know whether your forehead is dirty, then step forward now. Some but not all of the children step forward. Father repeats himself a third time, If you know whether your forehead is dirty, then step forward now. All of the remaining children step forward. How many children have muddy foreheads?

In figures 4.1, 4.2, 4.3 and 4.4, we show how we solve the Muddy Children puzzle , which we also briefly mentioned in section 1.2 and illustrated by figure 1.1.

Here, agent a, b and c corresponds to atoms p - "child a is muddy", q - "child b is muddy", and r - "child c is muddy". (We avoid using relation labels as propositional variables, as they will be interpreted as relations rather than variables by the parser). Figure 4.1 models a situation where none of the children (a, b and c) know whether they are muddy. Figure 4.2 is the new model after the father announces that at least one of the children is muddy - $p \vee q \vee r$. [2] This eliminates the world where none of the children are muddy

[1] Link to the tool itself: https://bemastberg.github.io/multi-relation-modal-logic/
Link to Github repository: https://github.com/bemastberg/multi-relation-modal-logic

[2] In our input field, this is formulated as $p|q|r$ as logical OR $\vee$ is difficult to replicate by a keyboard - an alternative could be backslash and slash \/, but one character is faster to write than two. Similarly, we use & for logical AND $\wedge$.

After this first announcement, the father asks the children to step forward if they know whether they have mud on their foreheads. None of them answers, letting everybody know that no one knows whether they are muddy themselves. We formulate this as an announcement of $\neg K_a p \wedge \neg K_b q \wedge \neg K_c r$. After this announcement, every world where an agent knows their corresponding atom is removed, as shown in figure 4.3.

A second time, the father asks whether any of the children know if they have a muddy forehead. This time, some of the children step forward: at least one of the children know whether they are muddy. Announcing this as $K_a p \vee K_b q \vee K_c r$, world 3 is removed, as no children know their status here. Thus we get the final model, as shown in figure 4.4.

Execution time We have not performed any formal performance tests on the program, but we have run some simple time test on various formulas and models. Of course, the time the program takes to execute will vary from computer to computer (as it runs locally in the browser), but we feel it can give an impression on how the program will work for the end user. For reference, the tests have been run on a relatively modest laptop with 8GB RAM and a dual-core AMD Ryzen 3 3250u processor.

On the original Muddy Children model (figure 4.1), a formula with a conjuction of negations of three modalities ($\neg K_a p \wedge \neg K_b q \wedge \neg K_c r$) took on average (of ten executions) 27 milliseconds (or 0.027 seconds), while a disjunction of three atoms ($p \vee q \vee r$) took on average 16 milliseconds (or 0.016 seconds). On a less complex model with four worlds (figure 4.3), the average times for the same formulas were 7 and 11 milliseconds.

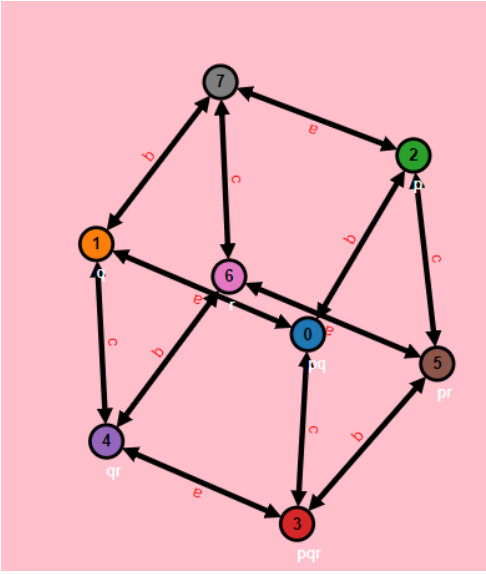

Figure 4.1: The muddy children puzzle

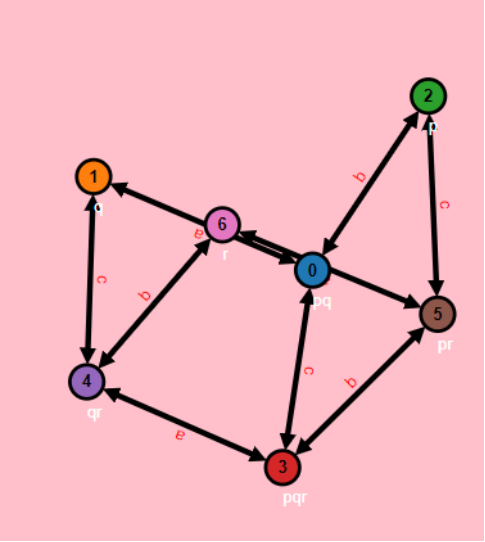

Figure 4.2: The puzzle after the first announcement ("at least one of you are muddy")

4.2 Discussion

4.2.1 Reflection over choices

The very first choice when it comes to technology was to use JavaScript. Given the fact that we wanted to make our project available as a web tool, this decision pretty much gave itself,

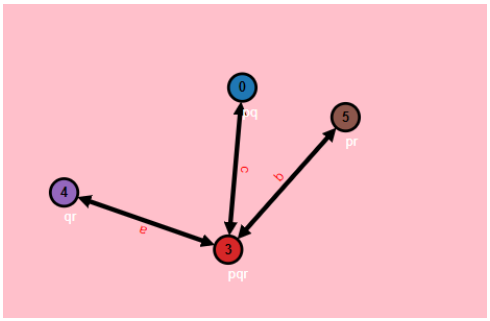

Figure 4.3: The muddy children puzzle after the second announcement ("no one knows whether they are muddy")

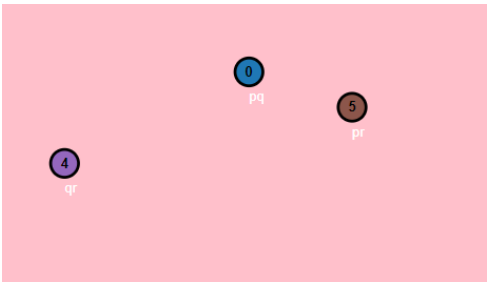

Figure 4.4: The puzzle after the third and final announcement ("somebody knows they are muddy"

as JavaScript is dominant when it comes to client-side programming. In addition, related work is written in the same language, and relevant libraries were available.

Further, for representing the relational structures of Modal Logic, we wanted a structure which was manageable to understand for anyone who may want to use the code for further projects, and we believed this could lower the threshold for those interested, so quite early we chose to represent the models (i.e. the worlds, relations and valuations) as JavaScript Objects. This was in part because this data structure is more human readable than, say, arrays of tuples:

```
relations = {
'a': {
'0': [0,1],
'1': [0,1,2],
'2': [0,2]
    },
'b': {
'0': [0],
'1': [0,1],
'2': [0,2]
    }
}
```

versus

```
    relations = [('a', 0, 0), ('a', 0, 1), ('a', 1, 0), ('a', 1, 1), ('a', 1,
       2), ('a', 2, 0), ('a', 2, 2), ('b', 0, 0), ('b', 1, 0), ('b', 1, 1),
       ('b', 2, 0) ('b', 2, 2)]
```

Another benefit of the Object structure is that it comes with the JavaScript Object Notation (JSON) file format, which is convenient when we want the user to be able to save their models locally for later usage.

We did, however, face some challenges with this chosen structure when the time came to build the visualization of the models. The library D3.js, which we use to show the graphs, is built upon a slightly different data structure where the sets of nodes and edges are represented as arrays of objects; one object for each node and edge:

```
let nodes = [
  { id: 0, vals: 'p', reflexive: false },
  { id: 1, vals: 'q', reflexive: true },
```

```
  { id: 2, vals: '', reflexive: false }
          ];
let links = [
  { source: nodes[0], target: nodes[1], left: false, right: true, id: "a01" },
  { source: nodes[1], target: nodes[2], left: false, right: true, id: "b12" }
          ];
```

When we discovered this (which we did a little late in the process), we had to choose between changing our original structures, or write code to transform back and forth between the two. As we would have to rewrite virtually everything we had so far if we were to choose the former, we went with the latter alternative. This is not ideal performance-wise, but the data structures will in practice never become large enough for it to become a problem for the user. As we explored at the end of section 4.1, even for relatively complex models (relative to the use this tool is meant for, that is) the execution time is negligible.

We use the library FormulaParser (*Kirsling*, 2017) for parsing and preparing formulas for evaluation. We chose this because it was already in use in similar projects, and seemed to be fit for the task. We did, however, encounter a problem when it came to parsing formulas that included arbitrary sets of relations, as discussed in 3.4. The solution, which became adding a new modality for every subset of the set of relations, is not optimal, but again, we deal with structures with sizes so (relatively) small that there in reality will be no effect on the user experience.

The graph editor itself is also a library we chose to implement, with some modifications to suit our needs. This decision was, again, made because the editor had been used for similar projects and it fit with our needs and our technology choices.

4.2.2 Potential applications

We believe the program can be used as a tool for both students and educators, as it offers an efficient and simplified way of building models and evaluating formulas of Modal Logic. We also hope that it can be an interesting project to continue to further develop with new concepts, or to be used as inspiration for similar projects.

4.3 Contributions

Our main contribution is the creation of a tool for Modal Logic which has a richer language and more functionality than existing projects. Using the projects Modal Logic Playground and Dynamic Epistemic Logic Playground as inspiration, we extended the languages used with some additional modalities, and we added another dynamic operator and action (compared to the latter). We also allow the user to require the models to have certain properties.

4.4 Conclusions and further work

Using some existing libraries and a mainstream programming language, we have successfully built a tool for building and reasoning about models of Modal Logic. We have implemented some features that were not included in similar projects, and built a project that can be further developed.

There are some things that we think could be done in a further development of this project. First and foremost, we would explore if it could be beneficial to rewrite parts of the code to fit the previously mentioned data structures fit for D3.js, as it would probably reduce the amount of code and possibly improve performance.

We would also consider implementing more properties for the user to require their model to adhere to.

We would also look into polishing the design and user experience. Right now, the program works fine and should be quite user friendly, but as we are neither front-end developers nor designers, there are almost certainly things that could be improved.

One big addition we can imagine, is for the user to define their own modalities and evaluate the models with those. This goes some way beyond our project, but it could make the program very valuable for especially researchers who experiment with modalities and study the implications they have in Modal Logic. For example, a more advanced version of the program could give the user general tools for defining additional relations (e.g. set difference) and to which one could associate some modality symbol. Then the user could use this just defined modality, semantically evaluated in a "standard" way (using existential or universal quantifier). Similarly, one could look for more general model operation, besides removing worlds of the domain (public announcement) and intersecting the relations (communication). If one allowed the user to define such operations and an associated modality, then the user could also explore the behaviour of different model-changing modalities.

For now, the only way anyone can do this is to write their own source code (which we would encourage people to do!), but this require both programming knowledge and time, and is probably a little too high a threshold for most.

Bibliography

Baltag, A., and B. Renne (2016), Dynamic Epistemic Logic, in *The Stanford Encyclopedia of Philosophy*, edited by E. N. Zalta, Winter 2016 ed., Metaphysics Research Lab, Stanford University. 1.2, 4.1

Blackburn, P., M. d. Rijke, and Y. Venema (2001), *Modal Logic*, Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, doi:10.1017/CBO9781107050884. 2.1, 2.1.1, 2.1.2

d3js.org (), D3.js. 3.1.3

Evans, E. (2022), Epistemic logic playground, https://github.com/vezwork/modallogic. 2.5

Goranko, V., and A. Rumberg (2022), Temporal Logic, in *The Stanford Encyclopedia of Philosophy*, edited by E. N. Zalta, Summer 2022 ed., Metaphysics Research Lab, Stanford University. 2.2.2

Hagberg, A. A., D. A. Schult, and P. J. Swart (2008), Exploring network structure, dynamics, and function using networkx, in *Proceedings of the 7th Python in Science Conference*, edited by G. Varoquaux, T. Vaught, and J. Millman, pp. 11 – 15, Pasadena, CA USA. 3.1.1

Humberstone, I. L. (1983), Inaccessible worlds., *Notre Dame Journal of Formal Logic*, *24*(3), 346 – 352, doi:10.1305/ndjfl/1093870378. 2.2.2

Kirsling, R. (2017), Formulaparser, https://github.com/rkirsling/formula-parser. 2.5, 3.1.2, 3.3, 4.2.1

Kirsling, R. (2021a), Modal logic playground, https://github.com/rkirsling/modallogic. 2.5

Kirsling, R. (2021b), Directed graph editor, https://gist.github.com/rkirsling/5001347. 3.8

python.org (2023), Built-in types, https://docs.python.org/3/library/stdtypes.html#set-types-set-frozenset. 3.1.1

Rendsvig, R., and J. Symons (2021), Epistemic Logic, in *The Stanford Encyclopedia of Philosophy*, edited by E. N. Zalta, Summer 2021 ed., Metaphysics Research Lab, Stanford University. 2.2.3

stackoverflow.com (2021), 2021 developer survey. 3.1.1

w3schools.com (2023), Javascript tutorial, https://www.w3schools.com/js/. 3.1.1