# Dynamic Tableaux *for* Dynamic Modal Logics

Thesis submitted in fulfillment of the requirements
for the degree of

Doctor of Philosophy

at

Vrije Universiteit Brussel

by

Jonas De Vuyst

Supervised by

Prof. Dr. Jean Paul Van Bendegem

& Dr. Patrick Allo

November 2013

(Minor Revision, January 2014)

# Acknowledgements

First and foremost I want to thank my supervisors Jean Paul Van Bendegem and Patrick Allo. This dissertation would not have been written without their guidance and advice. Their input was invaluable during the writing of a first and second thesis proposal, over the changes in research focus over the years, and to the creation of the concrete dissertation that you are now reading.

I want to thank Benedikt Löwe and Fenrong Liu for giving me the opportunity to spend a semester at the Institute for Logic, Language and Information in Amsterdam and at Tsinghua University in Beijing. I really learned a lot in Amsterdam, and in Beijing I made some of the fundamental breakthroughs that led to this dissertation.

I want to thank Lorenz Demey, Davide Grosse, Peter van Ormondt, Sonja Smets, Hans van Ditmarsch, Andreas Herzig, Giuseppe Primiero, Yanjing Wang, and Jens Ulrik Hansen for various discussions about logic that have benefitted me. This list is by no means exhaustive—I want to extend my gratitude to everyone else in the logic community that I have had the pleasure of meeting and to everyone at the Center for Logic and Philosophy of Science.

More generally, I want to thank my friends and family for their moral support over the years. I am especially indebted to my mother and my grandmother for having always encouraged and believed in my education.

Finally, I want to thank the Fund for Scientific Research – Flanders for funding my research the past four years and the Research Council at the Vrije Universiteit Brussel for funding my first year as a PhD student.

# Contents

1

# Introduction

In this PhD thesis we present new proof systems for several modal logics. We also present an implementation of one of these proof systems in the Clojure programming language.

The proof systems we present fall squarely in the category of tableau systems. Such systems have a long history. Many tableau systems have been developed for many different kinds of logic, including the logics found in this volume. Thus the contribution of our tableau systems is not to be found in their theoretical prowess. Rather, their intended benefits are conceptual simplicity, ease-of-use, modularity, and extensibility.

To understand how we believe we can deliver on these promises, let us take a step back and reflect on three core concepts of logic—syntax, semantics, and deduction.

## 1.1 Syntax, semantics, and deduction

Syntax can be understood as a delineation of the natural or formal language that we want to investigate. For example, we might be interested in sentences containing ‘atomic’ propositions, their negations, negations of these negations, and so on. Semantics refers to theories about the meaning of sentences and other terms of the language. Finally, deduction is a form of reasoning and refers to the deriviation of conclusions from premises. Both the premises and conclusions are here assumed to be expressible in the chosen language.

In formal logic the above three concepts are studied using tools from mathematics. We illustrate this using an example.

Consider the following syntax:

1. $p, q, r$ are atomic propositions.
2. Atomic propositions are well-formed formulas.
3. Recursively, if $\phi$ is a well-formed formula then so is not-$\phi$.
4. Nothing else is a well-formed formula.

Before we can give a semantics for this language, we need to define what constitutes a model. The job of the semantics is to tell us whether a given well-formed formula is true in a given model. Let $S$ be a model if and only if it is a set of atomic propositions. Also, let a proposition $p$ be true in a model $S$ if and only if $p \in S$; a negation not-$\phi$ is true in $S$ if and only if $\phi$ is not true in $S$. We now have a semantics and it unambiguously tells us that, for instance, not-not-not-$p$ is true in the model $\{q, r\}$.

In formal logic, the aspect of deduction is covered by proof systems. For instance, let's create a proof system that allows us to prove that $\psi$ (the conclusion) follows from $\phi$ (the premise). The system is as follows:

1. Start with the set $\{\phi, \text{not-}\psi\}$, called a tableau.
2. Recursively, if the tableau contains not-not-$\chi$ then add $\chi$ to the tableau.
3. If, for any atomic proposition $p$, the final tableau contains both $p$ and not-$p$ then $\psi$ follows from $\phi$; otherwise it does not follow.

The above proof system is called a tableau system. Using this system we can demonstrate that $p$ follows from or is 'entailed' by not-not-$p$. First of all, by step 1 we start with $\{\text{not-not-}p, \text{not-}p\}$. By step 2 this tableau is transformed into the tableau $\{\text{not-not-}p, \text{not-}p, p\}$. This tableau contains the atom $p$ and its negation not-$p$, which concludes our proof by step 3. Similarly, we can prove that not-not-$q$ follows from $q$ since step 1 gives us $\{q, \text{not-not-not-}q\}$ and step 2 leads to the tableau $\{q, \text{not-not-not-}q, \text{not-}q\}$.

**Figure 1.1.** In a sense, logic is a subfield of the study of languages. In formal logic, then, formal languages are studied. Once the language is circumscribed, a formal semantics and a proof system are sought out. Ideally, the proof system is sound and complete with respect to the semantics.

$$\begin{tikzcd}
 & \text{Syntax} \arrow[dl, "\textit{Meaning}"'] \arrow[dr, "\textit{Deduction}"] & \\
\text{Semantics} \arrow[rr, bend left, dashed, "\textit{Completeness}"] & & \text{Proof System} \arrow[ll, bend left, dashed, "\textit{Soundness}"]
\end{tikzcd}$$

The above proof system is sound. This means it allows us to prove that $\psi$ follows from $\phi$ only if in all models in which $\phi$ is true, $\psi$ is also true. It is also complete, meaning that if $\psi$ is true in all models in which $\phi$ is true then we can prove this entailment.

In languages that have an operator 'If … then …' it is sometimes possible to do away with premises. Instead of proving that $\psi$ follows from $\phi$ we prove that 'If $\phi$ then $\psi$' always hold. For languages in which the deduction theorem holds these statements are equivalent. In such languages we can prove that $\psi$ follows $\phi$ by starting with the tableau $\{$not- If $\phi$ then $\psi\}$. That is, we prove that 'If $\phi$ then $\psi$' without using premises. In other words, in many logics it suffices that a proof system is 'weakly complete', meaning that it can prove tautologies—well-formed formulas that are true in every model.

## 1.2 The semantics–proof system gap

In formal logic the usual aim is to discover proof systems that are sound and complete with respect to a semantics. However, both historically and in much of contemporary literature, proof systems and semantics are developed relatively independently. This is sometimes true in a temporal sense, such as when a proof system is devised long before a semantics is discovered (or vice versa). However, this gap also exists in a conceptual sense. Understanding many a proof system requires familiarity with a number of tautologies (technically, 'tautology schemas'), and it is often non-obvious why these tautologies hold simply by looking at the semantics.

We conjecture that the gap between semantics and proof systems is rooted in the long, rich, and partially independent histories that both these areas boast. We want to take a break from this tradition. In the proof systems in this dissertation we attempt to reflect the semantics of our languages as closely as possible. We aspire to the following benefits.

- Conceptual simplicity. Looking at the semantics, the proof system should be easy to understand.

- Ease of use. For simple formulas, humans should be able to create proofs and verify proofs easily. The proof system should also allow missteps to be unwound without requiring unrelated results to be erased.

- Modularity. There should be an injective mapping from the discrete elements of the semantics to the discrete elements of the proof system.

- Extensibility. Semantics tend to be easy to extend: After adding one operator to the language it typically suffices to add but one clause to the semantics. Proof systems should aspire to such extensibility also. Ideally, extending the proof system should not require revising previous elements of the proof system and it should not affect modularity.

## 1.3 What's in a proof?

There's one striking difference between tableau systems and most other formal proof systems that deserves a special mention. Direct proofs are the paradigm for most proofs systems. A proof might look as follows.

1. All people are mortal. *Premise.*

2. Philosophers are people. *Premise.*

3. Socrates is a philosopher. *Premise.*

4. Socrates is a person. *Inference from (2) and (3).*

5. Socrates is mortal. *Inference from (1) and (4).*

Here we only make use of logically correct inferences. Therefore every step in this proof is either (i) itself a premise or (ii) entailed by the premises. Thus the line (1) to (5) comprise a proof that from “All people are mortal”, “Philosophers are people”, and “Socrates is a philosopher” it follows that “Socrates is mortal”.

In contrast, proofs by contradiction are the paradigm for tableau systems. Here is one example of such a proof.

1. All people are mortal. *Premise.*

2. Philosophers are people. *Premise.*

3. Socrates is a philosopher. *Premise.*

4. Socrates is not mortal. *Negation of the intended conclusion.*

5. Socrates is not a person. *Inference from (1) and (4).*

6. Socrates is not a philosopher. *Inference from (2) and (5).*

This is a correct proof because lines (3) and (6) contradict each other. What happened is that we tried to construct a model in which the premises were true and the conclusion was false, but failed. Thus, constructing a proof in a tableau system corresponds to a trial-and-error method of learning. Opinions on this matter differ, but we personally feel trial-and-error methods are a more instructive paradigm of learning.

## 1.4 Modal Logics

Modal logics are logics for reasoning about possibility, knowledge, beliefs, preferences, time, and other modalities. Their semantics are almost always based on Saul Kripke's possible world semantics.

Kripke semantics provides an intuitive way to look at modalities. It is so intuitive that, from today's perspective, it is almost puzzling that for the first few decades after proof systems for modal logic were first invented, no one managed to develop semantics for them [21].

The intuitiveness of Kripke semantics suggests that tableaux for modal logic too should be easy to understand. This is indeed the case.

Dynamic modal logics are modal logics with dynamic operators for public announcements, belief revision, preference upgrades, and so on. The dynamic modal logics that interest us here are those that use Kripke semantics as a starting point and define their dynamic operators via mathematical operations on those semantics. Thus, for example, a belief revision operator in the syntax would correspond to a belief revision operation on models.

The 'dynamic' semantics of dynamic modal logics are a clever way of extending languages without compromising on intuitiveness. In this PhD thesis we present 'dynamic' tableau systems for these dynamic semantics, with the express aim to make them conceptually simple, easy to use, modular, and extensible.

# 2

# Notation and Structures

The research that this volume reports on is heavily grounded in basic set theory. As such, it should not come as a surprise that it contains a lot of mathematical notation. Mostly, that's a good thing!

It's a feature of mathematical notation that set-theoretic notions (among others) can be described in a very precise and concise way. It's no exaggeration to say that half a line of mathematical notation can often express ideas that would take several plain English sentences to explain. This adds up quickly when several such notions need to be related. Such verbosity is especially problematic when the idea that is being communicated is complex enough that it warrants repeated scanning. Additionally, mathematical notation is highly structured, which further facilitates parsing (given some practice).

Throughout this work it's assumed that the reader has a good grasp of basic set theory. This includes an understanding of functions and relations, graphs, and logical quantification.

In this chapter we discuss pivotal mathematical structures and introduce succinct notation for them. Admittedly, some of this notation is unconventional; however, it will be used often enough for the brevity that it affords to pay off.

## 2.1 Tuples and sets of tuples

The following general mathematical structures are used throughout this volume: sets, tuples, sets of tuples, and labeled graphs. We will not make any special assumptions with respect to sets. As such, let us go through some tuple-related notions straight away. (Notice that new names and notations are enclosed in angle quotes.)

**Definition 2.1.** A «tuple» $t = \langle e_0 \dots e_l \rangle$ is an ordered list of elements $e_0, \dots, e_l$. A tuple with $n$ elements is also called an «$n$-tuple». Moreover, a tuple with one element is called a «single», a 2-tuple is called a «pair», and a 3-tuple is called a «triple».

We will assume that the number of elements in a tuple is unambiguous. With $n \neq m$, no $n$-tuple is also an $m$-tuple.

Finally, the notation «$E_0 \times \dots \times E_l$» is used to denote the set of all $l$-tuples $\langle e_0 \dots e_l \rangle$ such that $e_0 \in E_0, \dots, e_l \in E_l$.

The above notation is somewhat unorthodox in that elements of tuples are not separated by commas. It is, however, in line with common notation for relationships—e.g. $Rabc$—and below we extend this convention even further.

Sets of tuples will prove to be a crucially important mathematical structure. Thus, we define extra notation that is optimized for our intended interactions with such sets.

**Definition 2.2.** With $S$ a set of tuples, we use «$Se_0 \dots e_l$» as a shorthand for '$\langle e_0 \dots e_l \rangle \in S$'.

The shorthand $Se_0 \dots e_l$ allows us to treat sets of tuples as if they were relations. This is sensible because relations and sets of tuples are virtually the same thing. The only difference is that relations have explicit domains.

**Proposition 2.1.** Let $T \subseteq D_0 \times \dots \times D_l$ be a set of tuples. $T$ corresponds to the relation $R$ between the elements of $D_0, \dots, D_l$ such that for all $e_0 \in D_0, \dots e_l \in D_l$ it is the case that $Te_0 \dots e_l \iff Re_0 \dots e_l$.

Given a sequence of elements $e_0, \dots, e_{i'}, \dots e_{l'}$ such that $e_{i'} \notin D_{i'}$ for at least one $i'$ (where $0 \leq i' \leq l$) it is the case that $Te_0 \dots e_{l'}$ is false whereas $Re_0 \dots e_{l'}$ is undefined.

One important use case for sets of tuples is filtering elements that match a certain pattern. The following notation makes this more convenient.

**Definition 2.3.** Let «$S[e_0 \ldots ? \ldots e_{l-1}]$» be the set $\{x \mid Se_0 \ldots x \ldots e_{l-1})\}$. Additionally, given a sequence $*_0, \ldots, *_{l-1}$ of $n$ objects and $m = l - n \geq 2$ metasyntactic variables ?, let «$S[*_0 \ldots *_{l-1}]$» be the set of $m$-tuples such that $\langle e_0 \ldots e_{m-1}\rangle \in S[*_0 \ldots *_{l-1}]$ if and only if $S$ contains the tuple that results from substituting the elements $e_0, \ldots, e_{m-1}$ for the ?-placeholders in $*_0, \ldots, *_{l-1}$, preserving their order.

When we know that the result of a query $S[*_0 \ldots *_{l-1}]$ is a singleton we sometimes write «$S(*_0 \ldots *_{l-1})$» to denote its unique element.

Take notice that when a query $S[*_0 \ldots *_{l-1}]$ contains exactly one ?, the result is a set of objects that are not necessarily tuples. In contrast, when such a query contains $m \geq 2$ variables then the result is a set of $m$-tuples.

**Example.** Given a set $S = \{\langle a\rangle, \langle abc\rangle, \langle abd\rangle\}$, it is the case that

- $S[ab?] = \{c, d\}$
- $S[a??] = \{\langle bc\rangle, \langle bd\rangle\}$
- $S[??c] = \{\langle ab\rangle\}$
- $S(??c) = \langle ab\rangle$

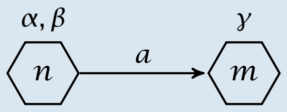
Given the correspondence of relations to sets of tuples we will sometimes also use this notation to query relations.

## 2.2 Labeled graphs

Many definitions of labeled graphs start by stating that a graph is a triple of the following three sets: a set of vertices, a set of vertex-label pairs, and a set of (labeled) edges. We opt for a slightly different notational approach and instead define graphs as sets that contain three kinds of tuples: singles that contain vertices, vertex-label pairs, and triples that represent the labeled edges.

**Definition 2.4.** $\mathcal{G} = \{\langle n\rangle \mid n \in N\} \cup L \cup E$ is an «lgraph» if and only if

**Figure 2.1.** Diagram of the lgraph $\{\langle n\rangle, \langle m\rangle, \langle anm\rangle, \langle n\alpha\rangle, \langle n\beta\rangle, \langle m\gamma\rangle\}$.



- $N$ is a set of objects. Every $n \in N$ is called a «node» or «vertex» (plural: vertices) of $\mathcal{G}$.

- $L$ is a set of pairs. We say that $l$ is a «label» (of $\mathcal{G}$) for $n$ if and only if $Lnl$. The «label-set» of a node $n$ (of $\mathcal{G}$) is the set $\mathcal{G}[n?]$. We also say that the node $n$ contains $l$ if and only if $l \in \mathcal{G}[n?]$.

  For every $\langle nl\rangle \in L$ it is the case that $n$ is a node of $\mathcal{G}$.

- $E$ is a set of triples. We say that $n'$ is «accessible» from $n$ over $a$ if and only if $Eann'$. Every $\langle ann'\rangle \in E$ is called a (labeled) «edge» or «link» (of $\mathcal{G}$) from $n$ to $n'$ indexed by $a$. Sometimes $\langle ann'\rangle$ is also described as the «$a$-edge» from $n$ to $n'$. Finally, $\langle ann'\rangle \in E$ is called an «outgoing» $a$-link of $n$ (in $\mathcal{G}$) and an «incoming» $a$-link of $n'$ (in $\mathcal{G}$).

  For every $\langle ann'\rangle \in E$ it is the case that $n$ and $n'$ are nodes of $\mathcal{G}$.

A «path» or «chain» in an lgraph $\mathcal{G}$ is a sequence $n_0, \dots, n_i, n_{i+1}, \dots, n_l$ such that for every natural number $i < l$ it is the case that $\mathcal{G}[?n_i n_{i+1}] \neq \emptyset$. If, moreover, $\mathcal{G}an_i n_{i+1}$ for all natural numbers $i < l$ then the sequence is called an «$a$-path» or «$a$-chain».

Given an lgraph $\mathcal{G}$, let «$\text{root}(\mathcal{G})$» be any $n \in \mathcal{G}[?]$ such that there is a path from $n$ to any other node of $\mathcal{G}$ (or undefined if no such node exists).

One convenient feature of our notation is that its nodes, labels, and edges can be accessed as elements of $\mathcal{G}$ in a fairly straightforward way.

**Example.** For any lgraph $\mathcal{G}$ it is the case that

- $\mathcal{G}n \iff n \in \mathcal{G}[?]$

- $\mathcal{G}nl \iff \langle nl\rangle \in \mathcal{G}[??]$

- $\mathcal{G}ann' \iff \langle ann' \rangle \in \mathcal{G}[???]$

Sometimes we will want to talk about graphs in combination with one specially designated node. Other times we will be interested in the vertices and edges of a graph, but not in its labels. Hence we introduce the following terminology.

**Definition 2.5.** With $\mathcal{G}$ an lgraph and $n$ one of its nodes, the pair $\langle \mathcal{G}n \rangle$ is called a «pointed graph». Here, $n$ is called the «current node» (of $\mathcal{G}$).

The «frame» of an lgraph $\mathcal{G}$ is the lgraph $\mathcal{F} \coloneqq \{\langle n \rangle \mid n \in \mathcal{G}[?]\} \cup \mathcal{G}[???]$.

We will investigate many different kinds of operations on graphs, many of which are monotone—that is, they only ever extend graphs. The following definition gives us a rough notion of what part of the graph was modified.

**Definition 2.6.** Given a monotone unary operation $f$ on graphs and a graph $\mathcal{G}$, we say that the nodes «affected» by applying $f$ to $\mathcal{G}$ are the elements of the following set:

$$\Delta[?] \cup \{n \mid \Delta[n?] \neq \emptyset, \Delta[?n?] \neq \emptyset, \text{ or } \Delta[??n] \neq \emptyset\},$$

where $\Delta = \mathcal{G}' - \mathcal{G}$.

That is to say, the nodes affected by $f$ are (i) the nodes that are added, (ii) the nodes to or from which edges are added, and (iii) the nodes to which new labels are added.

## 2.3 Graphs and relational structures

In the following chapters we will define Kripke models and action models as lgraphs. This is unusual. Normally these models are defined as relational structures. As mentioned above, however, sets of tuples and relations are practically the same thing. Therefore it is not surprising that lgraphs can easily be converted to relational structures (and vice versa).

**Definition 2.7.** A «relational Kripke structure» is a triple $\langle WRV \rangle$ such that $W$ is a set of nodes, $R$ is a function from indices to binary relations on $W$, and $V$ is a function from worlds to sets of labels.

**Proposition 2.2.** A relational Kripke structure $\langle WRV \rangle$ is equivalent to an lgraph $\mathcal{G}$ if and only if

- $W = \mathcal{G}[?]$.
- For all indices $a$ and all $\{w, v\} \subseteq W$, $R(a)wv \iff \mathcal{G}awv$.
- For all $w \in W$ it is the case that $V(w) = \mathcal{G}[w?]$.

The reason why we will define Kripke models as lgraphs rather than as relational structures is that throughout this thesis will often want to compare models to (other) lgraphs. Directly representing models as lgraphs makes this much more straightforward.

## 2.4 Embeddings and bisimulations

Embeddings and bisimulations are two general mechanisms for comparing lgraphs and other structures. Embeddings map structure and contents of one lgraph onto a second lgraph.

**Definition 2.8.** Given two lgraphs $\mathcal{G}$ and $\mathcal{G}'$, we say that the function $h : \mathcal{G}[?] \to \mathcal{G}'[?]$ is an «embedding» from $\mathcal{G}$ in $\mathcal{G}'$ if and only if

- $\mathcal{G}n \implies \mathcal{G}'h(n)$
- $\mathcal{G}nl \implies \mathcal{G}'h(n)l$
- $\mathcal{G}ann' \implies \mathcal{G}'ah(n)h(n')$

The above definition maps labels and indices in $\mathcal{G}$ onto themselves (in $\mathcal{G}'$). This leaves room for further generalization. The present definition is sufficiently abstract for our needs, however.

**Figure 2.2.** The lgraph on the left is embedded in the lgraph on the right (but not vice versa). The embedding is the function that maps $n$ and $n'$ onto $m$.

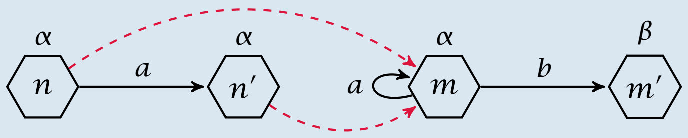
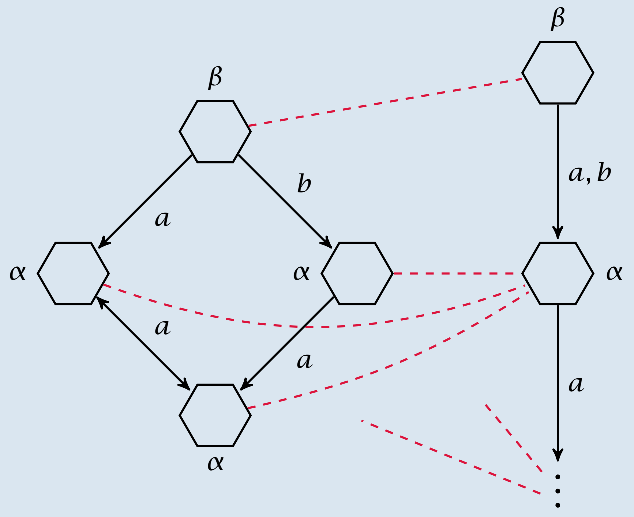


**Figure 2.3.** Two bisimilar lgraphs. One bisimulation is the relation that holds between the two $\beta$-nodes and between all $\alpha$-nodes. In fact, this particular bisimulation also happens to be the union of all bisimulations between these graphs.



It is instructive to reflect on the following question: When are two lgraphs or nodes ‘equivalent’? One answer would be to say that two lgraphs $\mathcal{G}$ and $\mathcal{G}'$ are equivalent if and only if there’s an embedding $h$ from $\mathcal{G}$ to $\mathcal{G}'$ such that its inverse function $h^{-1}$ is an embedding from $\mathcal{G}'$ to $\mathcal{G}$. This property is called isomorphism. Isomorphism is but one notion of equivalence, however.

Bisimulations represent a more flexible kind of equivalence of two nodes. The underlying idea is, metaphorically speaking, that we embark on two journeys starting at $n$ and $n'$ and are allowed only (i) to travel over the outgoing edges and (ii) to compare the labels of the current nodes. If after no such journey we can tell $n$ and $n'$ apart then both nodes are deemed bisimilar.

**Definition 2.9.** A «bisimulation» between (the nodes of) two lgraphs $\mathcal{G}$ and $\mathcal{G}'$ is a relation $Z$ between $\mathcal{G}[?]$ and $\mathcal{G}'[?]$ such that $Znn'$ if and only if the following conditions are met:

- $\mathcal{G}[n?] = \mathcal{G}'[n'?]$.
- For every $\langle am \rangle \in \mathcal{G}[?n?]$ there is an $m' \in \mathcal{G}'[an'?]$ such that $Zmm'$.
- For every $\langle am' \rangle \in \mathcal{G}'[?n'?]$ there is an $m \in \mathcal{G}[an?]$ such that $Zmm'$.

Two pointed lgraphs $\langle \mathcal{G}n \rangle$ and $\langle \mathcal{G}'n' \rangle$ are «bisimilar» if and only if there is a bisimulation $Z$ between them such that $Znn'$. We also write «$\langle \mathcal{G}n \rangle \underline{\leftrightarrow} \langle \mathcal{G}'n' \rangle$» if and only if $\langle \mathcal{G}n \rangle$ is bisimilar to $\langle \mathcal{G}'n' \rangle$.

## 2.5 Notation for formal syntax

In the introduction we explained that formal logic encompasses the study of formal languages. This means we need meta-syntax for describing said languages—specifically, we need conventions (i) for specifying their syntax or grammar and (ii) for denoting subsets of these languages that match certain patterns.

We specify the syntax of formal languages using a variant of Backus-Naur Form (BNF) that is commonplace in literature on formal logic. Let's start with an example.

**Example.** Let language $\mathcal{L}$ be the set of all formulas that are recursively defined as follows:

$$\phi ::= p \mid \neg\phi \mid (\phi \wedge \phi),$$

where $p \in \text{Prop}$.

This notation raises many questions. For instance, what kind of elements are in $\mathcal{L}$ and in Prop?

The elements of Prop are 'symbols' by virtue of their use in the definition of $\mathcal{L}$. It may help to think of Prop as consisting of letters of the Latin alphabet or as Unicode ‘codepoints’, although from a formal point of view they can be anything.

The elements of $\mathcal{L}$ are concatenations, sequences, or ‘strings’ of the elements of Prop and the symbols ‘¬’, ‘(’, ‘)’, and ‘∧’. Of course it is not the case that every string consisting of those symbols is an element of $\mathcal{L}$. Only elements that fit our recursive definition are.

It might help some readers if we recast our previous definition in a BNF commonly used in computer science. Suppose that Prop contains the symbols ‘p’, ‘q’, ‘r’, and nothing else. $\mathcal{L}$ then contains exactly those symbols that can be derived as ⟨expr⟩ using the following rules.

```
⟨atom⟩ ::= ‘p’ | ‘q’ | ‘r’
 ⟨expr⟩ ::= ⟨atom⟩ | ‘¬’ ⟨expr⟩ | ‘(’ ⟨expr⟩ ‘∧’ ⟨expr⟩ ‘)’.
```

That is to say, $\mathcal{L}$ is the smallest set that meets the following criteria:

- The elements of Prop are in $\mathcal{L}$.
- The concatenation of ‘¬’ and any element of $\mathcal{L}$ is an element of $\mathcal{L}$.
- The concatenation of ‘(’, any element of $\mathcal{L}$, ‘∧’, any element of $\mathcal{L}$, and ‘)’ is an element of $\mathcal{L}$.

We use the notation $|\phi|$ to denote the length of the string $\phi$.

Throughout this thesis we use $\phi, \psi, \chi,$ or $\xi$ to refer to strings of symbols. We use lowercase italic letters $p, q, r$ to refer to single symbols from the set Prop and $a, b, c$ to refer to symbols from the set Ind. Finally, we sometimes combine all of these, as well as the symbols (, ), [, ], ¬, ∧, !, and so on to refer to strings that match certain patterns. Thus, outside of syntax specifications, $(\phi \wedge \psi)$ refers to any string of the pattern ‘(’ ⟨string⟩ ‘∧’ ⟨string⟩ ‘)’. Moreover, if $(\phi \wedge \psi) = (\chi \wedge p)$ then $\phi = \chi$ and $\psi = p$. Therefore, outside the scope of BNF rules, $(\phi \wedge \phi)$ may refer to the string $(p \wedge p)$ but not to $(p \wedge q)$.

# 3 Modal Logic

Logic is often defined as the study of (correct) reasoning. In formal logic—also known as symbolic logic—this study is approached in a mathematical way. From here on, when we say ‘logic’ we mean ‘formal logic’.

Propositions are represented by sentences that are either true or false. For instance, the sentences “Snow is white” and “Cats meow and dogs bark” express propositions; contrariwise “Is it raining?” and “moon” are expressions that are neither true nor false. Propositions should not be confused with the sentences that represent them; rather, on one account propositions are the metaphysical entities which those sentences *refer* to [29]. Propositional logics are logics for reasoning about propositions.

In one popular way of doing propositional logic the following steps can be distinguished:

1. A language is defined in a formal way by inductively defining a «syntax» or «grammar». The language is the set of all sentences that adhere to this syntax.

   Propositional languages consist of atomic propositions $p, q, r, \dots$ that can be combined with a unary negation operator (‘not’, $\neg$) and binary operators for conjunction (‘and’, $\wedge$), disjunction (‘or’, $\vee$), and implication (‘if … then’, $\rightarrow$).

2. Each sentence of the formal language is given a meaning or «semantics» in set-theoretic terms. This makes it possible to unambiguously decide

for each sentence if it is true or false in a given «model» (or possibly at a certain 'point' in a model).

3. The semantics for the language may sometimes allow models to be constructed that, upon reflection, may not be germane to the language. For instance, suppose the semantics allow for a model in which a sentence $A$ is true. By appealing to intuition $A$ might in fact seem incoherent—meaning it should always be false. The logician's job is then to identify the set-theoretic property $P$ that singles out such subversive models. He or she then stipulates that $P$-models are not models for the logic that is being developed.

4. A proof system is devised. This involves specifying what a proof looks like and specifying rules for checking whether or not a proof is «correct». The outcome of a correct proof—after discharging all assumptions—is a sentence of the formal language that is 'logically true'. That is, a proof serves to demonstrate that a certain sentence is true in every model (excluding the subversive ones). This property is called «soundness». Ideally, all sentences that are true in every model can also be proved in the proof system. This is called «completeness».

   The outcome of a proof is a sentence—a purely syntactic object—and not a proposition. This means that when we prove that $(p \rightarrow p)$, we put no restrictions on what $p$ refers to. We would have proved that 'If Brussels is the capital of Belgium then Brussels is the capital of Belgium' and that 'If Prague is the capital of Columbia then Prague is the capital of Columbia'. In other words, logic concerns itself with the form of reasoning.

In this chapter we discuss an extension of (a classical interpretation of) propositional logic that is called modal logic. In subsequent chapters we extend our investigations to dynamic modal logics.

## 3.1 Syntax and semantics of $\mathcal{L}^{U\Box}$

Modal propositional logics extend (classical) propositional logic with modalities such as ‘necessarily’, ‘knows that’, ‘believes that’, or ‘it ought to be the case that’.

**Definition 3.1.** Let the language of multi-modal propositional logic $\mathcal{L}^{U\Box}$ consist of all «well-formed formulas» (wffs) $\phi$ composed as follows:

$$\phi \ ::= \ p \mid \neg\phi \mid (\phi \wedge \phi) \mid U\phi \mid \Box_a\phi,$$

with $p$ an element of a nonempty set of atomic propositions or atoms «Prop» and $a$ an element of a set of indices «Ind».

For convenience we also define the set of «literal propositions» or «literals»: Let « Lit » $\coloneqq \mathrm{Prop} \cup \{\neg p \mid p \in \mathrm{Prop}\}$.

The disjunction $(\phi \vee \psi)$ can be defined as $\neg(\neg\phi \wedge \neg\psi)$. The implication $(\phi \rightarrow \psi)$ is commonly regarded as equivalent to $(\neg\phi \vee \psi)$ and thus can also be defined in terms of $\neg$ and $\wedge$. There are also two logic symbols that do not take any arguments: $\top$ (‘top’), which means ‘logically true’, and its negation $\bot$ (‘bottom’, contradiction). These symbols can be reduced in terms of $\neg$, $\wedge$, and an arbitrary atomic proposition (on the assumption that Prop is nonempty). The symbol $\top$ will prove very useful in chapter 5. For the modal operators $U$ (‘universally’) and $\Box_a$ (‘box-$a$’) a ‘dual’ operator can be defined. Thus $\neg U \neg$ is often abbreviated as $E$ (‘somewhere’) and $\Diamond_a$ (‘diamond-$a$’) stands for $\neg\Diamond_a\neg$.

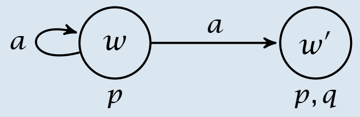
In contexts where it’s clear that Ind is a singleton, we sometimes omit the subscript from the box and diamond operators.

It is commonplace to interpret modal logics in terms of Kripke semantics—to the extent that the terms ‘modal logic’ and ‘Kripke semantics’ are often treated as if they were synonymous.

Where in non-modal propositional logic you might want to know if a wff holds in a model, in Kripke semantics you would want to know if a wff holds at a ‘world’ in a model.

**Definition 3.2.** A «Kripke model» $\mathcal{M}$ is an lgraph such that

**Figure 3.1.** The Kripke model $\{\langle w\rangle, \langle w'\rangle, \langle aww\rangle, \langle aww'\rangle, \langle wp\rangle, \langle w'p\rangle, \langle w'q\rangle\}$. Notice how $\Box_a p$ is true in $w$ because $p$ is a label for every world that is accessible from $w$ over $a$. The formula $\Box_a q$ is false in $w$. Finally, $\Diamond_a \top$ is true in $w$ and its negation $\Box_a \bot$ holds in $w'$.



- $\mathcal{M}[???] \subseteq \text{Ind} \times \mathcal{M}[?] \times \mathcal{M}[?]$
- $\mathcal{M}[??] \subseteq \mathcal{M}[?] \times \text{Prop}$

The nodes of a Kripke model are called «worlds», «states», or «points». The set of edges of a Kripke model is called its «accessibility relation». The «valuation» of an atomic proposition $p \in \text{Prop}$ in a world $w$ is ‘true’ if and only if $\mathcal{M}wp$; otherwise it is ‘false’.

Beware that our definition of Kripke models is slightly non-standard, as explained in section 2.3.

We can now have a look at the formal semantics for the formulas in $\mathcal{L}^{U\Box}$.

**Definition 3.3.** Let the «forcing relation» «$\Vdash$» be a binary relation such that for all $p \in \text{Prop}$, $a \in \text{Ind}$, and pointed Kripke models $\langle \mathcal{M}w\rangle$ it is the case that

$$\begin{aligned}
\langle \mathcal{M}w\rangle \Vdash p &\iff \mathcal{M}wp \\
\langle \mathcal{M}w\rangle \Vdash \neg\phi &\iff \text{not } \langle \mathcal{M}w\rangle \Vdash \phi \\
\langle \mathcal{M}w\rangle \Vdash (\phi \wedge \psi) &\iff \langle \mathcal{M}w\rangle \Vdash \phi \text{ and } \langle \mathcal{M}w\rangle \Vdash \psi \\
\langle \mathcal{M}w\rangle \Vdash U\phi &\iff \forall v \in \mathcal{M}[?] : \langle \mathcal{M}v\rangle \Vdash \phi \\
\langle \mathcal{M}w\rangle \Vdash \Box_a\phi &\iff \forall v \in \mathcal{M}[aw?] : \langle \mathcal{M}v\rangle \Vdash \phi
\end{aligned}$$

In a model $\mathcal{M}$, a formula $\phi$ is said to be true in, hold in, or satisfied by a world $w$ if and only if $\langle \mathcal{M}w\rangle \Vdash \phi$.

The above semantics is compositional. That is to say, its truth schemas reduce the question whether or not a formula $\phi$ (in a world $w$) is true to

**Table 3.1.** Some of the more popular (well-behaved) frame conditions. The right-most column contains the characteristic logical truth that results when the condition holds.

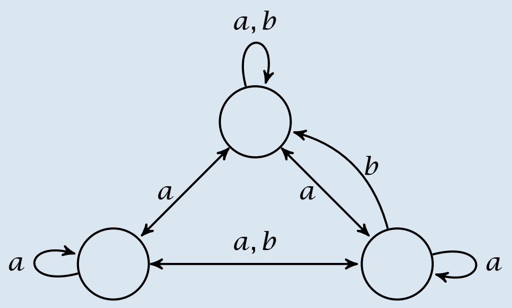
| Name | Predicate | Schema | Axiom |
|---|---|---|---|
| $K_a$ | | Always | $(\Box_a(\phi \to \psi) \to (\Box_a\phi \to \Box_a\psi))$ |
| $T_a$ | Reflexive | $\mathcal{G}ann$ | $(\Box_a\phi \to \phi)$ |
| $D_a$ | Serial | $\exists n' : \mathcal{G}ann'$ | $(\Box_a\phi \to \Diamond_a\phi)$ |
| $4_a$ | Transitive | $\mathcal{G}ann'$ and $\mathcal{G}an'n'' \implies \mathcal{G}ann''$ | $(\Box_a\phi \to \Box_a\Box_a\phi)$ |
| $B_a$ | Symmetric | $\mathcal{G}ann' \implies \mathcal{G}an'n$ | $(\phi \to \Box_a\Diamond_a\phi)$ |
| $5_a$ | Euclidean | $\mathcal{G}ann'$ and $\mathcal{G}ann'' \implies \mathcal{G}an'n''$ | $(\Diamond_a\phi \to \Box_a\Diamond_a\phi)$ |

the question of whether or not the 'subformulas' contained in $\phi$ are true (in worlds $w', w'', \ldots$). For instance, we evaluate the modal formula $\Box_a\phi$ by quantifying over the worlds that are accessible from the current world over $a$. Specifically, $\Box_a\phi$ is defined to be true if and only if $\phi$ evaluates to true in all the worlds accessible over $a$.

The intuitive understanding of the accessibility relation depends on the application. For instance, epistemic logics are modal logics where $\Box_a\phi$ is interpreted as 'agent $a$ knows that $\phi$'. It follows that $\Diamond_a\phi$ means 'for all $a$ knows, $\phi$ is the case'. Consequently, a link $\langle aww' \rangle$ could be thought of as representing agent $a$ as being unable to 'distinguish' $w'$ from $w$. In other words, as far as agent $a$ in $w$ can tell, she is situated in $w'$. We return to this issue in the next section.

Depending on the application the accessibility relations are restricted in various ways (table 3.1). For instance, in epistemic logic reflexivity is imposed, which has the effect of ensuring that if $\Box_a\phi$ is true in a world then so is $\phi$. This corresponds to the notion that it is impossible to know false propositions. Of course it is possible to *believe* false propositions and hence in doxastic logic, in which $\Box_a\phi$ is read as 'agent $a$ believes that $\phi$', the requirement that accessibility relations are reflexive is dropped. Instead seriality is substituted for reflexivity so that agents can believe false things but cannot believe con-

**Figure 3.2.** A frame $\mathcal{F}$ of which the set of $a$-edges constitutes an equivalence relation. Every frame that is reflexive and Euclidean, or serial, symmetric, and transitive is an equivalence frame. $\mathcal{F}$ is also serial with respect to $b$.



tradictions. Restrictions on the accessibility relations are known as frame conditions.

**Definition 3.4.** $\sigma$ is a (universally defined) «frame condition» if and only if it is a function from indices to predicates on lgraphs and, with $a$ any index, $\sigma(a)$ is invariant under isomorphisms, changes in labeling, and changes in edges that are not $a$-edges.

$\sigma$ is a «well-behaved» frame condition if and only if any lgraph can be extended to make it $\sigma(a)$-compliant. Specifically, for any given lgraph $\mathcal{G}$ and index $a$ there is an lgraph $\mathcal{G}' \supseteq \mathcal{G}$ for which $\sigma(a)$ holds.

Given two frame conditions $\sigma$ and $\sigma'$, let $\sigma \sqcup \sigma'$ be the frame condition such that for all indices $a$, $(\sigma \sqcup \sigma')(a)$ holds for an lgraph if and only if $\sigma(a)$ and $\sigma'(a)$ hold for it. Similarly, we write $\sigma \sqsubseteq \sigma'$ if and only if there is a frame condition $\sigma''$ such that $\sigma = \sigma' \sqcup \sigma''$.

We use the notation $\mathrm{K}_a$ to indicate the absence of constraints on index $a$. Thus $\mathrm{K}_a\mathcal{G}$ is true for every index $a$ and every lgraph $\mathcal{G}$. Despite the abuse of terminology, we say that the frame condition $\sigma$ contains no frame conditions if and only if $\sigma(a) = \mathrm{K}_a$ for every index $a$.

In addition to the predicates in table 3.1, we will also use the predicates 'equivalence' and 'preorder'. An accessibility relation with index $a$ is equivalent if and only if it meets $\mathrm{T}_a \sqcup 5_a$ or $\mathrm{T}_a \sqcup 4_a \sqcup \mathrm{B}_a$ (which amounts to the same thing). It is a preorder relation if and only if it equals $\mathrm{T}_a \sqcup 4_a$.

Finally, we say that a Kripke model $\mathcal{M}$ is a $\sigma$-model if and only if for all $a \in \text{Ind}$ it is the case that $\sigma(a)$ holds for $\mathcal{M}$.

## 3.2 Applications

Modal logics have many applications. The following overview is not exhaustive.

Alethic logics are logics for reasoning about logical or metaphysical possibility. Such logics have a single box operator. To put things differently, in alethic logics there is one index that represents possibility. What is true in all (accessible) possible worlds is considered necessary. Therefore $\Box\phi$ is read as 'It is necessarily so that $\phi$' and its dual $\Diamond\phi$ is read as 'It is possible that $\phi$'. Metaphysical possibility is sometimes taken to be an equivalence relation. This means it's assumed that possibility is a reflexive, transitive, and symmetric relation.

Epistemic logics allow us to reason about knowledge of one or more agents. Here, $\Box_a\phi$ is read as 'Agent $a$ knows that $\phi$. There is much disagreement about what frame conditions are appropriate for knowledge. Sometimes equivalence is assumed, but many logicians and philosophers would argue that this assumption is too strong. The symmetry axiom ($\phi \to \Box_a\neg\Box_a\neg\phi$) seems particularly untenable for it implies that if $\phi$ is true then you know that you don't know $\neg\phi$.

Reading the literature it would appear that many logicians think knowledge is at least reflexive and transitive. Philosophers, on the other hand, sometimes argue that knowledge is not transitive. They argue that it is not the case that if you know that $\phi$ then you know that you know that $\phi$, a principle that is also known as 'positive introspection'. A historically important defence of positive introspection can be found in [24] (chapter 5). See [39] for one powerful philosophical defense of knowledge as a modality that is not transitive. In conclusion, we want to point out that this issue is closely aligned with the internalism/externalism debate in epistemology.

Doxastic logics are logics of beliefs. The most notable difference between knowledge and belief is that knowledge is factual—if it is known that $\phi$ then

$\phi$ must in fact be true. Belief, on the other had, allows for mistakes. It is perfectly possible to believe something that turns out to be false.

Deontic logics deal with ethics, law, norms, and obligations. They encourage a reading of $\Box\phi$ as '$\phi$ ought to be the case', 'One is obliged to do $\phi$', '$\phi$ is required by law', and so on. In terms of Kripke semantics it can also be read as '$\phi$ is the case in all perfect worlds'.

Temporal logics are logics for reasoning about time. Commonly two accessibility relations are used to represent time. The first one is a transitive relation that orders instances in time from old to new. On this modality $\Box\phi$ is read as 'It will always be the case that $\phi$' and $\Diamond\phi$ is read as '$\phi$ is the case at some point in the future'. The second relation is the inverse of the first and orders instances in time from new to old. It affords statements about the past such as 'It was always the case that $\phi$' or '$\phi$ was true at some point'.

Temporal logics often also feature more advanced expressions, such as '$\phi$ holds until $\psi$ becomes true'. Because a plethora of incompatible views exist on the nature of time it should not come as a surprise that logicians disagree on how to model it. For instance, is time deterministic or is it open ended? Is time continuous or is it discrete?

Dynamic propositional logics (DPL) are logics for reasoning about state transitions. Thus they can be used for reasoning about computer programs, computer chips, and other state systems. In DPL $\Box_a\phi$ is written as $[a]\phi$ and the index $a$ represents a state transition. Additionally, DPL provides several operations for creating complex indices out of simple ones. Thus, for instance $[a \cup b]\phi$ could stand for '$\phi$ is the case after state transition $a$ or $b$'. There are usually also operators for expressing sequential operation, iteration, and testing.

Modal logic has also been suggested as a type system for programming languages. Type systems analyse programs for correctness and can prevent programs from running if they have type errors. Programming languages with type systems based on modal logic have been proposed. For instance, [11] presents a language for staged computing and [31] describes a programming language for safe computing with distributed resources.

## 3.3 A tableau proof system for $\mathcal{L}^{U\Box}$

On encountering a well-formed formula $\phi$, a formal logician will immediately want to know two things:

- Is $\phi$ valid?
- Is $\phi$ satisfiable?

**Definition 3.5.** A well-formed formula $\phi$ is «$\sigma$-valid» if and only if for every pointed $\sigma$-model $\langle \mathcal{M} w \rangle$ it is the case that $\langle \mathcal{M} w \rangle \Vdash \phi$. The notation $\vDash_\sigma \phi$ also indicates that $\phi$ is $\sigma$-valid. $\sigma$-valid formulas are also called «$\sigma$-theorems». A wff that is not $\sigma$-valid is called «$\sigma$-invalid».

Well-formed formulas that hold in at least one pointed $\sigma$-model are called «$\sigma$-satisfiable». Formulas that are not $\sigma$-satisfiable are «$\sigma$-unsatisfiable» and are also called «$\sigma$-contradictions».

A wff $\phi$ is «$\sigma$-contingent» if and only if it is true in at least one pointed $\sigma$-model and false in at least one pointed model.

We sometimes simply write that a formula $\phi$ is valid, invalid, satisfiable, unsatisfiable, or contingent when $\sigma$ contains no frame conditions.

The following proposition is the fundamental insight behind tableau proof systems.

**Proposition 3.1.** A well-formed formula $\phi$ is $\sigma$-valid if and only if $\neg\phi$ is not $\sigma$-satisfiable.

This gives rise to the following general strategy: Systematically try to construct a (pointed) model for a formula $\phi$. If successful and if the model-constructing system used is sound then this results in a pointed model for $\phi$. If unsuccessful and if the model-constructing system is complete then this means that its negation is valid.

Tableau systems are one way to implement this general strategy. The tableau systems we will use are based on graph-rewriting.

**Definition 3.6.** An «$\mathcal{L}$-tableau» $\mathcal{T}$ is an lgraph with edges that are indexed by elements of Ind and with well-formed formulas of the language $\mathcal{L}$ for labels.

When it is unambiguous what $\mathcal{L}$ is, we drop it as a prefix.

Tableau systems are based on rules which look for patterns in a tableau and, based on the patterns they encounter, add new nodes, edges, or labels to it. We also allow rules to transform tableaux into tableaux that embed them.

We can now formally define what a tableau rule is.

**Definition 3.7.** A «tableau rule» $R$ is a «non-destructive» relation between tableaux, meaning that with $\mathcal{T}$ and $\mathcal{T}'$ tableaux, if $R\mathcal{T}\mathcal{T}'$ then $\mathcal{T}$ is embedded in $\mathcal{T}'$.

Let Rules be the collection of tableau rules defined in table 3.2 and let Rules$_\sigma$ be the largest subcollection of Rules that contains a frame condition rule $\mathbf{R}^{\mathbf{a}}_{\mathrm{X}}$ only if $\mathrm{X}_a \sqsubseteq \sigma$ (where $\mathrm{X}_a$ is a frame condition as defined in table 3.1).

In later chapters we will add further constraints to Rules. However, those rules will concern formulas outside $\mathcal{L}^{U\Box}$, and as such will not affect the results in this chapter.

As for $\mathbf{R}_\star$, it is important to understand that it can be used to merge nodes, add nodes and edges, add literals to nodes, and rename nodes. One purpose of $\mathbf{R}_\star$ is to help us turn those situations where the tableau rules conspire to create acyclic never-ending paths around by transforming these paths into loops. We will find additional uses for $\mathbf{R}_\star$ in later chapters.

One interesting property of the mandatory rules of Rules—where $\mathbf{R}_\star$ is the only optional or non-mandatory rule—is that they are non-destructive in a strong sense.

**Proposition 3.2.** If $R$ is a mandatory rule of Rules then, with $\mathcal{T}$ and $\mathcal{T}'$ tableaux, if $R\mathcal{T}\mathcal{T}'$ then $\mathcal{T} \subseteq \mathcal{T}'$.

The mandatory rules also have the property that they add at most three elements to the tableau.

**Proposition 3.3.** If $R$ is a mandatory rule of Rules then, with $\mathcal{T}$ and $\mathcal{T}'$ tableaux, if $R\mathcal{T}\mathcal{T}'$ then the cardinality of $\mathcal{T}' - \mathcal{T}$ is at most 3.

**Table 3.2.** The tableau rules in schematic form. Formulas $\mathbf{R}_\neg$ to $\mathbf{R}_\Diamond$ act on the presence of formulas in the tableau. The rules $\mathbf{R}^{\mathbf{a}}_{\mathbf{T}}$ to $\mathbf{R}^{\mathbf{a}}_{\mathbf{5}}$ deal with frame conditions. These schemas should be interpreted as follows: With $\mathcal{T}$ and $\mathcal{T}'$ tableaux and $R$ one of the rules below, $R\mathcal{T}\mathcal{T}'$ if and only if $\mathcal{T}$ meets the precondition and $\mathcal{T}'$ is a minimal extension of $\mathcal{T}$ that meets the postcondition.

| Name | Precondition | Postcondition |
|---|---|---|
| $\mathbf{R}_\neg$ | $\mathcal{T}n\neg\neg\phi$ | $\mathcal{T}'n\phi$ |
| $\mathbf{R}_\wedge$ | $\mathcal{T}n(\phi \wedge \psi)$ | $\mathcal{T}'n\phi$ and $\mathcal{T}'n\psi$ |
| $\mathbf{R}_\vee$ | $\mathcal{T}n\neg(\phi \wedge \psi)$ | $\mathcal{T}'n\neg\phi$ or $\mathcal{T}'n\neg\psi$ |
| $\mathbf{R}_{\mathbf{U}}$ | $\mathcal{T}nU\phi$ and $\mathcal{T}n'$ | $\mathcal{T}'n'\phi$ |
| $\mathbf{R}_{\mathbf{E}}$ | $\mathcal{T}n\neg U\phi$ and $\mathcal{T}[?\neg\phi] = \varnothing$ | $\mathcal{T}'n'$ and $\mathcal{T}'n'\neg\phi$ for some new $n' \in \mathbb{N}$ |
| $\mathbf{R}_\Box$ | $\mathcal{T}ann'$ and $\mathcal{T}n\Box_a\phi$ | $\mathcal{T}'n'\phi$ |
| $\mathbf{R}_\Diamond$ | $\mathcal{T}n\neg\Box_a\phi$ and $\mathcal{T}[an?] \cap \mathcal{T}[?\neg\phi] = \varnothing$ | $\mathcal{T}n', \mathcal{T}'ann'$, and $\mathcal{T}'n'\neg\phi$ for some new $n' \in \mathbb{N}$ |
| $\mathbf{R}^{\mathbf{a}}_{\mathbf{T}}$ | $\mathcal{T}n$ | $\mathcal{T}'ann$ |
| $\mathbf{R}^{\mathbf{a}}_{\mathbf{D}}$ | $\mathcal{T}n$ and $\mathcal{T}[an?] = \varnothing$ | $\mathcal{T}'n'$ and $\mathcal{T}'ann'$ for some new $n' \in \mathbb{N}$ |
| $\mathbf{R}^{\mathbf{a}}_{\mathbf{4}}$ | $\mathcal{T}ann'$ and $\mathcal{T}an'n''$ | $\mathcal{T}'ann''$ |
| $\mathbf{R}^{\mathbf{a}}_{\mathbf{B}}$ | $\mathcal{T}ann'$ | $\mathcal{T}'an'n$ |
| $\mathbf{R}^{\mathbf{a}}_{\mathbf{5}}$ | $\mathcal{T}ann'$ and $\mathcal{T}ann''$ | $\mathcal{T}'an'n''$ |
| $\mathbf{R}_\star$ | Anytime / Optional | Any non-destructive transformation |

Moreover, whenever a mandatory rule of Rules is triggered to add a formula $\phi$ to the tableau, $\phi$ is shorter than the formula that triggered the rule.

**Definition 3.8.** A tableau rule $R$ is «reductive» if and only if, given any two tableaux $\mathcal{T}$ and $\mathcal{T}'$, if $R\mathcal{T}\mathcal{T}'$ then for all $\langle n\phi\rangle \in \mathcal{T}' - \mathcal{T}$ it is the case that there is a node-label pair $\langle n'\psi\rangle \in \mathcal{T}$ such that $|\phi| \leq |\psi|$.

**Proposition 3.4.** All mandatory rules of Rules are reductive.

To further aid our understanding of the tableau rules it may be instructive to reflect on the following situations.

**Example.** Given a tableau $\mathcal{T}$ and a tableau rule $R$, we make a distinction between the following cases:

- $R[\mathcal{T}?] = \emptyset$. For the rules in table 3.2 this occurs when the precondition does not match a pattern in $\mathcal{T}$.
- $R\mathcal{T}\mathcal{T}$. If $R$ is a rule of table 3.2 then this means $R$ has already been applied to $\mathcal{T}$.
- $R[\mathcal{T}?] - \{\mathcal{T}\} \neq \emptyset$. This indicates that $R$ can be applied to $\mathcal{T}$ to produce a new tableau.
- $\{\mathcal{T}', \mathcal{T}''\} \subseteq R[\mathcal{T}?] - \{\mathcal{T}\}$ and $\mathcal{T}' \neq \mathcal{T}''$. Here $\mathcal{T}'$ and $\mathcal{T}''$ are two branches. In other words, they represent two possible outcomes of applying $R$ to $\mathcal{T}$.

It is important to realize that the cases $R[\mathcal{T}?] = \emptyset$ and $R[\mathcal{T}?] = \{\mathcal{T}\}$ have the same effect: $R$ cannot be used to transform $\mathcal{T}$ into a new tableau.

Tableaux represent (partial) attempts at constructing a model that satisfies a certain formula. We now define terminology for describing this process.

**Definition 3.9.** A tableau $\mathcal{T}'$ is a «$\sigma$-development» of a tableau $\mathcal{T}$ if and only if it is either an element of or the limit of a sequence $\mathcal{T}, \ldots, \mathcal{T}_i, \mathcal{T}_{i+1}, \ldots$ such that for every $\mathcal{T}_{i+1}$ in the sequence there is an $R \in \text{Rules}_\sigma$ such that $R\mathcal{T}_i\mathcal{T}_{i+1}$ and $\mathcal{T}_i \neq \mathcal{T}_{i+1}$.

A tableau $\mathcal{T}$ is a $\sigma$-tableau «for» $\langle n\phi\rangle$ if and only if (i) $\mathcal{T} n\phi$ and (ii) $\mathcal{T}$ is a $\sigma$-development of a tableau $\mathcal{T}_0 \coloneqq \{\langle n'\rangle, \langle n'\phi\rangle\}$ (for some $n'$).

We sometimes also say that a tableau $\mathcal{T}$ is a $\sigma$-tableau for $\phi$ if and only if there is an $n$ such that $\mathcal{T}$ is a $\sigma$-tableau for $\langle n\phi\rangle$.

**Proposition 3.5.** If $\mathcal{T}$ is a $\sigma$-tableau for $\phi$ then any $\sigma$-development of $\mathcal{T}$ is a $\sigma$-tableau for $\phi$.

We now introduce the notion of saturation to indicate that all mandatory rules have been applied to a tableau. In other words, saturated tableaux are tableaux that cannot be changed further by mandatory rules.

**Definition 3.10.** A tableau $\mathcal{T}$ is said to be «$R$-saturated» if and only if $R[\mathcal{T}?] - \{\mathcal{T}\} = \emptyset$. With $\sigma$ a frame condition, a tableau is said to be «$\sigma$-saturated» if and only if it is $R$-saturated for all mandatory rules $R \in \text{Rules}_\sigma$. Finally, a tableau is said to be «saturated» if and only if it is $R$-saturated for all mandatory rules $R \in \text{Rules}$ that act on the presence of formulas.

To be clear, saying that a tableau is saturated is synonymous with saying it is $\text{K}_a$-saturated.

For every tableau there is a $\sigma$-saturated extension.

**Proposition 3.6.** If $S = \mathcal{T}_0, \dots, \mathcal{T}_i, \mathcal{T}_{i+1}, \dots$ is a sequence of tableaux such that for every $\mathcal{T}_{i+1}$ in the sequence there is a mandatory rule $R \in \text{Rules}_\sigma$ such that $R\mathcal{T}_i\mathcal{T}_{i+1}$ then there is a $\sigma$-saturated tableau $\mathcal{T}$ that $S$ converges to (in the limit).

**Proof.** By propositions 3.2 and 3.3, $S$ is a countable sequence of sets ordered by the subset relation. Therefore the limit of $S$ equals $\bigcup S$. This proves that $\mathcal{T}$ exists. By construction of $S$ it also follows that $\mathcal{T}$ is $\sigma$-saturated.

When we want to (try to) construct a $\sigma$-model for a formula $\phi$, we will first construct a tableau $\{\langle n\rangle, \langle n\phi\rangle\}$ (for some $n$). Next, we apply the mandatory rules from $\text{Rules}_\sigma$ until we arrive at a $\sigma$-saturated tableau $\mathcal{T}'$. Finally we check if the tableau we found can be transformed into a model.

**Definition 3.11.** A tableau $\mathcal{T}$ contains a «literal contradiction» if and only if there is a node $n$ and an an atom $p \in \text{Prop}$ such that $\mathcal{T}np$ and $\mathcal{T}n\neg p$. Otherwise $\mathcal{T}$ is free of literal contradictions.

Rather than speak of tableaux free of literal contradictions, it is customary to instead say that we are looking for 'open' saturated tableaux.

**Definition 3.12.** An $\mathcal{L}^{U\Box}$-tableau is «open» if and only if it doesn't contain literal contradictions. If a tableau is not open then it is «closed».

For now no distinction is made between tableaux free of literal contradictions and open tableaux. Nevertheless, such a distinction exists for the tableaux for dynamic modal logics that we will discuss in subsequent chapters.

We now turn to proving soundness. We want it to be the case that if all $\sigma$-saturated tableaux for $\phi$ are closed then no $\sigma$-model exists in which $\phi$ is true (at some world). We will prove the contrapositive—if there's a $\sigma$-model for $\phi$ then there's an open $\sigma$-saturated tableau for $\phi$.

First, we define what models a tableau can be said to represent.

**Definition 3.13.** A model $\mathcal{M}$ «satisfies» a tableau $\mathcal{T}$ up to $\phi$ via a function $f : \mathcal{T}[?] \to \mathcal{M}[?]$ if and only if

1. $\mathcal{T}n\psi \implies \langle \mathcal{M}f(n)\rangle \Vdash \psi$
2. $\mathcal{T}ann' \implies \mathcal{M}af(n)f(n')$

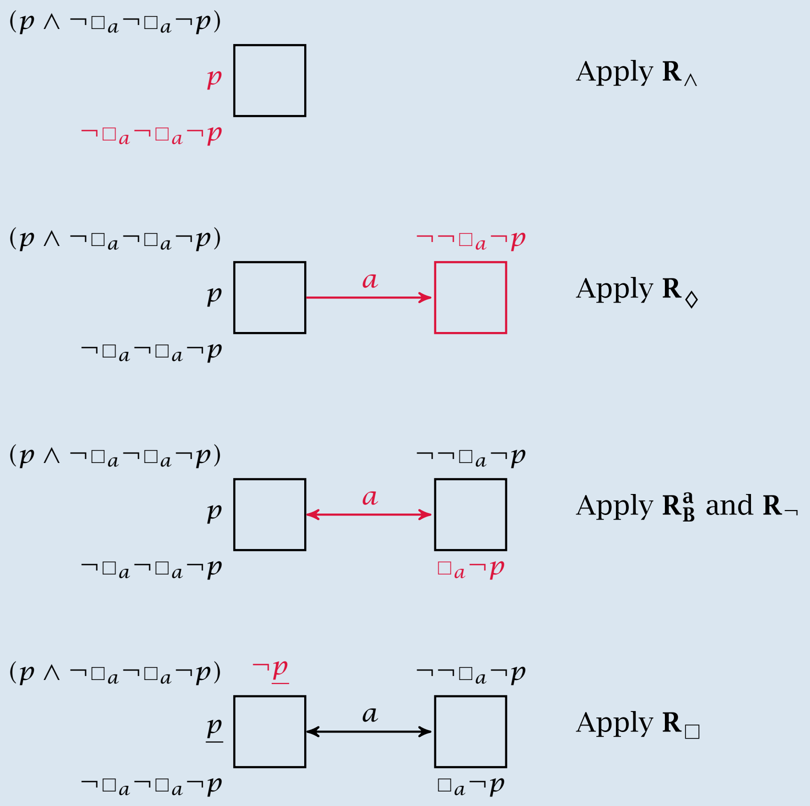
for all $\psi$ such that $|\psi| \leq \phi$.

We simply say that $\mathcal{M}$ satisfies $\mathcal{T}$ (via $f$) if and only if $\mathcal{M}$ satisfies $\mathcal{T}$ (via $f$) up to arbitrary formulas.

Finally, we say that $\mathcal{M}$ «natively satisfies» $\mathcal{T}$ if and only if $\mathcal{M}$ satisfies $\mathcal{T}$ via the identity function.

Suppose there is a $\sigma$-model $\mathcal{M}$ such that $\langle \mathcal{M}w\rangle \Vdash \phi$. It is easy to see that the tableau $\{\langle n\rangle, \langle n\phi\rangle\}$ is satisfied by $\mathcal{M}$ via the function that maps $n$ onto $w$. Let this be the base step. Next, let $\mathcal{T}$ be *any* tableau that is satisfied by

**Figure 3.3.** Proof that $(p \to \Box_a \Diamond_a p)$ is $\mathrm{B}_a$-valid. As a preliminary, we eliminate the operators $\to$ and $\Diamond_a$ since they are not primitives of $\mathcal{L}^{U\Box}$. This yields the formula $\neg(p \wedge \neg\Box_a\neg\Box_a\neg p)$. Next, we attempt to construct an open $\mathrm{B}_a$-saturated tableau for its negation (for ease of presentation we remove the sequence $\neg\neg$). However, we discover that we inevitably run into a literal contradiction. This proves that $(p \to \Box_a\Diamond_a p)$ is $\mathrm{B}_a$-valid.


$(p \wedge \neg\Box_a\neg\Box_a\neg p)$
$p$
$\neg\Box_a\neg\Box_a\neg p$
Apply $\mathbf{R}_\wedge$
$\neg\neg\Box_a\neg p$
$a$
Apply $\mathbf{R}_\Diamond$
$\Box_a\neg p$
Apply $\mathbf{R}^{\mathrm{a}}_{\mathrm{B}}$ and $\mathbf{R}_\neg$
$\neg p$
Apply $\mathbf{R}_\Box$

$\mathcal{M}$. We want to show that, for any mandatory rule $R \in \mathrm{Rules}_\sigma$, if there is an $R$-development of $\mathcal{T}$ then there is an $R$-development of $\mathcal{T}$ that is satisfied by $\mathcal{M}$.

**Lemma 3.7 (Basic Lemma to Soundness).** Let $\mathcal{T}$ be a tableau that is satisfied by a $\sigma$-model $\mathcal{M}$ via a function $f$. For every $R \in \mathrm{Rules}_\sigma$ such that $\mathcal{T}$ is not $R$-saturated it is then the case that there is a tableau $\mathcal{T}' \in R[\mathcal{T}] - \{\mathcal{T}\}$ and a function $f'$ such that $\mathcal{M}$ satisfies $\mathcal{T}'$ via $f'$.

**Proof.** Suppose a rule is triggered by $\mathcal{T} n \phi$.

- If $\phi$ is $\neg\neg\psi$ or $(\psi \wedge \chi)$ then $\mathcal{T}'$ is uniquely determined and $\mathcal{T}'$ has the same frame as $\mathcal{T}$. Let $f' \coloneqq f$.

- Suppose $\phi$ is $\neg(\psi \wedge \chi)$. By definition of $\wedge$ and $\neg$ either $\langle \mathcal{M} f(n)\rangle \Vdash \neg\psi$ or $\langle \mathcal{M} f(n)\rangle \Vdash \neg\chi$. Let $\mathcal{T}' = \mathcal{T} \cup \{\langle n \neg\psi\rangle\}$ or $\mathcal{T}' = \mathcal{T} \cup \{\langle n \neg\chi\rangle\}$ accordingly and let $f' \coloneqq f$.

- If $\phi$ equals $U\psi$ then $\mathcal{T}' = \mathcal{T} \cup \{\langle n' \psi\rangle\}$ for some $n' \in \mathcal{T}[?]$. By definition of $U$, however, $\langle \mathcal{M} f(n')\rangle \Vdash \psi$. Thus, let $f' \coloneqq f$.

- If $\phi$ equals $\neg U\psi$ then it is the case that $\mathcal{T}' = \mathcal{T} \cup \{\langle n'\rangle, \langle n' \neg\psi\rangle\}$ for some $n' \notin \mathcal{T}[?]$. Since $\langle \mathcal{M} f(n)\rangle \Vdash \neg U\psi$ it follows that there is a $w$ such that $\mathcal{M}w$ and $\langle \mathcal{M}w\rangle \Vdash \neg\psi$. Hence, let $f'$ be the smallest extension of $f$ such that $f'(n') = w$.

- Suppose $\phi$ is $\Box_a\psi$. This could yield a tableau $\mathcal{T}' \coloneqq \mathcal{T} \cup \{\langle n' \psi\rangle\}$, for some $n' \in \mathcal{T}[an?]$. By definition of $\Box_a$ it is so that $\langle \mathcal{M} n''\rangle \Vdash \phi$ for every $n'' \in \mathcal{M}[af(n)?]$. Since for every $n''' \in \mathcal{T}[an?]$ it is the case that $\mathcal{M} a f(n) f(n''')$ it follows that $\mathcal{M}$ satisfies $\mathcal{T}$ via $f$; thus let $f' \coloneqq f$.

- Suppose $\phi$ is $\neg\Box_a\psi$. This means $\mathcal{T}' = \mathcal{T} \cup \{\langle n'\rangle, \langle ann'\rangle, \langle n' \neg\psi\rangle\}$ for some $n' \notin \mathcal{T}[?]$. Since $\langle \mathcal{M} f(n)\rangle \Vdash \neg\Box_a\psi$ there is a $w$ such that $\mathcal{M} a f(n) w$ and $\langle \mathcal{M}w\rangle \Vdash \neg\psi$. Thus let $f'$ be the smallest extension of $f$ such that $f'(n') = w$.

Suppose a tableau $\mathcal{T}'$ is the result of applying a rule $R$ for enforcing frame conditions.

- If $R$ is an instance of $\mathbf{R^a_T}$, $\mathbf{R^a_4}$, $\mathbf{R^a_B}$, or $\mathbf{R^a_5}$ then let $f' \coloneqq f$.

- If $R$ is an instance of $\mathbf{R^a_D}$ then it is assumed that the model $\mathcal{M}$ is serial. This implies that there is a a $w$ such that $\mathcal{M} a f(n) w$. Thus, let $\mathcal{T}' \coloneqq \mathcal{T} \cup \langle ann'\rangle$ (with $n' \notin \mathcal{T}[?]$) and let $f'$ be the smallest extension of $f$ such that $f'(n) = w$.

**Theorem 3.8 (Soundness).** Given some $\phi \in \mathcal{L}^{U\Box}$, if all $\sigma$-saturated tableaux for $\phi$ are closed then $\vDash_\sigma \neg\phi$.

**Proof.** We prove the contrapositive—viz. that if there is a pointed $\sigma$-model $\langle \mathcal{M} w \rangle$ such that $\langle \mathcal{M} w \rangle \Vdash \phi$ then there is an open saturated $\sigma$-tableau $\mathcal{T}$ for $\langle w \phi \rangle$.

Evidently, the pointed model $\langle \mathcal{M} w \rangle$ natively satisfies the tableau $\{\langle w \rangle, \langle w \phi \rangle\}$. By repeatedly applying lemma 3.7 it follows, by proposition 3.6, that there is a saturated open $\sigma$-tableau for $\langle w \phi \rangle$ that is satisfied by $\mathcal{M}$ via some function. Because it is satisfied by a model it also follows that the tableau is free of literal contradictions and that it is open since an atom cannot be both true and false in a single point in a model.

Our next task is to prove that our tableau method is complete. We would like it to be the case that if a tableau $\mathcal{T}$ for $\phi$ is $\sigma$-saturated and open then we can transform $\mathcal{T}$ into a $\sigma$-model for $\phi$.

Here's what we will treat as the orthodox transformation of a tableau into a model.

**Definition 3.14.** A model $\mathcal{M}$ is the «stock model» of a tableau $\mathcal{T}$ if and only if

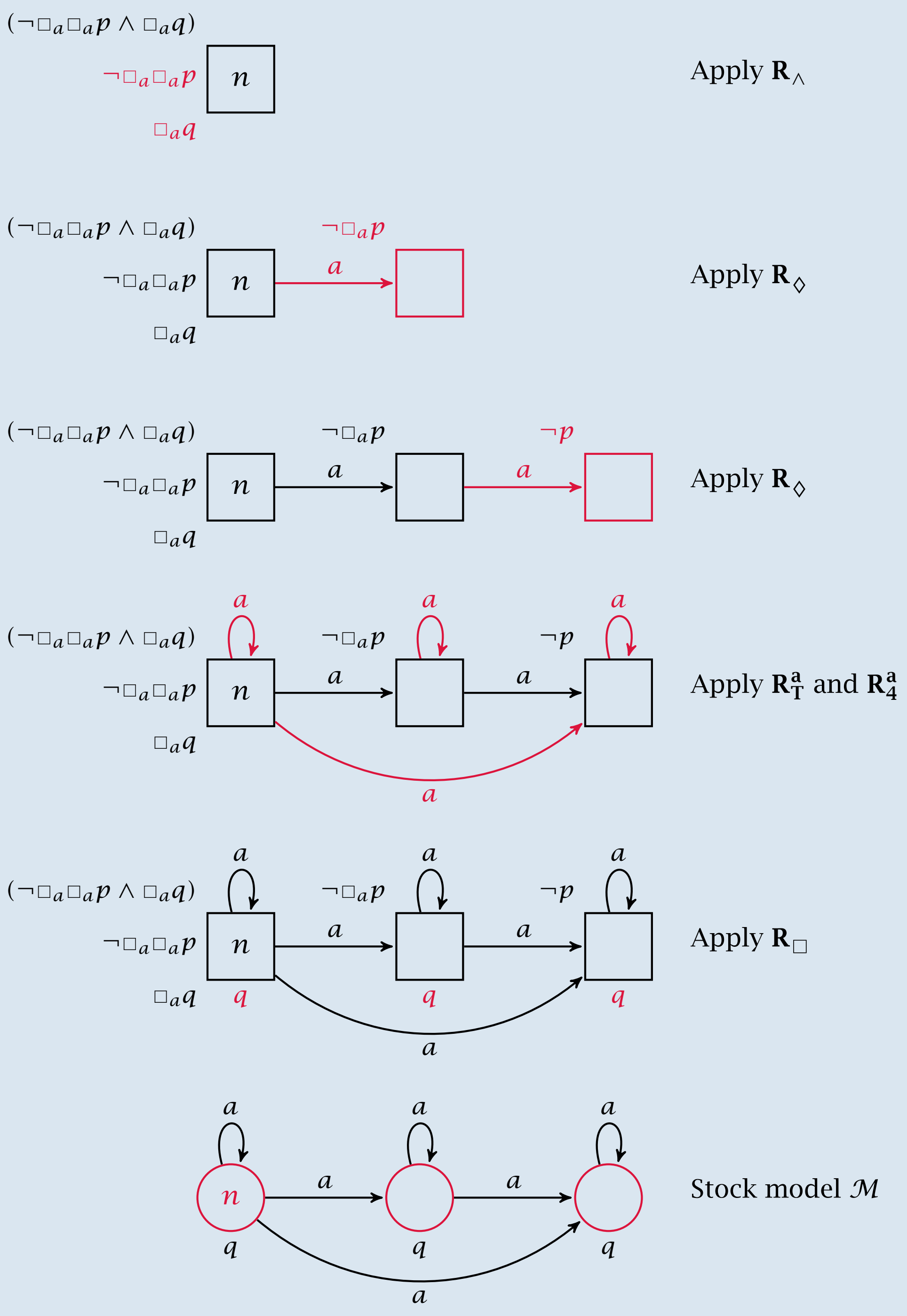
- $\mathcal{M}[?] = \mathcal{T}[?]$
- $\mathcal{M}[???] = \mathcal{T}[???]$
- $\mathcal{M}[??] = \mathcal{T}[??] \cap (\mathcal{T}[?] \times \mathrm{Prop})$

We now prove that if $\mathcal{M}$ is the stock model of an open saturated tableau $\mathcal{T}$ then $\mathcal{M}$ satisfies $\mathcal{T}$.

**Lemma 3.9 (Basic Lemma to Completeness).** Let $\mathcal{T}$ be any open saturated tableau. If $\mathcal{T} n \phi$ then, with $\mathcal{M}$ the stock model of $\mathcal{T}$, $\langle \mathcal{M} n \rangle \Vdash \phi$.

**Proof.** Proof by induction. By induction hypothesis (IH) it is assumed that the lemma holds for all $\psi$ shorter than $\phi$.

- Suppose $\phi$ is an atom. It is then the case that $\mathcal{T} n \phi$. Because $\mathcal{M}$ is the stock model of $\mathcal{T}$ it follows that $\langle \mathcal{M} n \rangle \Vdash \phi$.



$(\neg\Box_a\Box_a p \wedge \Box_a q)$
$\neg\Box_a\Box_a p$
$\Box_a q$
$n$
Apply $\mathbf{R}_\wedge$
$\neg\Box_a p$
$a$
Apply $\mathbf{R}_\Diamond$
$\neg p$
Apply $\mathbf{R}_\Diamond$
Apply $\mathbf{R}^{\mathbf{a}}_{\mathbf{T}}$ and $\mathbf{R}^{\mathbf{a}}_{\mathbf{4}}$
$q$
Apply $\mathbf{R}_\Box$
Stock model $\mathcal{M}$

**Figure 3.4.** We use the tableau system to contruct a pointed $(\mathrm{T}_a \sqcup 4_a)$-model for $(\neg\Box_a\Box_a p \wedge \Box_a q)$. We start with a tableau that consists of a single node and a single label. Next, we repeatedly apply the mandatory tableau rules. Because the final, $(\mathrm{T}_a \sqcup 4_a)$-saturated, tableau is open we can convert it into a $(\mathrm{T}_a \sqcup 4_a)$-model $\mathcal{M}$ by deleting all labels that are not atoms. It can be verified that $\langle \mathcal{M} n\rangle \Vdash (\neg\Box_a\Box_a p \wedge \Box_a q)$.

- Suppose $\phi$ is $\neg p$ (with $p \in \text{Prop}$). Since $\mathcal{T}$ is an open tableau it is then not the case that $\mathcal{T}np$. Since $\mathcal{M}$ is the stock model of $\mathcal{T}$ it follows that not $\langle \mathcal{M}n \rangle \Vdash \phi$.

- Suppose $\phi$ is $\neg\neg\psi$. By $\mathbf{R}_{\neg}$, since $\mathcal{T}$ is saturated, it is the case that $\mathcal{T}n\psi$. By IH it follows that $\langle \mathcal{M}n \rangle \Vdash \psi$. By definition of $\neg$ it now also follows that $\langle \mathcal{M}n \rangle \Vdash \neg\neg\psi$.

- Suppose $\phi$ is $(\psi \wedge \chi)$. By $\mathbf{R}_{\wedge}$ it is the case that $\mathcal{T}n\psi$ and $\mathcal{T}n\chi$. By induction hypothesis it follows that $\langle \mathcal{M}n \rangle \Vdash \psi$ and $\langle \mathcal{M}n \rangle \Vdash \chi$. By definition of $\wedge$ this entails that $\langle \mathcal{M}n \rangle \Vdash (\psi \wedge \chi)$.

- Suppose $\phi$ is $\neg(\psi \wedge \chi)$. By $\mathbf{R}_{\vee}$ either $\mathcal{T}n\neg\psi$ or $\mathcal{T}n\neg\chi$. In the first case, by IH, $\langle \mathcal{M}n \rangle \Vdash \neg\psi$. In the second case $\langle \mathcal{M}n \rangle \Vdash \neg\chi$ for the same reason. By definition of $\wedge$ and $\neg$ it follows that $\langle \mathcal{M}n \rangle \Vdash \neg(\psi \wedge \chi)$.

- Suppose $\phi$ is $U\psi$. It is then the case that $\mathcal{T}n'\psi$ for all $n' \in \mathcal{T}[?]$. By IH it follows that $\langle \mathcal{M}n' \rangle \Vdash \psi$ for all $n' \in \mathcal{T}[?]$. By definition of $U$ it now follows that $\langle \mathcal{M}n \rangle \Vdash U\psi$.

- Suppose $\phi$ is $\neg U\psi$. There is then an $n' \in \mathcal{T}[?]$ such that $\mathcal{T}n'\neg\psi$. By IH it is also the case that $\langle \mathcal{M}n' \rangle \Vdash \neg\psi$. Finally, by definition of $\neg$ and $U$ it follows that $\langle \mathcal{M}n' \rangle \Vdash \neg\psi$.

- Suppose $\phi$ is $\Box_a\psi$. For all $n'$ such that $\mathcal{T}ann'$ it is the case that (i) $\mathcal{T}n'\psi$ (by $\mathbf{R}_{\Box}$), (ii) $\langle \mathcal{M}n' \rangle \Vdash \psi$ (by IH), and (iii) $\mathcal{M}ann'$ (since $\mathcal{M}$ is the stock model of $\mathcal{T}$). By definition of $\Box_a$ it follows that $\langle \mathcal{M}n \rangle \Vdash \Box_a\psi$.

- Suppose $\phi$ is $\neg\Box_a\psi$. By $\mathbf{R}_{\Diamond}$ it is the case that $\mathcal{T}n'\neg\psi$ and $\mathcal{T}ann'$ for some $n'$. By IH it follows that $\langle \mathcal{M}n' \rangle \Vdash \neg\psi$. Finally, it is a consequence of the definition of $\neg$ and $\Box_a$ that $\langle \mathcal{M}n \rangle \Vdash \neg\Box_a\psi$.

**Theorem 3.10 (Completeness).** If $\vDash_\sigma \neg\phi$ (with $\phi \in \mathcal{L}^{U\Box}$) then all saturated $\sigma$-tableaux $\mathcal{T}$ for $\phi$ are closed.

**Proof.** Again we prove the contrapositive: If there is an open $\sigma$-saturated tableau $\mathcal{T}$ for $\langle n\phi \rangle$ then there's a $\sigma$-model $\mathcal{M}$ such that $\langle \mathcal{M}n \rangle \Vdash \phi$.

Let $\mathcal{M}$ be the stock model of $\mathcal{T}$. By lemma 3.9 it immediately follows that $\langle \mathcal{M} n \rangle \Vdash \phi$. That $\mathcal{M}$ is a $\sigma$-model follows from the definitions of the frame condition rules.

## 3.4 Decidability of $\mathcal{L}^{U\Box}$-tableaux

The above soundness and completeness results entail that if we possessed infinite computing power, our tableau system could tell us if a formula was valid, unsatisfiable, or contingent.

We now build on these results and show that even with finite computing time we can decide whether or not a formula is satisfiable. This is all we need because if a formula is unsatisfiable then by extension its negation is valid. Additionally, if a formula and its negation are satisfiable then it is contingent.

To make the search for a saturated open tableau terminate after a finite number of steps we need to keep an eye out for redundant copies of nodes and replace them by loops.

First, we devise a formal method for summing up the constraints that are imposed on a tableau node $n$ by $\Box_a$-formulas.

**Definition 3.15.** Given a tableau $\mathcal{T}$ and a node $n \in \mathcal{T}[?]$, define

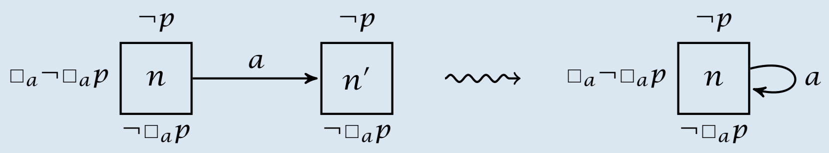
$$«\Box_a^{\mathcal{T}}(n)» \coloneqq \{\phi \mid \exists n' \in \mathcal{T}[a?n] : \mathcal{T} n' \Box_a \phi\}.$$

We can now specify what makes for a virtual loop and a redundant copy.

**Definition 3.16.** Given a tableau $\mathcal{T}$, a «virtual $a$-loop» (in $\mathcal{T}$) from $n_0$ to $n_l$ is a sequence $S = n_0, \dots, n_i, n_{i+1}, \dots n_l$ such that

- For every $n_{i+1}$ in the sequence it is the case that $\mathcal{T} a n_i n_{i+1}$.
- All links between nodes in the sequence are $a$-links.
- There is a set of formulas $F$ such that for every $n_i$ in the sequence, $\Box_a^{\mathcal{T}}(n_i) = F$.
- $n_l$ has no outgoing links.

**Figure 3.5.** If we were to iteratively $4_a$-develop the tableau $\mathcal{T}$ on the left in a naive way, we would be constructing a countably infinite tableau. However, notice that $S \coloneqq n, n'$ is a virtual $a$-loop and that $\mathcal{T}$ is in fact $\lfloor 4_a \rfloor$-saturated. Thus, given that $S$ is the only virtual loop in $\mathcal{T}$, we can construct a $4_a$-saturated tableau by folding $S$ in $\mathcal{T}$.



- All labels of $n_l$ are labels of $n_0$ (i.e. $\mathcal{T}[n_l?] \subseteq \mathcal{T}[n_0?]$).

A node $n'$ is called a «copy» of a node $n$ (in $\mathcal{T}$) if and only if there is a virtual $a$-loop (for some $a$) from $n$ to $n'$ (in $\mathcal{T}$).

Notice that the fact that $n_l$ has no outgoing links implies that $n_0 \neq n_l$.

As we will see shortly, in certain unsaturated tableaux it is necessary to replace the virtual loops by actual loops in order to ensure that the further development of these tableaux eventually terminates. To this end our general strategy is to block rules from adding outgoing links to copies. When the only rules that remain to be applied are those that would add outgoing links to copies, we transform the virtual loops into actual loops.

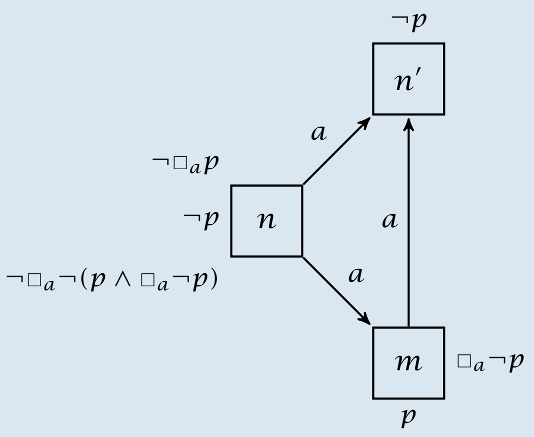
We now revise some notions relating to rule application such that rules do not add outgoing links to copies.

**Definition 3.17.** Given a tableau rule $R$, let «$\lfloor R \rfloor$» be such that for any two tableaux $\mathcal{T}$ and $\mathcal{T}'$ it is the case that $\lfloor R \rfloor \mathcal{T}\mathcal{T}'$ if and only if

- $R\mathcal{T}\mathcal{T}'$.
- There is no edge $\langle ann' \rangle \in \mathcal{T}'[???] - \mathcal{T}[???]$ where $n$ is a copy of another node in $\mathcal{T}$ (based on any $b$-loop).

A tableau $\mathcal{T}'$ is a «$\lfloor \sigma \rfloor$-development» of a tableau $\mathcal{T}$ if and only if it is either an element of or the limit of a sequence $\mathcal{T}, \ldots, \mathcal{T}_i, \mathcal{T}_{i+1}, \ldots$ such that for every $\mathcal{T}_{i+1}$ in the sequence there is a rule $R \in \text{Rules}_\sigma$ such that $\lfloor R \rfloor \mathcal{T}_i \mathcal{T}_{i+1}$ and $\mathcal{T}_i \neq \mathcal{T}_{i+1}$.

**Figure 3.6.** The tableau $\mathcal{T}$ below is but one edge away from being $(4_a \sqcup 5_a)$-saturated (although some labels have been omitted). Nonetheless, it's instructive to check if $S = n, n'$ is a virtual $a$-loop. The answer is no. $S$ is not a virtual $a$-loop because $\Box_a^{\mathcal{T}}(n) \neq \Box_a^{\mathcal{T}}(n')$.
Indeed, suppose that $S$ was folded. The resulting tableau would no longer have a node $n'$ and the $a$-edge from $m$ to $n'$ would have been replaced by an $a$-edge from $m$ to $n$. Thus, by $\mathbf{R}_4^{\mathbf{a}}$ an $a$-link from $m$ to $m$ would have to be added. Subsequently, by $\mathbf{R}_\Box$, the label $\neg p$ would have to be added to $m$. This, however, introduces a literal contradiction.



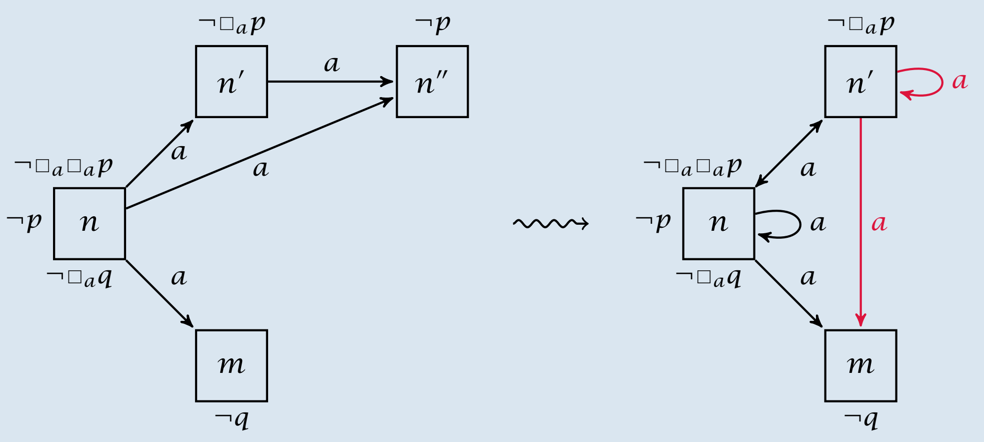
A tableau $\mathcal{T}$ is «$\lfloor\sigma\rfloor$-saturated» if and only if for all mandatory rules $R \in \text{Rules}_\sigma$ it is the case that $\mathcal{T}$ is $\lfloor R\rfloor$-saturated.

Finally, we define the embedding that transforms virtual $a$-loops into actual loops.

**Definition 3.18.** $\mathcal{T}'$ is the result of «folding» a virtual $a$-loop $S$ from $n$ to $n'$ in a tableau $\mathcal{T}$ if and only if, with $h : \mathcal{T}[?] \to \mathcal{T}[?] - \{n'\}$ such that $h(n') = n$ and $h(x) = x$ for all $x \neq n'$, it is the case that

- $\mathcal{T}'[?] = \mathcal{T}[?] - \{n'\}$
- $\mathcal{T}'[???] = \{\langle ah(m)h(m')\rangle \mid \mathcal{T}amm'\} \cup (\{a\} \times I \times O)$, where
  - $I \coloneqq \{h(m) \mid \exists n_i \in S : \mathcal{T}amn_i\}$
  - $O \coloneqq \{h(m) \mid \exists n_i \in S : \mathcal{T}an_im\}$

**Figure 3.7.** The tableau $\mathcal{T}$ on the left is a $4_a$-developed tableau. Notice that $\mathcal{T}$ has an $a$-loop $S = n, n', n''$. The tableau $\mathcal{T}'$ on the right is the folding of $S$ in $\mathcal{T}$.
$\mathcal{T}'$ has four new edges. The edges from $n$ and $n'$ to $n$ are straightforward to explain: They are a result of remapping $n''$ onto $n$. The edges from $n'$ to $n'$ and to $m$ are more interesting. They were added because (i) in $\mathcal{T}$ there was an $a$-edge from $n'$ to a node of $S$ (viz. $n''$) and (ii) in $\mathcal{T}$ there was an $a$-edge from a node of $S$ to $n'$ and from a node of $S$ to $n''$ (in both cases this node was $n$). Without these new edges $\mathcal{T}'$ would no longer have been closed under transitivity.



- $\mathcal{T}'[??] = \{\langle h(m)\phi\rangle \mid \mathcal{T} m \phi\}$

It is particularly important to notice that an $a$-edge is created between (i) every node $m_I$ that has an outgoing $a$-edge to a node of $S$ and (ii) every node $m_O$ that has an incoming $a$-edge that originates from a node of $S$. Figure 3.7 illustrates how this works.

The following insight is crucial, for it entails that by repeatedly applying folding, a $\lfloor\sigma\rfloor$-saturated tableau can be turned into a $\sigma$-saturated tableau.

**Lemma 3.11 (Folding Lemma).** If $\mathcal{T}'$ is the result of folding a virtual $a$-loop $S$ in a $\lfloor\sigma\rfloor$-saturated tableau $\mathcal{T}$ then $\mathcal{T}'$ is $\lfloor\sigma\rfloor$-saturated.

**Proof.** We need to show that for every mandatory rule $R \in \text{Rules}_\sigma$ it is the case that $\mathcal{T}'$ is $\lfloor R\rfloor$-saturated.

- If $R$ is $\mathbf{R}_\neg$, $\mathbf{R}_\wedge$, $\mathbf{R}_\vee$, $\mathbf{R}_\mathrm{U}$, or $\mathbf{R}_\mathrm{E}$ then $\mathcal{T}'$ is $\lfloor R\rfloor$-saturated since no labels were added or removed, and no nodes were added.

- Suppose $R$ equals $\mathbf{R}_\Box$ and is applied to the tuples $\langle bnn' \rangle$ and $\langle n\Box_b\phi \rangle$ in $\mathcal{T}'$. We distinguish two cases.
  - If $\mathcal{T}bnn'$ then it must already have been the case that $\mathcal{T}n'\phi$ since $\mathcal{T}$ is $\lfloor \mathbf{R}_\Box \rfloor$-saturated. This also deals with all cases where $b \neq a$ since only $a$-edges where added or moved.
  - Suppose $b = a$ and not $\mathcal{T}ann'$. It follows that $n$ has an $a$-link to an element of $S$ in $\mathcal{T}$ and that there is an element of $S$ that has an $a$-link to $n'$ in $\mathcal{T}$. This, however, implies that $\Box_a^{\mathcal{T}}(n) \cup \{\phi \mid \mathcal{T}n\Box_a\phi\} \subseteq \Box_a^{\mathcal{T}}(n')$. Consequently, there is nothing for $\lfloor \mathbf{R}_\Box \rfloor$ to do since $\mathcal{T}$ is already $\lfloor \mathbf{R}_\Box \rfloor$-saturated.
- Suppose $R$ is $\mathbf{R}_\Diamond$ and is applied to the node-label pair $\langle n\Diamond_b\phi \rangle$ in $\mathcal{T}'$. We distinguish three cases.
  - Suppose $n$ is not a copy in $\mathcal{T}$. Since $\mathcal{T}$ is $\lfloor \sigma \rfloor$-saturated this means that there is an $n'$ such that $\mathcal{T}bnn'$ and $\mathcal{T}n'\phi$. This edge and label would then also be in $\mathcal{T}'$ and hence there is nothing for $\lfloor \mathbf{R}_\Diamond \rfloor$ to do.
  - Suppose $n$ is not in the sequence $S$ but $n$ is a copy of another node. In this case $n$ must be part of a virtual $c$-loop $S'$.

    It is possible that a new link was created from a node $n''$ of $S$ to an element $n'$ of $S'$. However, in this case $\Box_a^{\mathcal{T}}(n'') \subseteq \Box_a^{\mathcal{T}}(n')$ since there must already have been a link from an element of $S$ to $n'$. Hence $n$ is still a copy in $\mathcal{T}'$.
  - Suppose $n$ is the last element of $S$ and thereby is a copy. $n$ is then remapped to the first element of $S$. This case is thereby analogous to the first case, where $n$ is not a copy.

    Notice that no other element of $S$ can be a copy.
- Suppose $R$ is $\mathbf{R}^{\mathbf{b}}_{\mathbf{T}}$. Trivially, because $\mathcal{T}$ is $\lfloor \sigma \rfloor$-saturated so is $\mathcal{T}'$.
- Suppose $R$ is $\mathbf{R}^{\mathbf{b}}_{\mathbf{D}}$. This is similar to the case $\mathbf{R}_\Diamond$.

- Suppose $R$ is $\mathbf{R}_4^\mathbf{b}$. If $\mathcal{T}'bnn'$ and $\mathcal{T}'bn'n''$ then we need to prove that $\mathcal{T}'bnn''$.

  - Suppose both $\mathcal{T}bnn'$ and $\mathcal{T}bn'n''$. It is then already the case that $\mathcal{T}bnn''$ since $\mathcal{T}$ is $\lfloor\mathbf{R}_4^\mathbf{b}\rfloor$-saturated and $n$ is not a copy.

    Notice that this case subsumes the case where $b \neq a$.

  - Suppose neither $\mathcal{T}ann'$ nor $\mathcal{T}an'n''$. By construction of $\mathcal{T}'$ it must then be the case that $n' \in S$ and that there is an $a$-link in $\mathcal{T}$ from $n$ to a node in $S$ and an $a$-link from a node in $S$ to $n''$.

    By construction of $\mathcal{T}'$ it follows that $\mathcal{T}'ann''$.

  - Suppose $\mathcal{T}ann'$ but not $\mathcal{T}an'n''$. It follows that there is an $a$-link from $n'$ to an element of $S$ and that there is an $a$-link in $\mathcal{T}$ from an element in $S$ to $n''$.

    By $\lfloor\mathbf{R}_4^\mathbf{a}\rfloor$ it follows that there is an $a$-link in $\mathcal{T}$ from $n$ to an element of $S$. By construction of $\mathcal{T}'$ it now follows that $\mathcal{T}'nn''$.

  - Suppose not $\mathcal{T}ann'$ but $\mathcal{T}an'n''$. It follows that there's an $a$-link from $n$ to an element of $S$ and that there is an $a$-link from an element of $S$ to $n'$.

    By $\lfloor\mathbf{R}_4^\mathbf{a}\rfloor$ it follows that there are $a$-links from elements of $S$ to $n''$ in $\mathcal{T}$. By construction of $\mathcal{T}'$ it now follows that $\mathcal{T}'nn''$.

- Suppose $R$ is $\mathbf{R}_\mathbf{B}^\mathbf{b}$. We need to demonstrate that if $\mathcal{T}'bnn'$ then $\mathcal{T}'bn'n$.

  Suppose $\mathcal{T}bnn'$. In this case $\mathcal{T}bn'n$ by $\lfloor\mathbf{R}_\mathbf{B}^\mathbf{b}\rfloor$. Hence $\mathcal{T}'bn'n$ by construction of $\mathcal{T}'$. This also covers the case where $b \neq a$.

  If not $\mathcal{T}ann'$ then there must be $\{m, m'\} \subseteq S$ such that $\mathcal{T}anm$ and $\mathcal{T}am'n'$. By $\lfloor\mathbf{R}_\mathbf{B}^\mathbf{a}\rfloor$ it is then also the case that $\mathcal{T}amn$ and $\mathcal{T}an'm'$.

  It follows by construction of $\mathcal{T}'$ that $\mathcal{T}'an'n$.

- Suppose $R$ is $\mathbf{R}_5^\mathbf{b}$. We have to show that if $\mathcal{T}'bnn'$ and $\mathcal{T}'bnn''$ then $\mathcal{T}'bn'n''$.

- Suppose $\mathcal{T}bnn'$ and $\mathcal{T}bnn''$. By $\lfloor \mathbf{R}_5^{\mathsf{b}} \rfloor$ it is then the case that $\mathcal{T}bn'n''$. Hence $\mathcal{T}'bn'n''$ by construction of $\mathcal{T}'$. This also covers all cases where $b \neq a$.
- Suppose neither $\mathcal{T}ann'$ nor $\mathcal{T}ann''$.

  By construction of $\mathcal{T}'$ it follows that there are $m'$ and $m''$ in $S$ such that $\mathcal{T}am'n'$ and $\mathcal{T}am''n''$. Moreover $m'$ (like $m''$) is not the last element of $S$ since it has outgoing links. This also implies that $m'$ has an $a$-link to an element of $m^\star$ of $S$. By $\lfloor \mathbf{R}_5^{\mathsf{a}} \rfloor$ it follows that $\mathcal{T}an'm^\star$.

  Finally, since there are $a$-links in $\mathcal{T}$ from $n'$ to an element of $S$ and from an element of $S$ to $n''$ it follows that $\mathcal{T}'an'n''$.
- Suppose $\mathcal{T}ann'$ but not $\mathcal{T}ann''$.

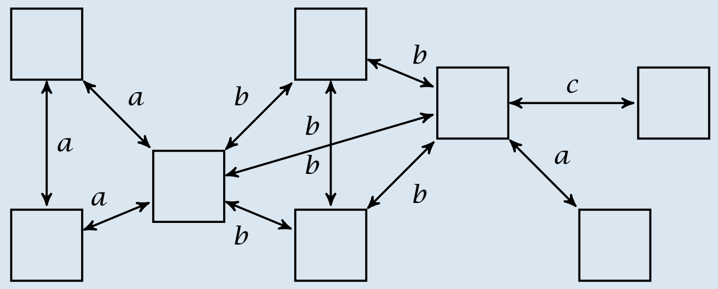
  By construction of $\mathcal{T}'$ it follows that there are $m$ and $m''$ in $S$ such that $\mathcal{T}anm$ and $\mathcal{T}am''n''$. By $\lfloor \mathbf{R}_5^{\mathsf{a}} \rfloor$ it also follows that $\mathcal{T}an'm$.

  It can now be seen that $\mathcal{T}'n'n''$ by construction of $\mathcal{T}'$.
- The case where $\mathcal{T}ann''$ but not $\mathcal{T}ann'$ is entirely analogous to the previous one.

The tableau method described in this chapter—including folding—is decidable. To understand why let us partition the mandatory tableau rules in two parts. The first kind of rules are those rules that add new formulas. These rules only propagate formulas that are shorter than the formulas that are already present in the tableau, however, and this places a bound on the number of distinct formulas in a tableau. The second kind of rules are rules that create new links and do not add new formulas. These rules only allow limited interaction between nodes that are connected over different indices, as is illustrated in figure 3.8. This implies that as formulas are propagated over differently indexed links, they decrease in length. This imposes a bound on the number of alternating indices in a chain of nodes. These insights lead to the following theorem.

**Theorem 3.12.** If there is a pointed $\sigma$-model $\langle \mathcal{M}w \rangle$ such that $\langle \mathcal{M}w \rangle \Vdash \phi$ (where $\phi \in \mathcal{L}^{U\Box}$) then an open saturated $\sigma$-tableau $\mathcal{T}$ for $\langle w\phi \rangle$ can be found in a finite number of steps.

**Figure 3.8.** Tableaux produced by the mandatory rules lead to clusters of nodes connected by the same index. Any two clusters can interact only to a limited extend via a single node that connects those two clusters. This is the case because every tableau rule contains at most one index. In the diagram below no new links (ignoring reflexive links) can be added using mandatory rules.



**Proof.** The procedure for finding the tableau is as follows. Start with the tableau $\mathcal{T} = \{\langle w\rangle, \langle w\phi\rangle\}$. Fairly apply the mandatory rules of $\text{Rules}_\sigma$ until an open $\lfloor\sigma\rfloor$-saturated tableau $\mathcal{T}'$ is found. Fair application means that all rules must be used (insofar they are applicable) and that it's not allowed to use a limited selection of rules to the exclusion of others. Next, repeatedly apply folding until there are no more virtual loops.

Notice that for all $\langle n\psi\rangle \in \mathcal{T}'$ it is the case that $|\psi| \leq |\phi|$. Hence there's only a finite number of possible label-sets.

Because there are only a finite number of possible label-sets, $\mathbf{R}_\Diamond$ can only attach a finite number of new links to any existing node $n$. The rule $\mathbf{R}^{\mathbf{a}}_{\mathbf{D}}$ is similarly conservative. Consequently, the only way to indefinitely expand a tableau using the mandatory rules of $\text{Rules}_\sigma$, is to create longer and longer chains of nodes.

However, the mandatory rules are such that for any two indices $a$ and $b$ such that $a \neq b$, if you take a set of nodes $A$ that are connected by $a$-edges and another set of nodes $B$ that are connected by $b$-edges then there is at most one node $n \in A$ such that there are $b$-edges between $n$ and nodes of $B$ (in either direction). In other words, differently indexed modalities do not interact.

Moreover, if such an $n \in A$ exists then there are no $a$-edges between nodes of $A$ and nodes of $B$. In fact, for any $c \neq b$ it is the case that there are no $c$-edges between nodes of $A$ and $B$. It follows that either

- For all labels $\phi$ of nodes in $B$ there is a label $\psi$ of a node in $A$ such that $|\psi| > |\phi|$. This happens when a node of $A$ was added before any of the nodes of $B$ were added.
- The same is true when substituting $A$ for $B$ and $B$ for $A$.

Thus there is a finite limit to the number of index alternations in a path.

It only remains to be shown that, for any index $a$, there is an upper bound to the length of $a$-paths before there are no more mandatory rules to apply or before a virtual $a$-loop is encountered.

It is easy enough to see that only a combination of $\mathbf{R}_\Diamond$ or $\mathbf{R}^{\mathbf{a}}_{\mathbf{D}}$, $\mathbf{R}_\Box$, and $\mathbf{R}^{\mathbf{a}}_4$ or $\mathbf{R}^{\mathbf{a}}_5$ could lead to problematic paths.

However, notice that the possible extensions for $\Box^{\mathcal{T}'}_a(n)$ are restricted by $\phi$ similar to how the possible label-sets are restricted by $\phi$. Moreover, if $\sigma$ includes $\mathbf{R}^{\mathbf{a}}_4$ or $\mathbf{R}^{\mathbf{a}}_5$ then $\Box^{\mathcal{T}'}_a(n) \subseteq \Box^{\mathcal{T}'}_a(n')$ if there's an $a$-path from $n$ to $n'$. Consequently, any sufficiently long $a$-path is repetitive in a way that can be dealt with by the process of folding.

## 3.5 Related work

Saul Kripke introduced tableau systems for modal logic alongside Kripke semantics in [26,27]. In these papers he presents three approaches for dealing with modal languages that do not have an operator $U$ and are not multiply indexed.

Kripke sometimes uses different terminology than we are using now. For instance, what we call a 'tableau' roughly corresponds to what Kripke calls an 'alternative set of tableaux'. In turn, the tableau nodes of our system correspond to Kripke's 'tableaux'. In our sketch of Kripke's tableau systems we use the same terminology used throughout this chapter and ignore Kripke's usage of the term 'tableau'.

Kripke's first approach is limited to models with equivalence frames. Due to this restriction all nodes in a tableau can be thought of as being implicitly linked, and thus there is no need for edges in tableau structures.

The second approach is similar to the path taken in this chapter. Kripke adds an accessibility relation $R$ to his tableau structures. The rules for $\Box$-formulas (and their negation) are similar to the rules in definition 3.7. There are no explicit rules for closing tableau frames under the desired frame conditions $\sigma$. Instead, Kripke merely stipulates that $R$ is closed under $\sigma$. The effect is almost the same. However, recall that our notions of $\lfloor\,\rfloor$-development and $\lfloor\,\rfloor$-saturation required blocking the creation of new links under certain conditions. This is not possible in Kripke's system. Tellingly, Kripke gives no decidable proof procedure for this tableau system. Instead he describes a third approach.

On Kripke's third approach there is again a relation $R$ but it is not closed under any relational properties. Hence $R$ is always a tree structure, which makes it easier to create a decidable proof procedure. Indeed, on this approach a loop is simply a sequence of two nodes $n, n'$ such that $Rnn'$ and such that every label of $n'$ is also a label of $n$. The downside of this approach is that the rule $\mathbf{R}_\Box$ needs to be modified in accordance with the chosen frame conditions. For example, this is the rule for serial and transitive tableaux: If $n$ has a label $\Box\phi$ then the label $\phi$ is added to $n$ and $\Box\phi$ is added to all nodes accessible from $n$. Finally, on this approach a stock model's frame is obtained by closing the tableau's frame under the desired frame conditions.

In the literature Kripke's third approach has perhaps won out over his second approach. There are other approaches but they widen the gap between Kripke models and tableau structures even further. See [10] for a general overview of tableau systems.

An interesting development occurs in [9,15]. In these papers tableau structures are represented by labeled graphs that are rooted and acyclic but not necessarily tree structures. This enables the authors to use a mixture of Kripke's second and third approach. They still use the third approach where convenient, but use rules of the second kind for seriality, denseness, and confluence. Using this approach they construct an extensible tableau system that can handle many combinations of frame conditions for modal languages without the operator $U$. In [14,19] a software application named Lotrec is presented that builds on these results. Lotrec implements an extensible tableau system and is able to automatically construct models and prove theorems.

This dissertation may be understood as a continuation of Kripke's second approach, using techniques from [9, 15]. With respect to the usability and ease of understanding of tableau systems, we believe this to be a superior approach.

Finally, the reader interested in learning more about modal logic might find it useful to start with [7, 25]. To learn about the history of modal logic, consult [21]. See [16] to learn more about proof systems for modal logic.

# 4 Public Announcement Logic

In the previous chapter we saw that modal logic can be interpreted in a variety of ways. In this chapter we discuss an extension of modal logic where the modal operators are interpreted in epistemic terms.

Public announcement logic (PAL) can be classified as a dynamic modal logic. It is 'dynamic' in two ways. First, it has an operator for representing events—public announcements—where all agents become aware of a certain fact. Thus the knowledge of agents in PAL is not static but dynamic—agents can learn. Second, the language of PAL $\mathcal{L}^{\square !}$ has a dynamic semantics. As before, static formulas of $\mathcal{L}^{\square !}$ are interpreted on Kripke models. Dynamic formulas, however, are reduced to simpler formulas which are then evaluated in models updated with the information that was publicly announced. These updated models are a function of the original model and the dynamic formula.

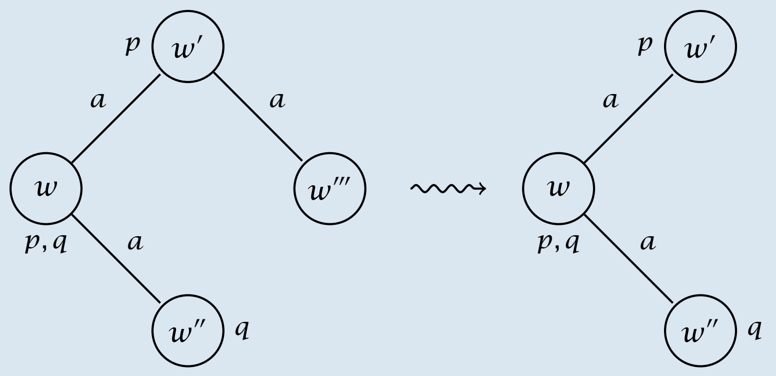
When discussing PAL it is customary to assume the frame conditions $\sigma$ are such that $\sigma(a)$ (where $a \in \mathrm{Ind}$) holds for a model $\mathcal{M}$ if and only if the $a$-frame of $\mathcal{M}$ is an equivalence frame. That is, it is assumed that $\sigma(a) = \mathrm{T}_a \sqcup 5_a$ or $\sigma(a) = \mathrm{T}_a \sqcup 4_a \sqcup \mathrm{B}_a$. There is, however, no technical requirement for this to be the case.

## 4.1 $\mathcal{L}^{\square !}$ and its dynamic semantics

**Definition 4.1.** The language of public announcements $\mathcal{L}^{\square !}$ is the set of formulas $\phi$ recursively defined as follows:

$$\phi ::= p \mid \neg\phi \mid (\phi \wedge \phi) \mid \square_a\phi \mid [!\phi]\phi,$$

**Figure 4.1.** The model $\mathcal{M}$ (left) and the updated model $\mathcal{M}|_{!(p\vee q)}$ (right). The updated model contains exactly those worlds $x$ such that $\langle \mathcal{M}x\rangle \Vdash (p \vee q)$. All model diagrams in this section should be interpreted as having transitive and reflexive $a$-edges.



with $p$ any element of the set of atomic proposition Prop and $a$ a member of the set of agents Ind.

Read $\Box_a\phi$ as 'agent $a$ knows that $\phi$' and read $[!\,\phi]\psi$ as '$\psi$ is true after it is truthfully and publicly announced that $\phi$ is the case'.

Implicitly, we defined a forcing relation in definition 3.3 not only for $\mathcal{L}^{U\Box}$, but for any language that has compositional semantics that fit in the following schema:

$$\phi ::= p \mid \neg p \mid (\phi \wedge \phi) \mid U\phi \mid \Box_a\phi \mid \dots,$$

where $p \in$ Prop and $a \in$ Ind. In other words, our definition for $\Vdash$ was open ended. As such, we now define the semantics for $\mathcal{L}^{\Box!}$ by retroactively adding one more constraint to this relation.

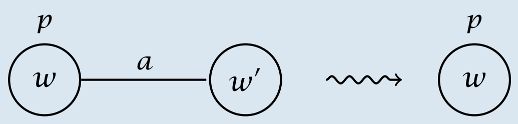
**Definition 4.2.** The forcing relation «$\Vdash$» is a relation that meets the stipulations of definition 3.3 in addition to the following constraint:

$$\langle \mathcal{M}w\rangle \Vdash [!\,\phi]\psi \iff \text{if } \langle \mathcal{M}w\rangle \Vdash \phi \text{ then } \langle \mathcal{M}|_{!\phi}w\rangle \Vdash \psi,$$

where

$$\mathcal{M}|_{!\phi} \coloneqq \mathcal{M} \cap (\{\langle w'\rangle \mid w' \in W'\} \cup (W' \times \text{Prop}) \cup (\text{Ind} \times W' \times W'))\,,$$

**Figure 4.2.** The very act of announcing a formula $\phi$ can sometimes make $\phi$ false. For instance, the Moore sentence '$p$, but you don't know that $p$' is false after it is truthfully announced. To wit, after I inform you that, even though you don't know it, it is raining outside, you will in fact know that it is raining outside. This phenomenon can be expressed in $\mathcal{L}^{\Box !}$. Let $\mathcal{M}$ be the model on the left. It is the case that $\langle \mathcal{M}w \rangle \Vdash (p \wedge \neg\Box_a p)$. However, it is not the case that $\langle \mathcal{M}w \rangle \Vdash [!(p \wedge \neg\Box_a p)](p \wedge \neg\Box_a p)$ since $\langle \mathcal{M}|_{!(p\wedge\neg\Box_a p)} w \rangle \Vdash \Box_a p$.



and

$$W' \coloneqq \{w' \in \mathcal{M}[?] \mid \langle \mathcal{M}w' \rangle \Vdash \phi\}.$$

Intuitively, $\mathcal{M}|_{!\phi}$ is the model that results from deleting all worlds in which $\phi$ does not hold.

## 4.2 Dynamic tableaux

We now extend the tableau system of $\mathcal{L}^{U\Box}$ in three steps.

First, we add a tableau rule for public announcements.

**Definition 4.3.** Let Rules and Rules$_\sigma$ be as in definition 3.7, except they also contain the tableau rule $\mathbf{R}_!$ from table 4.1.

Second, we define the relation $\xrightarrow{!\phi}$ between tableaux. This relation serves as a counterpart of $|_{!\phi}$ in the sense that $\mathcal{T} \xrightarrow{!\phi} \mathcal{T}'$ signifies that $\mathcal{T}'$ is the result of updating $\mathcal{T}$ with the fact that $\phi$ was publicly revealed.

**Definition 4.4.** Given a wff $\phi$ and two tableau $\mathcal{T}$ and $\mathcal{T}'$, let «$\mathcal{T} \xrightarrow{!\phi} \mathcal{T}'$» if and only if

- $\mathcal{T}'[?] = \mathcal{T}[?\phi]$
- $\mathcal{T}'[???] = \mathcal{T} \cap (\mathrm{Ind} \times \mathcal{T}[?\phi] \times \mathcal{T}[?\phi])$

**Table 4.1.** Public announcement logic necessitates one new tableau rule. Upon encountering a formula $[!\phi]\psi$ or $\neg[!\phi]\psi$ anywhere in the tableau, the rule $\mathbf{R}_!$ forces every other node to declare that $\phi$ or $\neg\phi$ should be true.

| Name | Precondition | Postcondition |
|---|---|---|
| $\mathbf{R}_!$ | $\mathcal{T}n$ and $\mathcal{T}n'[!\phi]\psi$ | $\mathcal{T}n\phi$ or $\mathcal{T}n\neg\phi$ |
| | $\mathcal{T}n$ and $\mathcal{T}n'\neg[!\phi]\psi$ | $\mathcal{T}n\phi$ or $\mathcal{T}n\neg\phi$ |

- $\mathcal{T}'[??] \cap (\mathcal{T}[?\phi] \times \mathrm{Prop}) = \mathcal{T}[??] \cap (\mathcal{T}[?\phi] \times \mathrm{Prop})$

The last line states that the atoms of the two tableaux must match in the nodes that survive the announcement.

Third, we extend the definition of ‘open tableau’ to cover cases where tableaux have dynamic formulas.

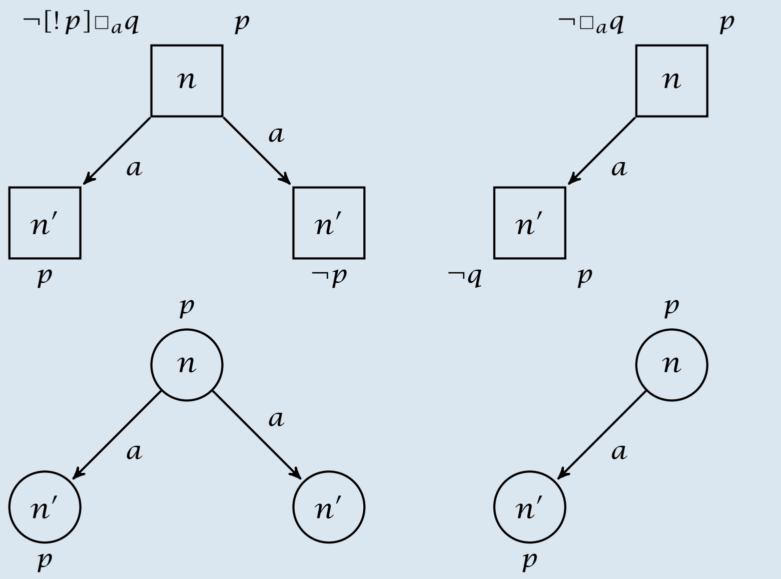
**Definition 4.5.** An $\mathcal{L}^{\Box !}$-tableau $\mathcal{T}$ is «open» if and only if

- $\mathcal{T}$ is free of literal contradictions.
- If $\mathcal{T}n[!\phi]\psi$ and $\mathcal{T}n\phi$ then there are no open saturated tableaux $\mathcal{T}'$ for $\langle n\neg\psi\rangle$ such that $\mathcal{T} \xrightarrow{!\phi} \mathcal{T}'$.
- If $\mathcal{T}n\neg[!\phi]\psi$ then $\mathcal{T}n\phi$ and there is an open saturated tableau $\mathcal{T}'$ for $\langle n\neg\psi\rangle$ such that $\mathcal{T} \xrightarrow{!\phi} \mathcal{T}'$.

Having just specified the tableau system for $\mathcal{L}^{\Box !}$, let us reflect on the relation $\xrightarrow{!\phi}$. A casual glance tells us this construction is remarkably similar to the operation $|_{!\phi}$. In order to better understand exactly how they relate, we first isolate tableaux in which every node contains either $\phi$ or its negation.

**Definition 4.6.** A tableau $\mathcal{T}$ is «$\phi$-declarative» if and only for every $n \in \mathcal{T}[?]$ it is the case that $\mathcal{T}n\phi$ or $\mathcal{T}n\neg\phi$.

**Figure 4.3.** The diagrams on the left represent a saturated open tableau $\mathcal{T}$ for $\langle n \neg [!\,p]\Box_a q\rangle$ and its stock model $\mathcal{M}$. The diagrams on the right depict a saturated open tableau $\mathcal{T}'$ for $\langle n \neg \Box_a q\rangle$ and its stock model $\mathcal{M}'$. Notice that $\mathcal{T} \xrightarrow{!p} \mathcal{T}$ and $\mathcal{M}' = \mathcal{M}|_{!p}$.



The following lemma goes a long way to prove that $\xrightarrow{!\phi}$ holds between any two $\phi$-declarative tableaux if and only if the stock model of the second tableau is the result of applying the operation $|_{!\phi}$ to the stock model of the first tableau. There is one minor caveat: We require a warrant that the first tableau is natively satisfied by its stock model. We need this guarantee because we have not yet demonstrated completeness for $\mathcal{L}^{\Box !}$-tableaux (insofar they contain dynamic formulas). On the contrary, we will use this lemma to prove soundness and completeness.

**Lemma 4.1.** If $\mathcal{M}$ is a stock model of a $\phi$-declarative tableau $\mathcal{T}$ such that $\mathcal{M}$ natively satisfies $\mathcal{T}$ up to $\neg\phi$ then it holds that $\mathcal{M}|_{!\phi}$ is the stock model for $\mathcal{T}'$ if and only if $\mathcal{T} \xrightarrow{!\phi} \mathcal{T}'$.

**Proof.** Since $\mathcal{T}$ is $\phi$-declarative either $\mathcal{T} n\phi$ or $\mathcal{T} n\neg\phi$ for every $n \in \mathcal{T}[?]$. If $\mathcal{T} n\phi$ then $\langle \mathcal{M} n\rangle \Vdash \phi$ since $\mathcal{M}$ natively satisfies $\mathcal{T}$ up to $\neg\phi$; if $\mathcal{T} n\neg\phi$ then $\langle \mathcal{M} n\rangle \Vdash \neg\phi$ by the same reasoning. Because by definition of $\neg$ it is not possible for a formula and its negation to be true in a single world, it follows that $\forall n \in \mathcal{T}[?] : \mathcal{T}n\phi \iff \langle \mathcal{M}n \rangle \Vdash \phi$.

On the left-to-right reading of the lemma it is assumed that $\mathcal{M}|_{!\phi}$ is a stock model of $\mathcal{T}'$. Hence $\mathcal{T}'[?] = \{w \in \mathcal{M}[?] \mid \langle \mathcal{M}w \rangle \Vdash \phi\}$. By the result in the previous paragraph it follows that $\mathcal{T}'[?] = \mathcal{T}[?\phi]$. Since $\mathcal{M}|_{!\phi}$ is the stock model of $\mathcal{T}'$ it also follows that $\mathcal{T}'[???] = \mathcal{T}[???] \cap (\mathrm{Ind} \times \mathcal{T}[?\phi] \times \mathcal{T}[?\phi])$. Finally, for all $n \in \mathcal{T}[?\phi]$ and all atoms $p$ it is the case that $\mathcal{T}np \iff \mathcal{T}'np$ since $\mathcal{M}$ is the stock model of $\mathcal{T}$. Consequently, $\mathcal{T} \xrightarrow{!\phi} \mathcal{T}'$.

On the right-to-left reading of the lemma it is assumed that $\mathcal{T} \xrightarrow{!\phi} \mathcal{T}'$. It follows that $\mathcal{T}'[?] = \mathcal{T}[?\phi]$, $\mathcal{T}'[???] = \mathcal{T}[??] \cap (\mathrm{Ind} \times \mathcal{T}[?\phi] \times \mathcal{T}[?\phi])$, and $\mathcal{T}[??] \cap (\mathcal{T}[?\phi] \times \mathrm{Prop}) = \mathcal{T}[??] \cap (\mathcal{T}[?\phi] \times \mathrm{Prop})$. By the results from the first paragraph and the fact that $\mathcal{M}$ is the stock model of $\mathcal{T}$ it is also the case that $\mathcal{T}'[?] = \{w \in \mathcal{M}[?] \mid \langle \mathcal{M}w \rangle \Vdash \phi\}$, that $\mathcal{T}'$ has the same frame as $\mathcal{M}|_{!\phi}$, and that for all $n \in \mathcal{T}'[?]$ and $p \in \mathrm{Prop}$ it is the case that $\mathcal{T}'np \iff \mathcal{M}|_{!\phi}np$. Thus $\mathcal{M}|_{!\phi}$ is the stock model of $\mathcal{T}'$.

In the previous chapter, lemma 3.7 did the heavy lifting with respect to proving soundness. We now extend this result to cover $\mathbf{R}_!$.

**Lemma 4.2.** If a $\sigma$-model $\mathcal{M}$ satisfies a tableau $\mathcal{T}$ via a function $f$ then it is the case that, if $\mathcal{T}$ is not $\mathbf{R}_!$-saturated then there is a tableau $\mathcal{T}' \in \mathbf{R}_![\mathcal{T}] - \{\mathcal{T}\}$ and a function $f'$ such that $\mathcal{M}$ satisfies $\mathcal{T}'$ via $f'$.

**Proof.** Suppose $\mathbf{R}_!$ is triggered by $\mathcal{T}n[!\phi]\psi$ (or $\mathcal{T}n\neg[!\phi]\psi$) and $\mathcal{T}n'$. If $\langle \mathcal{M}f(n) \rangle \Vdash \phi$ then let $\mathcal{T}' \coloneqq \mathcal{T} \cup \langle n'\phi \rangle$; otherwise let $\mathcal{T}' \coloneqq \mathcal{T} \cup \langle n'\neg\phi \rangle$. Since $\mathbf{R}_!$ does not add new nodes, let $f' \coloneqq f$. Evidently, $\mathcal{M}$ satisfies $\mathcal{T}'$ via $f$.

The above lemma plays a somewhat smaller role in the soundness proof for $\mathcal{L}^{\Box !}$-tableaux, however. The reason for that is that in this chapter a tableau's status as 'open' no longer merely depends on it being void of literal contradictions; it also depends on the existence (or non-existence) of other tableaux.

In a bit we will tackle these extended requirements as we round off the soundness and completeness proofs. We will address these requirements in tandem and use mutual induction to prove soundness and completeness of the dynamic fragment of $\mathcal{L}^{\Box !}$. First, however, we introduce a new concept that mirrors the notion of 'stock models' by mapping models onto tableaux.

**Definition 4.7.** Given a pointed $\sigma$-model $\langle \mathcal{M} w \rangle$ such that $\langle \mathcal{M} w \rangle \Vdash \phi$ we say that the tableau $\mathcal{T}$ is a «stock tableau» for $\langle \mathcal{M} w \phi \rangle$ if and only if $\mathcal{M}$ is a stock model for $\mathcal{T}$ and $\mathcal{T}$ is a saturated $\sigma$-tableau for $\langle w \phi \rangle$ such that $\mathcal{M}$ natively satisfies $\mathcal{T}$.

**Proposition 4.3.** If $\langle \mathcal{M} w \rangle$ is a pointed $\sigma$-model such that $\langle \mathcal{M} w \rangle \Vdash \phi$ then there is a stock tableau for $\langle \mathcal{M} w \phi \rangle$.

**Proof.** First, let $\mathcal{T} \coloneqq \{\langle w \rangle, \langle w \phi \rangle\}$. Trivially, $\mathcal{T}$ is natively satisfied by $\mathcal{M}$. Second, develop $\mathcal{T}$ into a saturated tableau $\mathcal{T}'$ that is satisfied via a function $f$ (as per lemma 4.2).

Third, let $\mathcal{T}''$ be the tableau $\{\langle f(n) \rangle \mid n \in \mathcal{M}[?]\} \cup \mathcal{M}[???] \cup \{\langle f(n) \phi \rangle \mid \mathcal{T}' n \phi\}$. It is easy to see that $\mathcal{M}$ natively satisfies $\mathcal{T}''$. Moreover, embeddings do not affect $R$-saturation for rules $R$ that are triggered by the presence of formulas in a tableau. Finally, $\mathcal{T}''$ is a $\sigma$-tableau and is $R$-saturated for the frame rules in $\text{Rules}_\sigma$ because $\mathcal{M}$ is a $\sigma$-model.

Everything is now in place to prove that the tableau system for $\mathcal{L}^{\Box !}$ is sound and complete. As mentioned above, the bulk of these proofs are based on mutual induction.

**Lemma 4.4 (Lemma to Soundness and Completeness).** Let $\mathcal{M}$ be the stock model of a $\sigma$-saturated tableau $\mathcal{T}$.

1. If $\mathcal{M}$ natively satisfies $\mathcal{T}$ and $\mathcal{T}$ is a tableau for $\phi \in \mathcal{L}^{\Box !}$ then $\mathcal{T}$ is open.
2. If $\mathcal{T}$ is open and $\mathcal{T} n \phi$ then $\langle \mathcal{M} n \rangle \Vdash \phi$.

**Proof.** Proof by mutual induction on the length of $\phi$. Assume that the lemma holds for all $\psi$ that are shorter than $\phi$.

**Part 1.** We address the requirements of the extended definition of 'open tableau' one by one:

- In lemmas 3.7 and 4.2 it is shown that $\mathcal{T}$ is free of literal contradictions. These lemmas also handle the base cases where $\phi \in \mathrm{Lit}$.
- For all $\langle n[!\psi]\chi \rangle \in \mathcal{T}$ it must be demonstrated that if $\mathcal{T} n \psi$ then there is no open saturated tableau $\mathcal{T}'$ for $\langle n \neg \chi \rangle$ such that $\mathcal{T} \xrightarrow{!\psi} \mathcal{T}'$.

  Suppose such a tableau $\mathcal{T}'$ did exist. By Part 2 of the IH it is then the case that, with $\mathcal{M}'$ the stock model of $\mathcal{T}'$, $\langle \mathcal{M}' n \rangle \Vdash \neg\chi$. By $\mathbf{R}_!$, since $\mathcal{T}$ is $\sigma$-saturated, it is the case that $\mathcal{T}$ is $\psi$-declarative. By lemma 4.1 it now follows that $\mathcal{M}' = \mathcal{M}|_{!\psi}$. However, since $\mathcal{M}$ natively satisfies $\mathcal{T}$ it is the case that $\langle \mathcal{M} n \rangle \Vdash \psi$. This contradicts the assumption that $\langle \mathcal{M} n \rangle \Vdash [!\psi]\chi$.
- For all $\langle n \neg [!\psi]\chi \rangle \in \mathcal{T}$ it must be demonstrated that $\mathcal{T} n \psi$ and that there is an open saturated tableau $\mathcal{T}'$ for $\langle n \neg \chi \rangle$ such that $\mathcal{T} \xrightarrow{!\psi} \mathcal{T}'$.

  Because $\mathcal{M}$ natively satisfies $\mathcal{T}$ it is the case that $\langle \mathcal{M} n \rangle \Vdash \psi$ and that $\langle \mathcal{M}|_{!\psi} n \rangle \Vdash \neg\chi$. By Part 1 of the IH it follows that any stock tableau $\mathcal{T}'$ for $\langle \mathcal{M}|_{!\psi} n \neg \chi \rangle$ is open. Also notice that $\mathcal{T}$ is $\psi$-declarative because of $\mathbf{R}_!$. By lemma 4.1 it follows that $\mathcal{T} \xrightarrow{!\psi} \mathcal{T}'$ and this concludes our proof.

**Part 2.** Most of this proof has already been covered in lemma 3.9, which can be interpreted as proceeding by induction on Part 2 of the current IH. In what follows we only deal with the patterns that are new to $\mathcal{L}^{\Box !}$.

- Suppose $\phi$ is $[!\psi]\chi$. It has to be demonstrated that if $\langle \mathcal{M} n \rangle \Vdash \psi$ then $\langle \mathcal{M}|_{!\psi} n \rangle \Vdash \chi$. The proof proceeds by contradiction. Assume that $\langle \mathcal{M} n \rangle \Vdash \psi$ and $\langle \mathcal{M}|_{!\psi} n \rangle \Vdash \neg\chi$.

  By Part 1 of the IH it follows that the stock tableau $\mathcal{T}'$ for $\langle \mathcal{M}|_{!\psi} n \neg \chi \rangle$ is open.

  Because of $\mathbf{R}_!$ it is the case that $\mathcal{T}$ is $\psi$-declarative and by Part 2 of the IH it follows that $\mathcal{M}$ natively satisfies $\mathcal{T}$ up to $\neg\psi$. By lemma 4.1 it now follows that $\mathcal{T} \xrightarrow{!\psi} \mathcal{T}'$.

However, since $\mathcal{T}$ is an open tableau we know that there is no saturated open tableau $\mathcal{T}''$ for $\langle n\neg\chi\rangle$ such that $\mathcal{T} \xrightarrow{!\psi} \mathcal{T}''$. This concludes the proof by contradiction.

- Suppose $\phi$ is $\neg[!\,\psi]\chi$. We need to show that $\langle \mathcal{M}n\rangle \Vdash \psi$ and $\langle \mathcal{M}|_{!\psi}\rangle \Vdash \neg\chi$.

  Since $\mathcal{T}$ is open it follows that $\mathcal{T}n\psi$ and that there is an open saturated tableau $\mathcal{T}'$ for $\langle n\neg\chi\rangle$ such that $\mathcal{T} \xrightarrow{!\psi} \mathcal{T}'$.

  Notice that $\mathcal{T}$ is $\psi$-declarative by $\mathbf{R}_!$. Additionally, by Part 2 of the IH it also follows that $\mathcal{M}$ natively satisfies $\mathcal{T}$ up to $\neg\psi$. By lemma 4.1 it is now the case that $\mathcal{M}|_{!\psi}$ is a stock model of $\mathcal{T}'$.

  Finally, by Part 2 of the IH it is the case that $\langle \mathcal{M}n\rangle \Vdash \psi$ and $\langle \mathcal{M}|_{!\psi}n\rangle \Vdash \neg\chi$.

Notice that where Part 1 and Part 2 make use of lemma 4.1, they either use a combination of (i) induction on Part 1 and a left-to-right reading of lemma 4.1 or (ii) induction on Part 2 and a right-to-left reading of lemma 4.1. This means that lemma 4.1 could just as well be incorporated in the above lemma, although the resulting lemma would arguably be convoluted. This does, however, explain the surprising requirement of lemma 4.1 that the first tableau must have been demonstrated to be natively satisfied by its stock model.

At this point proving soundness and completeness is a straightforward matter.

**Theorem 4.5 (Soundness).** If all $\sigma$-saturated tableaux for $\phi \in \mathcal{L}^{\Box!}$ are closed then $\vDash_\sigma \neg\phi$.

**Proof.** As before, we prove the contrapositive—namely, that if there is a pointed $\sigma$-model $\langle \mathcal{M}w\rangle$ such that $\langle \mathcal{M}w\rangle \Vdash \phi$ then there's an open saturated $\sigma$-tableau $\mathcal{T}$ for $\langle w\phi\rangle$.

By Part 1 of lemma 4.4 the stock tableau for $\langle \mathcal{M}w\phi\rangle$ is open. This concludes our proof.

**Theorem 4.6 (Completeness).** If $\vDash_\sigma \neg\phi$ (with $\phi \in \mathcal{L}^{\Box !}$) then all saturated $\sigma$-tableaux $\mathcal{T}$ for $\phi$ are closed.

**Proof.** The contrapositive states that if there is an open $\sigma$-saturated tableau $\mathcal{T}$ for $\langle n\phi\rangle$ then there's a $\sigma$-model $\mathcal{M}$ such that $\langle \mathcal{M}n\rangle \Vdash \phi$.

By Part 2 of lemma 4.4 it indeed follows that $\phi$ holds in $n$ in the stock model of $\mathcal{T}$.

## 4.3 Tableau cascades and decidability

How do we decide, for any given tableau $\mathcal{T}$, if there's another tableau $\mathcal{T}'$ such that $\mathcal{T} \xrightarrow{!\phi} \mathcal{T}'$? This concern makes searching for tableaux for dynamic formulas quite the different enterprise from searching for $\mathcal{L}^{U\Box}$-tableaux.

Multiple approaches are conceivable. We discuss two approaches with different trade-offs.

In both approaches, upon encountering a formula $\neg[!\,\phi]\psi$ in a node, we add $\phi$ to the node and create a new tableau for $\neg\psi$. We then try and see if this new tableau can be developed into a saturated open tableau that stands in the relation $\xrightarrow{!\phi}$ to the original tableau. Similarly, when we find formulas $[!\,\phi]\psi$ and $\phi$ in a node, we create a tableau for $\psi$ and try to develop it into a saturated open tableau that stands in a $\xrightarrow{!\phi}$ relation to the original tableau. This works because $|_{!\phi}$ always yields exactly one model; by lemma 4.1 this means we only need to find one open tableau.

The question remains, however, how we can develop two tableaux such that they are saturated, open, and stand in a relation $\xrightarrow{!\phi}$ to one another. What's clear is that this will typically require the use of the tableau rule $\mathbf{R}_\star$ (table 3.2), with the caveat that this makes finding a decidable proof procedure non-obvious.

On the first approach we forgo the use of a rigid proof procedure. Instead we can let our intuition advise us on how to apply $\mathbf{R}_\star$. Given two tableaux $\mathcal{T}_1$ and $\mathcal{T}_2$ it is sometimes quite easy to find tableaux $\mathcal{T}_1' \supset \mathcal{T}_1$ and $\mathcal{T}_2' \supset \mathcal{T}_2$ such that $\mathcal{T}_1' \xrightarrow{!\phi} \mathcal{T}_2'$. Using our intuition we might very well be able to prove that a formula is satisfiable. However, we cannot prove that a formula is unsatisfiable—at best we can show that our intuition failed to find a tableau that satisfies it.

In the remainder of this section we will discuss a second approach, which involves a more mechanical proof procedure. This procedure has the property that if a formula is satisfiable then we can find a saturated and demonstrably open tableau for it in a finite number of steps. That is to say, the procedure is decidable.

To start, we define a tree structure for keeping track of the different tableaux we are developing.

**Definition 4.8.** A «tableau cascade» $C$ is an lgraph such that

- Every node has exactly one label and this label is a tableau.
- The indices of the edges are relations $\xrightarrow{!\phi}$ as defined in definition 4.4.

An edge $\langle \xrightarrow{!\phi} tt' \rangle \in C$ signals the intent to transform $C$ into a tableau cascade $C'$ such that $C'(t?) \xrightarrow{!\phi} C'(t'?)$.

We first encountered the notion that models satisfy certain other constructs in definition 3.3. We stipulated that a pointed model $\langle \mathcal{M}w \rangle$ satisfies a wff $\phi$ if and only if $\langle \mathcal{M}w \rangle \Vdash \phi$. In definition 3.13 we generalized this notion, saying that $\mathcal{M}$ satisfies a tableau $\mathcal{T}$ via a function $f$ if for all $\langle n\psi \rangle \in \mathcal{T}$ it is the case that $\langle \mathcal{M}f(n) \rangle \Vdash \psi$. We now abstract this concept even further by defining what it means for a model to satisfy a tableau cascade.

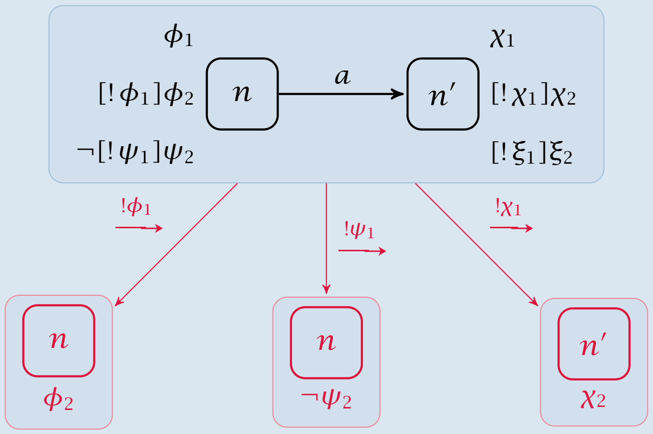
**Definition 4.9.** A pointed tableau cascade $\langle Ct \rangle$ is «satisfied» by a model $\mathcal{M}$ via a function $f$ if and only if

- $\mathcal{M}$ satisfies $C(t?)$ via $f$.
- For every $\langle \xrightarrow{!\phi} t' \rangle \in C[?t?]$ it is the case that $\mathcal{M}|_{!\phi}$ satisfies $\langle Ct' \rangle$ via $f$.

We say that a tableau cascade $C$ is satisfied by a model $\mathcal{M}$ via a function $f$ if and only if $\langle C\,\mathrm{root}(C) \rangle$ is satisfied by $\mathcal{M}$ via $f$.

There are three kinds of operations that we want to perform on tableau cascades. The first kind of operation merely constitutes applying a tableau rule to a node in a tableau cascade.

**Figure 4.4.** The diagram below shows what happens when the tableau cascade that consists solely of the top tableau is primed—viz. the three bottom tableaux are added to the tableau cascade. The leftmost one is added as a result of $\phi_1$ and $[!\phi_1]\phi_2$ being labels for $n$ in the top tableau. The label $\neg[!\psi_1]\psi_2$ for $n$ gives rise to the middle tableau. The rightmost tableau is created because $n'$ contains $\chi_1$ and $[!\chi_1]\chi_2$. Finally, the label $[!\xi_1]\xi_2$ for $n'$ does not result in a new tableau because $n'$ does not contain $\xi_1$.



**Definition 4.10.** $C'$ is an outcome of «applying» a tableau rule $R$ to $t$ in a tableau cascade $C$ if and only if $RC(t?)C'(t?)$ and $C' - \langle tC'(t?)\rangle = C - \langle tC(t?)\rangle$.

**Lemma 4.7.** Given a tableau cascade $C$ that is satisfied by a model $\mathcal{M}$ via a function $f$, if $R$ is a tableau rule and the tableau $C(t?)$ is not $R$-saturated then there is an outcome $C' \neq C$ of applying $R$ to $t$ in $C$ such that $C'$ is satisfied by $\mathcal{M}$ via some function $f'$.

**Proof.** This is a direct result of lemmas 3.7 and 4.2.

The second kind of operation looks for public announcements in the different tableaux in the cascade and adds new tableaux to the cascade as a result.

**Definition 4.11.** $C'$ is the result of «!-priming» a tableau cascade $C$ if and only if $C'$ is a minimal extension of $C$ (and the tableaux therein) such that for all $\langle t\mathcal{T}\rangle \in C$ and $\langle n\phi\rangle \in \mathcal{T}$,

- If $\phi = [!\psi]\chi$ and $\mathcal{T}n\psi$ then there is a node-tableau pair $\langle t'\mathcal{T}'\rangle \in C'$ such that $\mathcal{T}'$ is a tableau for $\langle n\chi\rangle$ and $\langle \xrightarrow{!\psi} tt'\rangle \in C'$.

- If $\phi = \neg[!\psi]\chi$ then $\mathcal{T}n\psi$ and there is a node-tableau pair $\langle t'\mathcal{T}'\rangle \in C'$ such that $\mathcal{T}'$ is a tableau for $\langle n\neg\chi\rangle$ and $\langle \xrightarrow{!\psi} tt'\rangle \in C'$.

**Lemma 4.8.** Given a tableau cascade $C$ that is satisfied by a model $\mathcal{M}^\star$ via a function $f$, if $C'$ is the result of !-priming $C$ then $C'$ is satisfied by $\mathcal{M}^\star$ via $f$.

**Proof.** We need to prove that for all new nodes $t'$ that are accessible from a node $t$ over $\xrightarrow{!\phi}$ in $C'$, if $\mathcal{T} \coloneqq C(t?)$ is satisfied by $\mathcal{M}$ via $f$ then $\mathcal{T}' \coloneqq C'(t'?)$ is satisfied by $\mathcal{M}|_{!\phi}$ via $f$.

We distinguish the two cases that could have led to this situation:

1. $\mathcal{T}n[!\phi]\psi$, $\mathcal{T}n\phi$, and $\mathcal{T}' = \{\langle n\rangle, \langle n, \psi\rangle\}$.

   It is then the case that $\langle \mathcal{M}f(n)\rangle \Vdash \phi$ and $\langle \mathcal{M}|_{!\phi}f(n)\rangle \Vdash \psi$. Hence $\mathcal{M}|_{!\phi}$ satisfies $\mathcal{T}'$ via $f$.

2. $\mathcal{T}n\neg[!\phi]\psi$ and $\mathcal{T}' = \{\langle n\rangle, \langle n, \neg\psi\rangle\}$.

   It is then the case that $\langle \mathcal{M}f(n)\rangle \Vdash \phi$ and $\langle \mathcal{M}|_{!\phi}f(n)\rangle \Vdash \neg\psi$ by definition of $\neg$ and $[!\phi]$. Hence $\mathcal{M}|_{!\phi}$ satisfies $\mathcal{T}'$ via $f$.

The third and final type of operation ensures that the different tableaux in a tableau cascade can be properly compared by dynamic relations $\xrightarrow{!\phi}$. This means copying edges and atomic propositions between tableaux and ensuring that only $\phi$-nodes 'survive' an update.

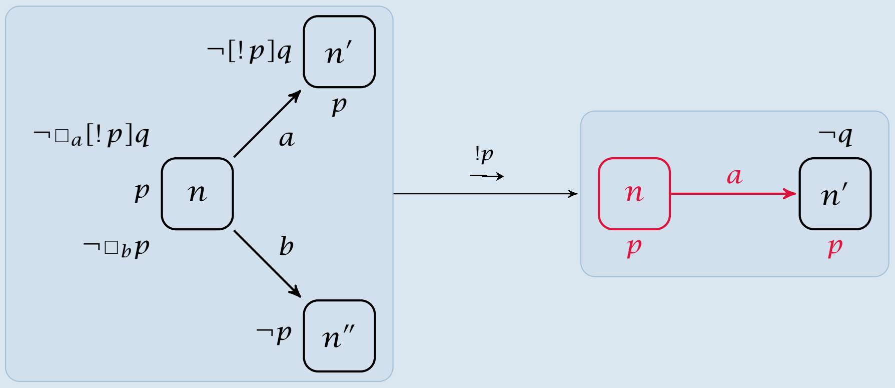
**Definition 4.12.** $C'$ is the result of «synchronizing» a tableau cascade $C$ if and only if

- $C$ and $C'$ have the same frame.

- $C'$ is an extension of $C$ in the sense that $\forall t \in C[?] : C(t?) \subseteq C'(t?)$.

- The set $\{\langle tx\rangle \mid t \in C[?] \text{ and } x \in C'(t?) - C(t?)\}$ is a minimal set (ordered by the subset relation) such that for some edge $\langle \xrightarrow{\pi} t_1t_2\rangle \in C$ it is the case that $C'(t_1?) \xrightarrow{\pi} C'(t_2?)$ but not $C(t_1?) \xrightarrow{\pi} C(t_2?)$.

**Figure 4.5.** Forward propagation of nodes, edges, and atoms. Suppose $p$ was added to $n$ and $n'$ by $\mathbf{R}_!$ in the tableau on the left. By synchronizing the tableau cascade, the node $n$ is copied from the tableau on the left to the tableau on the right because it contains $p$. Atoms and edges are also copied from the left to the right insofar they concern surviving nodes.



We also say that a tableau cascade is «fully synchronized» if and only if it cannot be synchronized.

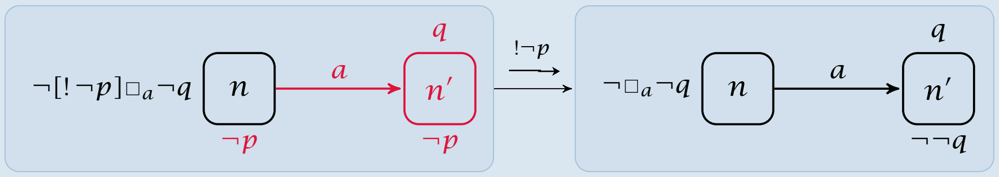
**Proposition 4.9.** A synchronization operation affects one or two tableau cascade nodes.

The following lemma entails that the satisfiability of a tableau cascade is unaffected by synchronization.

**Lemma 4.10.** Let $C$ be a tableau cascade such that $\langle \xrightarrow{!\phi} tt' \rangle \in C$ and such that $C(t?)$ is satisfied by a model $\mathcal{M}$ via a function $f$ and $C(t'?)$ is satisfied by $\mathcal{M}|_{!\phi}$ via $f$. If $C$ can be synchronized such that at most $t$ and $t'$ are affected then there's an outcome $C'$ of synchronizing $C$ such that $\mathcal{M}$ satisfies $C'(t?)$ via $f$ and $\mathcal{M}|_{!\phi}$ satisfies $C'(t'?)$ via $f$.

**Proof.** We need to demonstrate that there are finite tableaux $\mathcal{T} \supseteq C(t?)$ and $\mathcal{T}' \supseteq C(t'?)$ (i) which are satisfied via $\mathcal{M}$ and $\mathcal{M}|_{!\phi}$ via $f$ and (ii) such that $\mathcal{T} \xrightarrow{!\phi} \mathcal{T}'$. It then immediately follows that there are minimal tableaux that are supersets of $C(t?)$ and $C(t'?)$ and meet conditions (i) and (ii).

**Figure 4.6.** Backward propagation of nodes, edges, and atoms. By synchronizing the tableau cascade, all nodes, edges, and atoms from the right hand tableau are copied to the left. Moreover, the label $\neg p$ is added to the tableau on the left for all nodes that also exist in the tableau on the right.



First, let $N$ be the known domain of $f$. That is, $N \coloneqq C(t?)[?] \cup C(t'?)[?]$. Informally, $N$ is the totality of nodes of the tableaux named $t$ and $t'$.

Next, let $\mathcal{T}$ be the following tableau:

- $\mathcal{T}[?] = N$
- $\mathcal{T}[???] = \{\langle ann' \rangle \in \text{Ind} \times N \times N \mid \langle af(n)f(n') \in \mathcal{M} \rangle\}$
- $\mathcal{T}[??] = C(t?)[??] \cup \{\langle np \rangle \in N \times \text{Prop} \mid \mathcal{M}f(n)p\} \cup \{\langle n\phi \rangle \mid n \in N \text{ and } \langle \mathcal{M}f(n) \rangle \Vdash \phi\}$

$\mathcal{T}$ is a superset of $C(t?)$. For consider that (i) $N$ is a superset of $C(t?)[?]$, (ii) for every edge $\langle ann' \rangle$ of $C(t?)$ there is an edge $\langle af(n)f(n') \rangle$ in $\mathcal{M}$ by definition 3.13, and (iii) the formulas of $C(t?)$ are copied verbatim.

Let $\mathcal{T}'$ be the following tableau:

- $\mathcal{T}'[?] = \mathcal{T}[?\phi]$
- $\mathcal{T}'[???] = \{\langle ann' \rangle \in \text{Ind} \times \mathcal{T}'[?] \times \mathcal{T}'[?] \mid \langle af(n)f(n') \rangle \in \mathcal{M}\}$
- $\mathcal{T}'[??] = C(t'?)[??] \cup \{\langle np \rangle \in \mathcal{T}'[?] \times \text{Prop} \mid \mathcal{M}f(n)p\}$

Again, $\mathcal{T}'$ is a superset of $C(t'?)$. For assume, to the contrary, that there was a node $n$ of $C(t'?)$ that was not in $\mathcal{T}'$. This would mean that not $\mathcal{T}n\phi$. But by construction this implies that not $\langle \mathcal{M}f(n) \rangle \Vdash \phi$. However, such a node cannot be in $C(t'?)$ since $C(t'?)$ is satisfied by $\mathcal{M}|_{!\phi}$ via $f$.

That $\mathcal{T} \xrightarrow{!\phi} \mathcal{T}'$ holds is also clearly the case by construction.

**Theorem 4.11.** If there is a pointed $\sigma$-model $\langle \mathcal{M} w\rangle$ such that $\langle \mathcal{M} w\rangle \Vdash \phi$ (where $\phi \in \mathcal{L}^{\Box !}$) then an open saturated $\sigma$-tableau $\mathcal{T}$ for $\langle w\phi\rangle$ can be found in a finite number of steps.

**Proof.** Start with the tableau cascade $C_0 = \{\langle 0\rangle, \langle 0\{\langle w\rangle, \langle w\phi\rangle\}\rangle\}$. Thus $C_0$ contains a node 0 and the label for 0 is a tableau that consists of a node $w$ with a label $\phi$.

Next, consider all series $C_1, \dots, C_i, C_{i+1}, \dots$ such that for every natural number $i$, $C_{i+1}$ is the result of applying a rule from Rules to any tableau in $C_i$ or a rule from Rules$_\sigma$ to $C_i(0?)$ and such that

1. The mandatory rules that do not add new edges (or nodes) are applied first.

2. Fairly select zero or more of the mandatory rules that were skipped in the first step and apply them if the following condition is met: Do not add outgoing edges to a node $n$ if and only if there is a node $n'$ such that in every tableau of $C_i$, $n$ either does not exist or $n$ is a copy of $n'$.

   Informally speaking, fair selection means that given due time every rule is chosen (repeatedly).

3. !-prime the tableau cascade.

4. Synchronize the tableau cascade until it is fully synchronized.

Repeat this process until there are no more rules to apply. At that point, fold all virtual $a$-loops.

It's straightforward to see how the above steps will indeed result in saturated tableaux and how on some of the resulting branches all tableaux are free of literal contradictions.

Moreover, the above steps always lead to saturation in a finite number of steps. The argument is the same as in theorem 3.12.

Finally, let $C$ be a tableau cascade derived from $C_0$ as detailed above such that all tableaux in $C$ are $\sigma$-saturated and free of literal contradictions. Because, moreover, $C$ is fully synchronized it is clear that all tableaux in $C$ are open.

## 4.4 Related work

Public announcement logic was originally devised in [30] and, later but independently, in [20].

Two similar tableau systems for public announcement are described in [2, 12]. However, these tableau systems are unintuitive and difficult to use because of the non-trivial relation of the tableau rules to the semantics of PAL.

A different approach is taken in [22]. In this system dynamic formulas with public announcements are rewritten to equivalent non-dynamic formulas using 'reduction rules'. Such rules are also used in axiomatic proof systems for PAL. Indeed, axiomatic systems for propositional logic can be extended to PAL by adding the following axiom schemas:

$$\begin{aligned}
[!\phi]p &\iff (\phi \to p)\\
[!\phi]\neg\psi &\iff (\phi \to \neg[!\phi]\psi)\\
[!\phi](\psi \wedge \chi) &\iff ([!\phi]\psi \wedge [!\phi]\chi)\\
[!\phi]\Box_a\psi &\iff (\phi \to \Box_a(\phi \to [!\phi]\psi))\\
[!\phi][!\psi]\chi &\iff [!(\phi \wedge [!\phi]\psi)]\chi
\end{aligned}$$

where $p \in \text{Prop}$ and $\{\phi, \psi, \chi\} \subseteq \mathcal{L}^{\Box !}$ [38]. The relation of the reduction rules to the semantics of PAL might not be immediately obvious, however. Therefore the proof system in this chapter is arguably easier to understand and use.

# 5 Dynamic Epistemic Logic with Action Models

In the previous chapter we discussed public announcement logic, which is a dynamic modal logic for reasoning about agents who discover facts in a public setting. In this chapter we will look at an extension of public announcement logic that allows us to reason about many more kinds of epistemic events. Using dynamic epistemic logic with action models (DEL) we can reason not only about public discovery of facts, but also about various forms of private and semi-private learning.

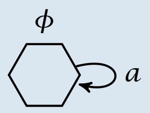
DEL can be used to reason about card games and other knowledge games. Some of these games are relevant for cryptography and security protocols. The link between knowledge games and dynamic epistemic logic is treated extensively in [36, 37].

## 5.1 Syntax and semantics of $\mathcal{L}^{\Box\otimes}$

The language of dynamic epistemic logic $\mathcal{L}^{\Box\otimes}$ adds formulas of the form $[\otimes\pi]\phi$ to the language of modal logic. Here $\pi$ represents an epistemic event or action and $\phi$ is a formula of $\mathcal{L}^{\Box\otimes}$. Epistemic events, in turn, are described by lgraphs that have $\mathcal{L}^{\Box\otimes}$ sentences for labels. These lgraphs are called action models.

The definitions for $\mathcal{L}^{\Box\otimes}$ and 'action models' are mutually recursive.

**Figure 5.1.** A basic action model $\mathcal{A}$ with one node. This action model represent the update where non-$\phi$ worlds are deleted; $a$-edges are kept as-is unless they originate from or point to a non-$\phi$ world. The update operation $|_{\otimes\mathcal{A}}$ is equivalent to the update operation $|_{!\phi}$.



**Definition 5.1.** The set «$\mathcal{L}^{\Box\otimes}$» is determined by the following grammar:

$$\phi ::= p \mid \neg\phi \mid (\phi \wedge \phi) \mid \Box_a\phi \mid [\otimes\mathcal{A}e]\phi.$$

As usual, $p$ stands for any element of the set of atomic proposition Prop and $a$ is any member of the set of agents Ind. Finally, let $\langle\mathcal{A}e\rangle$ be any pointed action model.

**Definition 5.2.** An «action model» $\mathcal{A}$ is an lgraph such that

- The nodes of $\mathcal{A}$ are called «actions» or «events».
- The edges of $\mathcal{A}$ are edges indexed by elements of Ind.
- Every node $n$ has exactly one label $\phi \in \mathcal{L}^{\Box\otimes}$, called a «precondition». By abuse of terminology, if $\phi = \top$ then we say that $n$ has no preconditions. The notation $\mathcal{A}(n?)$ denotes the precondition of $n$.

The mutual recursion is unproblematic if action models are thought of as syntactic structures. For instance, we could convert them to a string that describes their structure. It is then immediately apparent that action models are longer in 'length' than all of the labels they contain.

Our conversion of action models to syntactic structures makes the issue salient that having semantic objects as part of a language is unusual. We admit that this is unusual but don't think it's a real problem. We think it's simply the case that the most convenient way to describe an epistemic event is to draw a diagram. Indeed, Hans van Ditmarsch argues that we can think of action models as syntactic objects from the get-go [37, §6.1]. Johan van Benthem has a more poetic view: "If you are down-to-earth, you will find a syntax putting models in a language weird, and your life will be full of fears of inconsistency. If you are born to be wild, you will see DEL as a welcome flight of the imagination." [34, p. 86].

In the previous chapter we defined the forcing relation $\Vdash$ by taking the definition for $\Vdash$ from modal logic and adding an extra clause to it for sentences of the form $[!\phi]\psi$. That is, in chapter 3 the behavior of the forcing relation was left undefined for wffs $[!\phi]\psi$ and in chapter 4 we retroactively defined how to interpret such formulas.

We are again faced with a gap in the definition of $\Vdash$. Hence we extend the definition of the forcing relation again.

**Definition 5.3.** The forcing relation «$\Vdash$» is as before, but also meets the following constraint.

$$\langle \mathcal{M} w\rangle \Vdash [\otimes \mathcal{A} e]\phi \iff \text{if } \langle \mathcal{M} w\rangle \Vdash \mathcal{A}(e?) \text{ then } \langle \mathcal{M}|_{\otimes \mathcal{A}}\langle we\rangle\rangle \Vdash \phi$$

where

$$\mathcal{M}|_{\otimes \mathcal{A}} = \{\langle n\rangle \mid n \in N\} \cup E \cup L$$

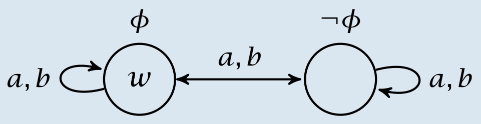
such that

- $N \coloneqq \{\langle w'e'\rangle \in \mathcal{M}[?] \times \mathcal{A}[?] \mid \langle \mathcal{M} w'\rangle \Vdash \mathcal{A}(e'?)\}$
- $E \coloneqq \{\langle a\langle w'e'\rangle\langle w''e''\rangle\rangle \in \text{Ind} \times N \times N \mid \mathcal{M} a w' w'' \text{ and } \mathcal{A} a e' e''\}$
- $L \coloneqq \{\langle\langle w'e'\rangle p\rangle \in N \times \text{Prop} \mid \mathcal{M} w' p\}$

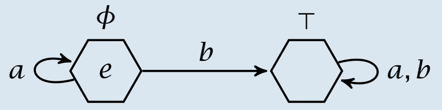
Thus the set of nodes of $\mathcal{M}|_{\otimes \mathcal{A}}$ is a subset of the Cartesian product of the set of nodes of $\mathcal{M}$ and $\mathcal{A}$. For any two nodes $\langle w'e'\rangle$ and $\langle w''e''\rangle$, there is an $a$-edge in $\mathcal{M}|_{\otimes \mathcal{A}}$ if and only if there is an $a$-edge between $w'$ and $w''$, and between $e'$ and $e''$. Finally, the atoms for any node $\langle w'e'\rangle$ in $\mathcal{M}|_{\otimes \mathcal{A}}$ are copied from $w'$ in $\mathcal{M}$; non-atomic labels are *not* transferred.

The above semantics also suggest an interpretation of action models as semantic objects. They can be thought of as relativizations of Kripke models (cf. [3, §1] and [37, §6.1]).

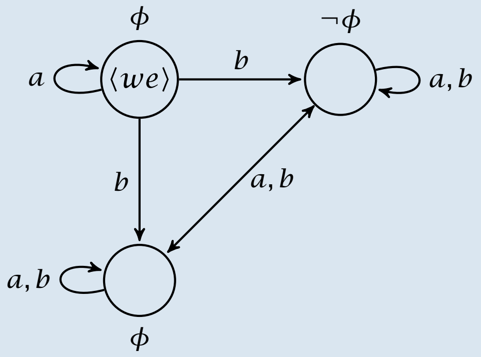
**Figure 5.2.** Below a Kripke model $\mathcal{M}$, an action model $\mathcal{A}$, and the updated Kripke model $\mathcal{M}|_{\otimes\mathcal{A}}$ are depicted. In the original model $\mathcal{M}$ the agents $a$ and $b$ are ignorant about the truth value of the formula $\phi$. That is, in both worlds it is the case that $\neg\Box_a\phi$, $\neg\Box_a\neg\phi$, $\neg\Box_b\phi$, and $\neg\Box_b\neg\phi$. The update model $\mathcal{A}$ represents an update where (i) all $\phi$-worlds are copied once and (ii) all worlds are copied indescriminately. Thus the $\phi$-worlds are copied twice. $\mathcal{A}$ also specifies that all $b$-links to the worlds of (i) are cut. The last figure shows the result of updating $\mathcal{M}$ with the action model $\mathcal{A}$. In $\langle we\rangle$ agent $a$ has come to learn that $\phi$ is true but $b$ doesn't know that $\phi$. Moreover, $b$ doesn't know that $a$ knows whether $\phi$ is true ($\neg\Box_b(\Box_a\phi \vee \Box_a\neg\phi)$) and $a$ knows that $b$ doesn't know this ($\Box_a\neg\Box_b(\Box_a\phi \vee \Box_a\neg\phi)$).



It is not know if $\phi$ is true or not.



$a$ learns that $\phi$ is true.



$a$ now knows that $\phi$ holds.

## 5.2 Dynamic Tableaux

We now extend our tableau system to cover all formulas of $\mathcal{L}^{\Box\otimes}$. We accomplish this through three amendments that are entirely analogous to how we implemented the tableau system for PAL.

First, we extend the tableau rules.

**Table 5.1.** For every $[\otimes\mathcal{A}e]\phi$ and $\neg[\otimes\mathcal{A}e]\phi$ in the tableau, the rule $\mathbf{R}_\otimes$ forces every node to decide, for all preconditions of $\mathcal{A}$, if the precondition or its negation should be true.

| Name | Precondition | Postcondition |
|---|---|---|
| $\mathbf{R}_\otimes$ | $\mathcal{T}n[\otimes\mathcal{A}e]\phi, \mathcal{T}n'$, and $\mathcal{A}e'$ | $\mathcal{T}n'\mathcal{A}(e'?)$ or $\mathcal{T}n'\neg\mathcal{A}(e'?)$ |
| | $\mathcal{T}n\neg[\otimes\mathcal{A}e]\phi, \mathcal{T}n'$, and $\mathcal{A}e'$ | $\mathcal{T}n'\mathcal{A}(e'?)$ or $\mathcal{T}n'\neg\mathcal{A}(e'?)$ |

**Definition 5.4.** Let the mandatory rules of Rules (and Rules$_\sigma$) include the additional rule in table 5.1.

Second, we translate the operation $|_{\otimes\mathcal{A}}$ to a relation between tableaux.

**Definition 5.5.** For any two tableaux $\mathcal{T}$ and $\mathcal{T}'$ the relationship «$\mathcal{T} \xrightarrow{\otimes\mathcal{A}} \mathcal{T}'$» holds if and only if

1. $\mathcal{T}'[?] = \{\langle ne\rangle \in \mathcal{T}[?] \times \mathcal{A}[?] \mid \mathcal{T}n\mathcal{A}(e?)\}$.
2. $\mathcal{T}'[???] = \{\langle a\langle ne\rangle\langle n'e'\rangle\rangle \in \mathrm{Ind} \times \mathcal{T}'[?] \times \mathcal{T}'[?] \mid \mathcal{T}ann' \text{ and } \mathcal{A}aee'\}$.
3. $\mathcal{T}'[??] \cap (\mathcal{T}'[?] \times \mathrm{Prop}) = \{\langle\langle ne\rangle p\rangle \in \mathcal{T}'[?] \times \mathrm{Prop} \mid \mathcal{T}np\}$.

Third and last, we redefine the notion of 'open tableau'.

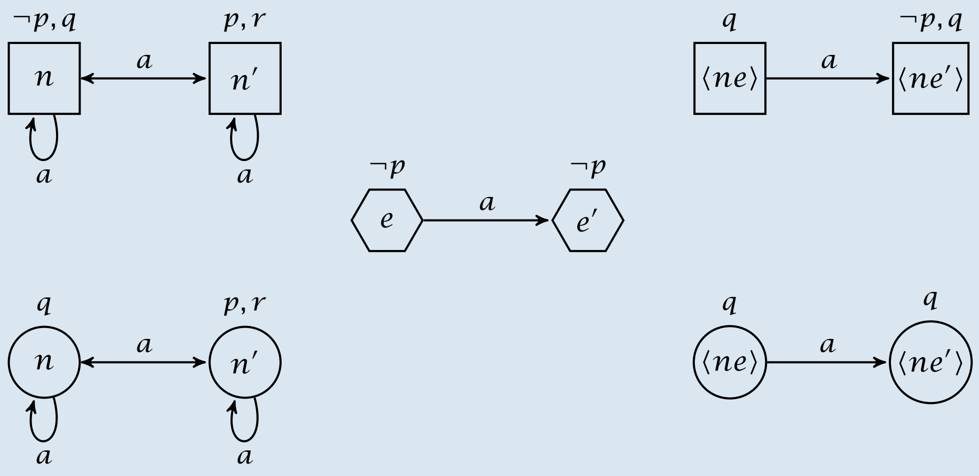
**Definition 5.6.** Let the predicate «open» be as before, except for two added conditions:

1. If $\mathcal{T}n[\otimes\mathcal{A}e]\phi$ and $\mathcal{T}n\mathcal{A}(e?)$ then there are no open saturated tableaux $\mathcal{T}'$ for $\langle\langle ne\rangle\neg\phi\rangle$ such that $\mathcal{T} \xrightarrow{\otimes\mathcal{A}} \mathcal{T}'$.
2. If $\mathcal{T}n\neg[\otimes\mathcal{A}e]\phi$ then $\mathcal{T}n\mathcal{A}(e?)$ and there is an open saturated tableau $\mathcal{T}'$ for $\langle\langle ne\rangle\neg\phi\rangle$ such that $\mathcal{T} \xrightarrow{\otimes\mathcal{A}} \mathcal{T}'$.

As in the previous chapter we define a declarativeness notion, and demonstrate a form of symmetry between $|_{\otimes\mathcal{A}}$ and $\xrightarrow{\otimes\mathcal{A}}$ for '$\mathcal{A}$-declarative' tableaux and the models that natively satisfy them.

**Definition 5.7.** A tableau $\mathcal{T}$ is $\mathcal{A}$-declarative if and only for every $n \in \mathcal{T}[?]$ and every $e \in \mathcal{A}[?]$ it is the case that $\mathcal{T}n\mathcal{A}(e?)$ or $\mathcal{T}n\neg\mathcal{A}(e?)$.

**Figure 5.3.** In this diagram two tableaux $\mathcal{T}$ (top left) and $\mathcal{M}'$ (top right), two Kripke models $\mathcal{M}$ (bottom left) and $\mathcal{M}'$ (bottom right), and an action model $\mathcal{A}$ (center) are shown. $\mathcal{M}$ and $\mathcal{M}'$ are stock models of $\mathcal{T}$ and $\mathcal{T}'$. Moreover, $\mathcal{T} \xrightarrow{\otimes\mathcal{A}} \mathcal{T}'$ and $\mathcal{M}' = \mathcal{M}|_{\otimes\mathcal{A}}$.



**Lemma 5.1.** Let $\mathcal{A}$ be an action model and let $\mathcal{M}$ be the stock model of an $\mathcal{A}$-declarative tableau $\mathcal{T}$ such that $\mathcal{M}$ natively satisfies $\mathcal{T}$ up to the negation of all formulas in $\mathcal{A}$. It is then the case that $\mathcal{M}|_{\otimes\mathcal{A}}$ is the stock model for $\mathcal{T}'$ if and only if $\mathcal{T} \xrightarrow{\otimes\mathcal{A}} \mathcal{T}'$.

**Proof.** First of all, observe that (i) $\mathcal{M}$ natively satisfies $\mathcal{T}$ (up to the negation of all preconditions in $\mathcal{A}$) and (ii) $\mathcal{T}$ is $\mathcal{A}$-declarative. Therefore it is the case that for all $n \in \mathcal{T}[?]$ and $e \in \mathcal{A}[?]$,

$$\mathcal{T} n \mathcal{A}(e?) \iff \langle \mathcal{M} n \rangle \Vdash \mathcal{A}(e?). \tag{5.1}$$

Proof for the left-to-right part of the lemma. Assume that $\mathcal{M}|_{\otimes\mathcal{A}}$ is the stock model of $\mathcal{T}'$. Thus,

$$\mathcal{T}'[?] = \{\langle we \rangle \in \mathcal{M}[?] \times \mathcal{A}[?] \mid \langle \mathcal{M} w \rangle \Vdash \mathcal{A}(e?)\}.$$

By equation (5.1) it follows that

$$\mathcal{T}'[?] = \{\langle ne \rangle \in \mathcal{T}[?] \times \mathcal{A}[?] \mid \mathcal{T} n \mathcal{A}(e?)\}.$$

Because $\mathcal{M}|_{\otimes\mathcal{A}}$ is the stock model of $\mathcal{T}'$ it also follows that

$$\mathcal{T}'[???] = \{\langle a\langle ne\rangle\langle n'e'\rangle \in \text{Ind} \times \mathcal{T}'[?] \times \mathcal{T}'[?]\rangle \mid \mathcal{T}ann' \text{ and } \mathcal{A}aee'\}$$

and that for all nodes $\langle ne\rangle \in \mathcal{T}'[?]$ and atoms $p$ it is the case that

$$\mathcal{T}np \iff \mathcal{T}'\langle ne\rangle p.$$

Finally, it follows that $\mathcal{T} \xrightarrow{\otimes\mathcal{A}} \mathcal{T}'$.

Proof from right to left. Assume that

$$\mathcal{T} \xrightarrow{\otimes\mathcal{A}} \mathcal{T}'.$$

It follows that

- $\mathcal{T}'[?] = \{\langle ne\rangle \in \mathcal{T}[?] \times \mathcal{A}[?] \mid \mathcal{T}n\mathcal{A}(e?)\}$
- $\mathcal{T}'[???] = \{\langle a\langle ne\rangle\langle n'e'\rangle \in \text{Ind} \times \mathcal{T}'[?] \times \mathcal{T}'[?]\rangle \mid \mathcal{T}ann' \text{ and } \mathcal{A}aee'\}$
- $\mathcal{T}'[??] \cap (\mathcal{T}'[?] \times \text{Prop}) = \{\langle\langle ne\rangle p\rangle \in \mathcal{T}'[?] \times \text{Prop} \mid \mathcal{T}np\}$.

By the results from the first paragraph and the fact that $\mathcal{M}$ is the stock model of $\mathcal{T}$ it is also the case that

$$\mathcal{T}'[?] = \{\langle we\rangle \in \mathcal{M}[?] \times \mathcal{A}[?] \mid \langle \mathcal{M}w\rangle \Vdash \mathcal{A}(e?)\},$$

that $\mathcal{T}'$ has the same frame as $\mathcal{M}|_{\otimes\mathcal{A}}$, and that

$$\mathcal{T}'np \iff \mathcal{M}|_{\otimes\mathcal{A}}np$$

for every atom $p$. Thus $\mathcal{M}|_{\otimes\mathcal{A}}$ is the stock model of $\mathcal{T}'$.

We extend lemma 3.7 once more and prove that $\mathbf{R}_{\otimes}$ preserves satisfiability.

**Lemma 5.2.** If a $\sigma$-model $\mathcal{M}$ satisfies a tableau $\mathcal{T}$ via a function $f$ then it is the case that, if $\mathcal{T}$ is not $\mathbf{R}_{\otimes}$-saturated then there is a tableau $\mathcal{T}' \in \mathbf{R}_{\otimes}[\mathcal{T}] - \{\mathcal{T}\}$ and a function $f'$ such that $\mathcal{M}$ satisfies $\mathcal{T}'$ via $f'$.

**Proof.** Suppose $\mathbf{R}_{\otimes}$ is triggered by $\mathcal{T}n[\otimes\mathcal{A}e]\phi$ (or $\mathcal{T}n\neg[\otimes\mathcal{A}e]\phi$), $\mathcal{A}e'$, and $\mathcal{T}n'$. If $\langle \mathcal{M}f(n')\rangle \Vdash \mathcal{A}(e'?)$ then let $\mathcal{T}' \coloneqq \mathcal{T} \cup \langle n'\mathcal{A}(e'?)\rangle$; otherwise let $\mathcal{T}' \coloneqq \mathcal{T} \cup \langle n'\neg\mathcal{A}(e'?)\rangle$. As $\mathcal{M}$ satisfies $\mathcal{T}'$ via $f$, let $f' \coloneqq f$.

It is now time to prove soundness and completeness. As before we use mutual induction.

**Lemma 5.3 (Lemma to Soundness and Completeness).** Given a stock model $\mathcal{M}$ for a saturated tableau $\mathcal{T}$,

1. If $\mathcal{M}$ natively satisfies $\mathcal{T}$ and $\mathcal{T}$ is a tableau for $\phi$ then $\mathcal{T}$ is open.
2. If $\mathcal{T}$ is open and $\mathcal{T}n\phi$ then $\langle \mathcal{M}n\rangle \Vdash \phi$.

**Proof.** Proof by mutual induction on the length of $\phi$. Assume that the lemma holds for all $\psi$ shorter than $\phi$.

**Part 1.**

- In lemmas 3.7 and 5.2 it is shown that $\mathcal{T}$ is free of literal contradictions.
- For all $\langle n[\otimes\mathcal{A}e]\psi\rangle \in \mathcal{T}$ it must be demonstrated that if $\mathcal{T}n\mathcal{A}(e?)$ then there is no open saturated tableau $\mathcal{T}'$ for $\langle\langle ne\rangle\neg\psi\rangle$ such that $\mathcal{T} \xrightarrow{\otimes\mathcal{A}} \mathcal{T}'$.

  Proof by contradiction. Suppose such a tableau $\mathcal{T}'$ did exist and that $\mathcal{T}n\mathcal{A}(e?)$. By Part 2 of the IH it is then the case that, with $\mathcal{M}'$ the stock model of $\mathcal{T}'$, $\langle \mathcal{M}'\langle ne\rangle\rangle \Vdash \neg\psi$. By lemma 5.1 it also follows that $\mathcal{M}' = \mathcal{M}|_{\otimes\mathcal{A}}$.

  Moreover, since $\mathcal{M}$ natively satisfies $\mathcal{T}$ it is the case that $\langle \mathcal{M}n\rangle \Vdash \mathcal{A}(e?)$. These results, however, contradict the assumption that $\langle \mathcal{M}n\rangle \Vdash [\otimes\mathcal{A}e]\psi$. This ends the proof by contradiction.

- For all $\langle n\neg[\otimes\mathcal{A}e]\psi\rangle \in \mathcal{T}$ it must be demonstrated that $\mathcal{T}n\mathcal{A}(e?)$ and that there is an open saturated tableau $\mathcal{T}'$ for $\langle\langle ne\rangle\neg\psi\rangle$ such that $\mathcal{T} \xrightarrow{\otimes\mathcal{A}} \mathcal{T}'$.

  First of all, by $\mathbf{R}_\otimes$ it is the case that $\mathcal{T}n\mathcal{A}(e?)$ or $\mathcal{T}n\neg\mathcal{A}(e?)$. Because $\mathcal{M}$ natively satisfies $\mathcal{T}$ and because $\mathcal{T}n\neg[\otimes\mathcal{A}e]\psi$, however, it follows that $\mathcal{T}n\mathcal{A}(e?)$ and not $\mathcal{T}n\neg\mathcal{A}(e?)$.

Since $\mathcal{M}$ natively satisfies $\mathcal{T}$ it is also the case that $\langle \mathcal{M}|_{\otimes\mathcal{A}}\langle ne\rangle\rangle \Vdash \neg\psi$. By Part 1 of the IH it now follows that the stock tableau $\mathcal{T}'$ for $\langle \mathcal{M}|_{\otimes\mathcal{A}}\langle ne\rangle\neg\psi\rangle$ is open. Also notice that $\mathcal{T}$ is $\mathcal{A}$-declarative because of $\mathbf{R}_\otimes$. By lemma 5.1 it follows that $\mathcal{T} \xrightarrow{\otimes\psi} \mathcal{T}'$ and this concludes our proof.

**Part 2.** The bulk of this proof has already been covered in lemma 3.9. We only provide the proofs for the patterns specific to $\mathcal{L}^{\Box\otimes}$.

- Suppose $\phi$ is $[\otimes\mathcal{A}e]\psi$. It has to be demonstrated that if $\langle\mathcal{M}n\rangle \Vdash \mathcal{A}(e?)$ then $\langle \mathcal{M}|_{\otimes\mathcal{A}}\langle ne\rangle\rangle \Vdash \psi$.

  The proof proceeds by contradiction. Assume that $\langle\mathcal{M}n\rangle \Vdash \mathcal{A}(e?)$ and $\langle \mathcal{M}|_{\otimes\mathcal{A}}\langle ne\rangle\rangle \Vdash \neg\psi$.

  By Part 1 of the induction hypothesis it follows that every stock tableau $\mathcal{T}'$ for $\langle \mathcal{M}|_{\otimes\mathcal{A}}\langle ne\rangle\neg\psi\rangle$ is open. Moreover, $\mathcal{T}$ is $\mathcal{A}$-declarative by $\mathbf{R}_\otimes$. By Part 2 of the IH it follows that $\mathcal{M}$ natively satisfies $\mathcal{T}$ up to the negation of all formulas in $\mathcal{A}$. By lemma 5.1 it now follows that $\mathcal{T} \xrightarrow{\otimes\mathcal{A}} \mathcal{T}'$.

  However, since $\mathcal{T}$ is an open tableau we know that there is no saturated open tableau $\mathcal{T}''$ for $\langle n\neg\psi\rangle$ such that $\mathcal{T} \xrightarrow{\otimes\mathcal{A}} \mathcal{T}''$. This concludes the proof by contradiction.

- Suppose $\phi$ is $\neg[\otimes\mathcal{A}e]\psi$. We need to demonstrate that $\langle\mathcal{M}n\rangle \Vdash \mathcal{A}(e?)$ and that $\langle \mathcal{M}|_{\otimes\mathcal{A}}\langle ne\rangle\rangle \Vdash \neg\psi$.

  First, since $\mathcal{T}$ is open it follows that $\mathcal{T}n\mathcal{A}(e?)$ and that there is an open saturated tableau $\mathcal{T}'$ for $\langle n\neg\psi\rangle$ such that $\mathcal{T} \xrightarrow{\otimes\mathcal{A}} \mathcal{T}'$.

  Second, $\mathcal{T}$ is $\mathcal{A}$-declarative by $\mathbf{R}_\otimes$. Hence, by Part 2 of the IH it also follows that $\mathcal{M}$ natively satisfies $\mathcal{T}$ up to the negation of all formulas in $\mathcal{A}$. By lemma 5.1 it is now the case that $\mathcal{M}|_{\otimes\mathcal{A}}$ is the stock model of $\mathcal{T}'$.

  Finally, again by Part 2 of the IH, it is the case that $\langle\mathcal{M}n\rangle \Vdash \mathcal{A}(e?)$ and $\langle \mathcal{M}|_{\otimes\mathcal{A}}\langle ne\rangle\rangle \Vdash \neg\psi$.

## 5.3 Decidability

In this section we demonstrate that tableaux for $\mathcal{L}^{\Box\otimes}$ are decidable. The proof extends the techniques used in the previous chapter.

First we need to tweak the definition of satisfiability with respect to tableau cascades.

**Definition 5.8.** A pointed tableau cascade $\langle Ct \rangle$ is «satisfied» by a model $\mathcal{M}$ via a function $f$ if and only if

- $\mathcal{M}$ satisfies the tableau $C(t?)$ via $f$.
- For every $\langle \xrightarrow{\otimes\mathcal{A}} t' \rangle \in C[?t?]$ it is the case that $\mathcal{M}|_{\otimes\phi}$ satisfies $\langle Ct' \rangle$ via the function $f' : \langle xy \rangle \mapsto \langle f(x)y \rangle$.

We say that a tableau cascade $C$ is satisfied by a model $\mathcal{M}$ via a function $f$ if and only if $\langle C\,\mathrm{root}(C) \rangle$ is satisfied by $\mathcal{M}$ via $f$.

Next, we expand the notion of priming to formulas of the form $[\otimes\mathcal{A}]\phi$ and $\neg[\otimes\mathcal{A}]\phi$.

**Definition 5.9.** $C'$ is the result of «$\otimes$-priming» a tableau cascade $C$ if and only if $C'$ is a minimal extension of $C$ such that for every $\langle t\mathcal{T} \rangle \in C[??]$,

- If $\mathcal{T}n[\otimes\mathcal{A}e]\phi$ and $\mathcal{T}n\mathcal{A}(e?)$ then there is a tableau $\mathcal{T}'$ for $\langle\langle ne \rangle\phi\rangle$ such that in $C$ there is a $\xrightarrow{\otimes\mathcal{A}}$-link from $t$ to $C(?\mathcal{T}')$.
- If $\mathcal{T}n\neg[\otimes\mathcal{A}e]\phi$ then there is a tableau $\mathcal{T}'$ for $\langle\langle ne \rangle\neg\phi\rangle$ such that in $C$ there is a $\xrightarrow{\otimes\mathcal{A}}$-link from $t$ to $C(?\mathcal{T}')$.

As before, priming does not affect the satisfiability of a tableau cascade.

**Lemma 5.4.** Given a tableau cascade $C$ that is satisfied by a model $\mathcal{M}^\star$ via a function $f^\star$, it is the case that any $C'$ that is the result of $\otimes$-priming $C$ is satisfied by $\mathcal{M}^\star$ via $f^\star$.

**Proof.** Let $t'$ be a node that is new in $C'$ and that is accessible from a node $t$ over $\xrightarrow{\otimes\mathcal{A}}$. Notice that $t$ is a node of both $C$ and $C'$ by construction. Define $\mathcal{T} \coloneqq C(t?)$ and $\mathcal{T}' \coloneqq C'(t'?)$.

Let $\mathcal{M}$ and $f$ be any model and function such that $\langle Ct\rangle$ is satisfied by $\mathcal{M}$ via $f$. We demonstrate that $\langle C't'\rangle$ is satisfied by $\mathcal{M}|_{\otimes\mathcal{A}}$ via $f' : \langle xy\rangle \to \langle f(x)y\rangle$.

The case where the precondition $\mathcal{A}(e?)$ is not in $n$ is trivial. Suppose $t'$ was added because $\mathcal{T}n[\otimes\mathcal{A}e]\phi$ and $\mathcal{T}n\mathcal{A}(e?)$. By definition of priming it is then the case that $\mathcal{T}' = \{\langle ne\rangle, \langle\langle ne\rangle\phi\rangle\}$. Because $C$ is satisfied by $\mathcal{M}$ via $f$ it is the case that $\langle\mathcal{M}f(n)\rangle \Vdash [\otimes\mathcal{A}e]\phi$ and $\langle\mathcal{M}f(n)\rangle \Vdash \mathcal{A}(e?)$. This implies that the model $\mathcal{M}|_{\otimes\mathcal{A}}$ is nonempty and contains a world $\langle f(n)e\rangle$. Moreover, $\langle\mathcal{M}|_{\otimes\mathcal{A}}\langle f(n)e\rangle\rangle \Vdash \phi$. We can now conclude that $\mathcal{M}|_{\otimes\mathcal{A}}$ satisfies $\mathcal{T}'$ via $f'$.

Suppose $t'$ was added because $\mathcal{T}n\neg[\otimes\mathcal{A}e]\phi$. From the definition of priming it follows that $\mathcal{T}' = \{\langle ne\rangle, \langle\langle ne\rangle\neg\phi\rangle\}$. Since $C$ is satisfied by $\mathcal{M}$ via $f$ it also follows that $\langle\mathcal{M}f(n)\rangle \Vdash \neg[\otimes\mathcal{A}e]\phi$ and that $\langle\mathcal{M}|_{\otimes\mathcal{A}}\langle f(n)e\rangle\rangle \Vdash \neg\psi$. Consequently, $\mathcal{M}|_{\otimes\mathcal{A}}$ satisfies $\mathcal{T}'$ via $f'$.

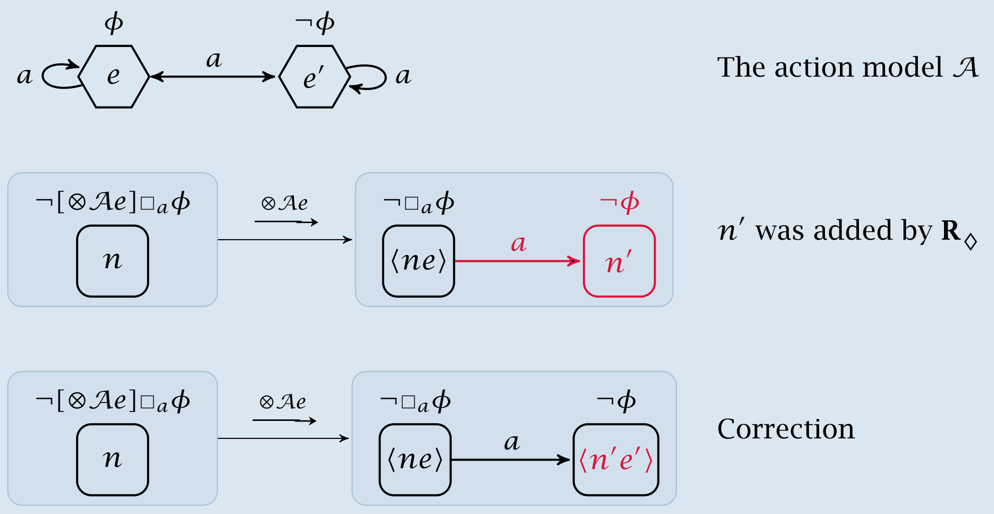
There's a small problem. The nodes created by $\mathbf{R}_\Diamond$ or $\mathbf{R}_\mathbf{D}^\mathbf{a}$ are elements of $\mathbb{N}$. However, the satisfaction definition we use in this chapter stipulates that in a tableau cascade $C$, for any node $t \neq \mathrm{root}(C)$ it is the case that the tableau nodes of $C(t?)$ are *not* elements of $\mathbb{N}$. Rather, with $\xrightarrow{\otimes\mathcal{A}_1}, \dots, \xrightarrow{\otimes\mathcal{A}_n}$ the edges from $\mathrm{root}(C)$ to $t$, the nodes of $C(t?)$ are elements of $\mathbb{N} \times \mathcal{A}_1 \times \cdots \times \mathcal{A}_n$. We introduce a simple operation to remedy this problem.

**Definition 5.10.** *$C'$ is the result of «$\otimes$-correcting» a tableau cascade $C$ if and only if*

- *$C$ and $C'$ have the same frame.*
- *For every tableau cascade node $t \in C[?]$ there is an embedding $h$ from $\mathcal{T} \coloneqq C(t?)$ to $\mathcal{T}' \coloneqq C'(t?)$ such that for all $n \in \mathcal{T}[?]$,*
  - $n \in \mathbb{N} \implies h(n) \in \{n\} \times \mathcal{A}_1[?] \times \cdots \times \mathcal{A}_n[?]$
  - $h(n) = n$ *otherwise*

  *where $\xrightarrow{\otimes\mathcal{A}_1}, \dots, \xrightarrow{\otimes\mathcal{A}_n}$ are the $\xrightarrow{\otimes}$-edges on the path from $\mathrm{root}(C)$ to $t$.*

**Figure 5.4.** Below a tableau cascade is shown right after the rule $\mathbf{R}_\Diamond$ was applied and after a correcting operation is performed. The correcting operation renames the node $n'$ to $\langle n'e' \rangle$, which fits the tableau cascade's configuration.



We prefer to introduce the correcting operation over changing the tableau rules because this problem only surfaces after the soundness and completeness proofs for DEL. We feel this makes for insufficient justification for complicating the tableau rules. This is all the more true considering that we use the same tableau rules for the other logics in this dissertation, which are not affected by this issue.

Notice that any tableau cascade can be corrected using only $\mathbf{R}_\star$. By correcting tableau cascades after a tableau rule is applied, satisfiability—as defined in definition 5.8—can be restored.

**Lemma 5.5.** Given a tableau cascade $C$ that is satisfied by a model $\mathcal{M}$ via a function $f$, if $R$ is a tableau rule and $C(t?)$ is not $R$-saturated then there is a corrected outcome $C' \neq C$ of applying $R$ to $C$ that is satisfied by $\mathcal{M}$ via some function $f'$.

**Proof.** By lemmas 3.7 and 5.2 it is the case that there is a tableau $\mathcal{T}^\star$ that is the result of applying $R$ to $C(t?)$ that is satisfied by some function $f^\star$. If $\mathcal{T}^\star$ contains no new nodes then $C'(t?) = \mathcal{T}^\star$ and $f' = f^\star$. If a new node was added then it merely has to be renamed in accordance with definition 5.8.

The notion of synchronization remains unchanged and so does the following property.

**Lemma 5.6.** Let $C$ be a tableau cascade such that $\langle \xrightarrow{\otimes\mathcal{A}} tt' \rangle \in C$ and such that $C(t?)$ is satisfied by a model $\mathcal{M}$ via a function $f$ and $C(t'?)$ is satisfied by $\mathcal{M}|_{\otimes\mathcal{A}}$ via $f' : \langle ne \rangle \mapsto \langle f(n)e \rangle$. If $C$ can be synchronized such that at most $t$ and $t'$ are affected then there's an outcome $C'$ of synchronizing $C$ such that $\mathcal{M}$ satisfies $C'(t?)$ via $f$ and $\mathcal{M}|_{!\phi}$ satisfies $C'(t'?)$ via $f'$.

**Proof.** The principles behind synchronization remain the same as before. Therefore we omit this proof.

We now have all the scaffolding we need to demonstrate that $\mathcal{L}^{\Box\otimes}$-tableaux are decidable.

**Theorem 5.7.** If there is a pointed model $\langle \mathcal{M}w \rangle$ such that $\langle \mathcal{M}w \rangle \Vdash \phi$ (where $\phi \in \mathcal{L}^{\Box\otimes}$) then an open saturated tableau $\mathcal{T}$ for $\phi$ can be found in a finite number of steps.

**Proof.** We skip most of the proof because it is analogous to the proof for $\mathcal{L}^{\Box !}$. The main exception is that $\otimes$-correction must be performed before synchronization. Because action models and tableau cascades are finite there are only a finite number of ways in which such $\otimes$-correction can be performed. Therefore this introduces no new issues.

## 5.4 Related work

The roots of DEL can be found in [4, 20, 30]. The seminal work describing action models is [3].

We know of only one tableau system for dynamic epistemic logic with action models, namely the system described by Hansen in [22]. We already discussed this tableau system in section 4.4 and the same comments apply here.

# 6 Dynamic Preorder Logics

In chapter 3 we saw that modal logic has a variety of applications. Up to this point, however, we only discussed dynamics for epistemic logics. In this chapter we focus on dynamic operators for changing preference and plausibility relations. Both of these are quintessential preorder relations—that is to say, they are reflexive and transitive. Sometimes they are also presumed to be connected relations but we will not make this assumption here.

In this chapter we take operators from two existing logics and combine them in a new language and proof system. The first of these existing logics is a logic for reasoning about preferences and changes in preferences. The second logic interprets plausibility relations as belief strength.

## 6.1 Belief revision and preference change

One particularly important work on belief revision is [1]. In this article the authors arrive at a series of postulates for belief revision. These postulates are generally known as the AGM postulates and are an important reference for assessing the coherence of belief change operations. That's not to say that they are perfect: The AGM postulates have problems with iterated belief revision [23] and only cover single-agent settings and beliefs about ontic facts.

Dynamic doxastic logic is the dynamic modal logic take on [1], while also addressing some of its shortcomings. In this chapter we discuss the lexical upgrade operator from [6]. Edges from $w$ to $w'$ are interpreted in [6] as meaning that $w'$ is at least as plausible as $w$.

[6] contains another operator for belief change that has the moniker 'conservative upgrade'. Unfortunately we failed to model it in our tableau system. The culprit is that it implicitly refers to a belief operator $[\text{best}]_a\phi$ which holds true in a world $w$ if and only if $\phi$ is true in the most plausible worlds connected to $w$ over $a$. It's an open problem whether the operator $[\text{best}]_a$ can be added in our tableau system without extending tableau structures. As extending tableau structures would go against the spirit of this thesis, we opted instead to ignore conservative upgrades in the rest of this chapter.

We also discuss two dynamic operators for preference change—namely the update and upgrade operators from [35]. An edge from $w$ to $w'$ is taken to mean that $w'$ is preferred at least as much as $w$.

## 6.2 $\mathcal{L}^{U\Box ¡\sharp\Uparrow}$ and its dynamic semantics

**Definition 6.1.** Let the dynamic doxastic and preference language $\mathcal{L}^{U\Box ¡\sharp\Uparrow}$ be the set of all formulas that conform to the following specification.

$$\phi ::= p \mid \neg\phi \mid (\phi \wedge \phi) \mid U\phi \mid \Box_a\phi \mid [¡\phi]\phi \mid [\sharp\phi]\phi \mid [\Uparrow\phi]\phi,$$

where $p \in \text{Prop}$ and $a \in \text{Ind}$. Moreover, let $\text{Ind}^P \subseteq \text{Ind}$ be indices for preferences and let $\text{Ind}^B \subseteq \text{Ind}$ be indices for doxastic plausibility relations.

Where $a \in \text{Ind}^P$, read $\Box_a\phi$ as '$a$ prefers $\phi$ to be the case'; where $a \in \text{Ind}^B$, read $\Box_a\phi$ as '$a$ safely believes that $\phi$ is true'.

When modeling a doxastic setting with preferences, every agent should usually be assigned one preference index and one doxastic index.

Our preorder language has two new types of dynamic formulas. Read $[¡\phi]\psi$ as 'after the state is updated with $\phi$, $\psi$ is the case'; $[\sharp\phi]\psi$ stands for the statement 'after $\phi$ is upgraded in the state, $\psi$ is the case'. Finally, read $[\Uparrow\phi]\psi$ as 'after $\phi$ is lexically upgraded, $\psi$ is the case'.

We extend definition 3.3 once more.

**Figure 6.1.** The meaning of edges in doxastic and preference models is illustrated below. Substitute the specific notions 'more plausible than' or 'preferred over' for the general description 'better than' at will.

$$\begin{tikzcd} \phi & \neg\phi \\ w \arrow[loop left, "a"] \arrow[r, "a"] & w' \arrow[loop right, "a"] \end{tikzcd}$$

$w'$ and $\neg\phi$ are better than $w$ and $\phi$.

$$\begin{tikzcd} \phi & \neg\phi \\ w \arrow[loop left, "a"] \arrow[r, leftrightarrow, "a"] & w' \arrow[loop right, "a"] \end{tikzcd}$$

$w$, $w'$, $\phi$, and $\neg\phi$ are equally good.

**Definition 6.2.** Let the forcing relation «⊩» be as before, except for the following additional constraints:

$$\begin{aligned} \langle \mathcal{M}w\rangle \Vdash [\text{¡}\phi]\psi &\iff \langle \mathcal{M}|_{\text{¡}\phi} w\rangle \Vdash \psi \\ \langle \mathcal{M}w\rangle \Vdash [\sharp\phi]\psi &\iff \langle \mathcal{M}|_{\sharp\phi} w\rangle \Vdash \psi \\ \langle \mathcal{M}w\rangle \Vdash [\Uparrow\phi]\psi &\iff \langle \mathcal{M}|_{\Uparrow\phi} w\rangle \Vdash \psi \end{aligned}$$

where

$$\begin{aligned} \mathcal{M}|_{\text{¡}\phi} \coloneqq \mathcal{M} - \{\langle aw'w''\rangle \in \mathcal{M} \mid a \in \mathrm{Ind}^P,\ &\text{and} \\ \langle \mathcal{M}w'\rangle \Vdash \phi \text{ xor } \langle \mathcal{M}w''\rangle \Vdash \phi\}&, \end{aligned}$$

$$\begin{aligned} \mathcal{M}|_{\sharp\phi} \coloneqq \mathcal{M} - \{\langle aw'w''\rangle \in \mathcal{M} \mid a \in \mathrm{Ind}^P&, \\ \langle \mathcal{M}w'\rangle \Vdash \phi,\ \text{and } \langle \mathcal{M}w''\rangle \Vdash \neg\phi\}&, \end{aligned}$$

and

$$\begin{aligned} \mathcal{M}|_{\Uparrow\phi} \coloneqq \{\langle aw'w''\rangle \mid\ & a \in \mathrm{Ind}^B, \mathcal{M}aw'w'' \text{ or } \mathcal{M}aw''w', \\ & \langle \mathcal{M}w'\rangle \Vdash \neg\phi,\ \text{and } \langle \mathcal{M}w''\rangle \Vdash \phi\} \\ \cup\ \{\langle aw'w''\rangle \in \mathcal{M} \mid\ & a \notin \mathrm{Ind}^B \text{ or} \\ & \langle \mathcal{M}w'\rangle \Vdash \phi \iff \langle \mathcal{M}w''\rangle \Vdash \phi\} \end{aligned}$$

**Figure 6.2.** Example of the update operation $|_{¡\phi}$. It is assumed that $a \in \mathrm{Ind}^P$. The reflexive $a$-edges are not displayed.

$$\begin{tikzcd}
\phi & \neg\phi & \phi & & \\
w \arrow[r, "a"] \arrow[rrrr, bend left=40, "a"] \arrow[rr, bend right, "a"'] \arrow[rrr, bend right=40, "a"'] & w' \arrow[r, "a"] \arrow[rrr, bend left=30, "a"] \arrow[rr, bend right, "a"'] & w'' \arrow[r, "a"] \arrow[rr, bend left=20, "a"] & w''' \arrow[r, "a"] & w'''' \\
 & & & \neg\phi & \phi
\end{tikzcd}$$

The original model $\mathcal{M}$.

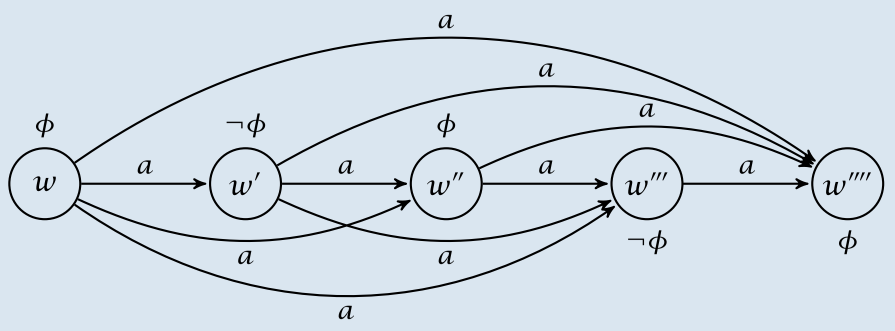
$$\begin{tikzcd}
w \arrow[rrrr, bend left=40, "a"] \arrow[rr, bend right, "a"'] & w' \arrow[rr, bend right, "a"'] & w'' \arrow[rr, bend left=20, "a"] & w''' & w''''
\end{tikzcd}$$

The updated model $\mathcal{M}|_{¡\phi}$.

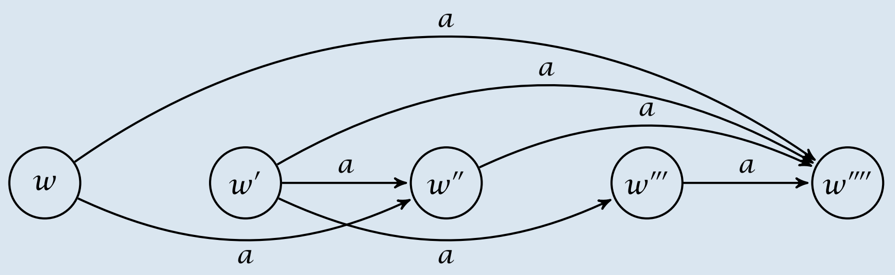
$$\begin{tikzcd}
w \arrow[r, "a"] \arrow[rr, bend left, "a"] & w'' \arrow[r, "a"] & w'''' & w' \arrow[r, "a"] & w'''
\end{tikzcd}$$

Another look at the updated model $\mathcal{M}|_{¡\phi}$.

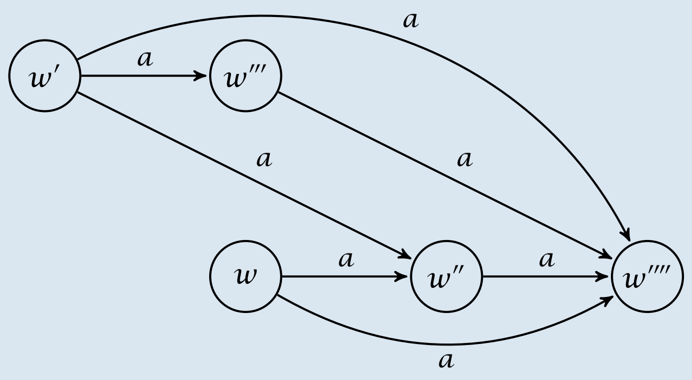
**Figure 6.3.** Example of the upgrade operation $|_{\sharp\phi}$. It is assumed that $a \in \mathrm{Ind}^P$. Loops are omitted.



The original model $\mathcal{M}$.

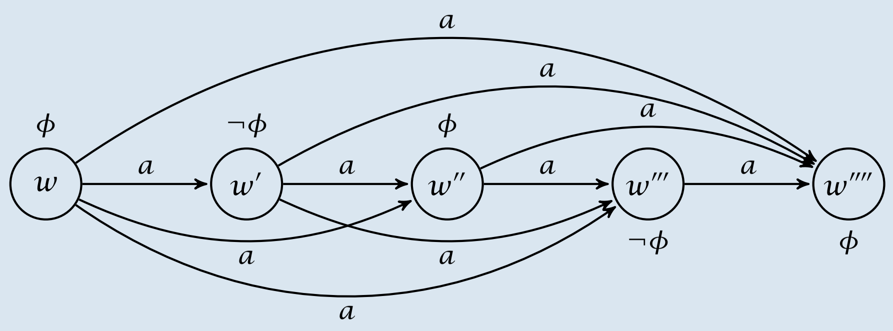


The upgraded model $\mathcal{M}|_{\sharp\phi}$.

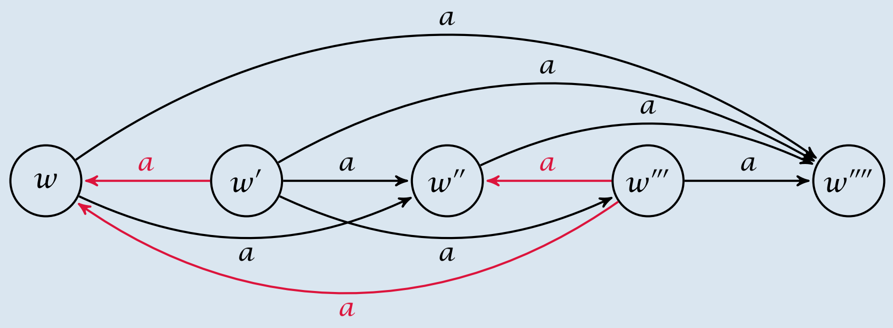


The model $\mathcal{M}|_{\sharp\phi}$ once more.

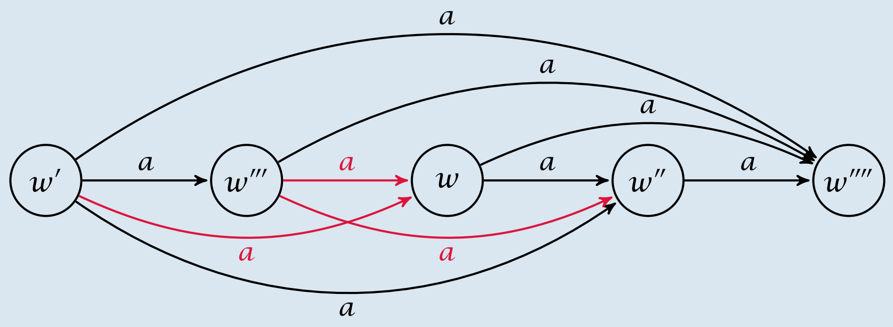
**Figure 6.4.** Example of the lexical upgrade operation $|_{\Uparrow\phi}$. It is assumed that $a \in \mathrm{Ind}^B$. New edges are drawn in red.



The model $\mathcal{M}$.



The lexically upgraded model $\mathcal{M}|_{\Uparrow\phi}$.



Once more the model $\mathcal{M}|_{\Uparrow\phi}$.

**Table 6.1.** For every $[\$\phi]\psi$ and $\neg[\$\phi]\psi]$ in the tableau (with \$ a placeholder for ¡, $\sharp$, and $\Uparrow$), we force every node to decide if $\phi$ or $\neg\phi$ should be true.

| Name | Precondition | Postcondition |
|---|---|---|
| $\mathbf{R}_{¡}$ | $\mathcal{T}n[¡\phi]\psi$ and $\mathcal{T}n'$ | $\mathcal{T}n'\phi$ or $\mathcal{T}n'\neg\phi$ |
| | $\mathcal{T}n\neg[¡\phi]\psi$ and $\mathcal{T}n'$ | $\mathcal{T}n'\phi$ or $\mathcal{T}n'\neg\phi$ |
| $\mathbf{R}_{\sharp}$ | $\mathcal{T}n[\sharp\phi]\psi$ and $\mathcal{T}n'$ | $\mathcal{T}n'\phi$ or $\mathcal{T}n'\neg\phi$ |
| | $\mathcal{T}n\neg[\sharp\phi]\psi$ and $\mathcal{T}n'$ | $\mathcal{T}n'\phi$ or $\mathcal{T}n'\neg\phi$ |
| $\mathbf{R}_{\Uparrow}$ | $\mathcal{T}n[\Uparrow\phi]\psi$ and $\mathcal{T}n'$ | $\mathcal{T}n'\phi$ or $\mathcal{T}n'\neg\phi$ |
| | $\mathcal{T}n\neg[\Uparrow\phi]\psi$ and $\mathcal{T}n'$ | $\mathcal{T}n'\phi$ or $\mathcal{T}n'\neg\phi$ |

In other words, the update operation $|_{¡\phi}$ deletes all preference links between $\phi$-worlds and $\neg\phi$-worlds. The upgrade operation $|_{\sharp\phi}$, on the other hand, deletes only preference links from $\phi$-worlds to $\neg\phi$-worlds.

Suppose an update operation $|_{¡p}$ (with $p$ an atom) is performed. From the point of view of a $p$-world $p$ then becomes preferable (as it would after a public announcement). Similarly, $\neg p$ becomes preferable for $\neg p$-worlds. If an upgrade operation $|_{\sharp p}$ is performed, however, then $p$ still becomes preferable for $p$-worlds but the preference of $\neg p$-worlds for $p$ is not affected.

The lexical upgrade operation $|_{\Uparrow\phi}$ changes doxastic plausibility orders. It makes $\phi$-worlds more plausible than $\neg\phi$-worlds. However, it only adds or removes $a$-edges between worlds that are connected over an $a$-edge to begin with.

## 6.3 Dynamic tableaux for dynamic preorder logics

We will now construct a tableau system for $\mathcal{L}^{U\Box¡\sharp\Uparrow}$ along familiar lines.

As usual, we start by adding rules for the dynamic operators. We add two rules for every dynamic modal operator.

**Definition 6.3.** Let the mandatory rules of Rules (and Rules$_\sigma$) include the additional rules in table 6.1.

Adapting the operations $|_{¡\phi}$, $|_{\sharp\phi}$, and $|_{\Uparrow\phi}$ to a tableau setting is again straightforward. This results in the following definitions.

**Definition 6.4.** For any two tableaux $\mathcal{T}$ and $\mathcal{T}'$ the relationship «$\mathcal{T} \xrightarrow{¡\phi} \mathcal{T}'$» holds if and only if

1. $\mathcal{T}'[?] = \mathcal{T}[?]$.

2. $\mathcal{T}'[???] = \mathcal{T}[???] - \{\langle ann' \rangle \mid a \in \text{Ind}^P\text{, and } \mathcal{T}n\phi \text{ xor } \mathcal{T}n'\phi\}$.

3. $\mathcal{T}'[??] \cap (\mathcal{T}'[?] \times \text{Prop}) = \mathcal{T}[??] \cap (\mathcal{T}'[?] \times \text{Prop})$.

The relationship «$\mathcal{T} \xrightarrow{\sharp\phi} \mathcal{T}'$» holds if and only if

1. $\mathcal{T}'[?] = \mathcal{T}[?]$.

2. $\mathcal{T}'[???] = \mathcal{T}[???] - \{\langle ann' \rangle \mid a \in \text{Ind}^P, \mathcal{T}n\phi\text{, and } \mathcal{T}n'\neg\phi\}$.

3. $\mathcal{T}'[??] \cap (\mathcal{T}'[?] \times \text{Prop}) = \mathcal{T}[??] \cap (\mathcal{T}'[?] \times \text{Prop})$.

The relationship «$\mathcal{T} \xrightarrow{\Uparrow\phi} \mathcal{T}'$» holds if and only if

1. $\mathcal{T}'[?] = \mathcal{T}[?]$.

2. $\mathcal{T}'[???] = \{\langle ann' \rangle \mid a \in \text{Ind}^B, \mathcal{T}ann' \text{ or } \mathcal{T}an'n, \mathcal{T}n\neg\phi\text{, and } \mathcal{T}n'\phi\}$
   $\cup\{\langle ann' \rangle \in \mathcal{T} \mid a \notin \text{Ind}^B \text{ or } \mathcal{T}n\phi \iff Tabn'\phi\}$

3. $\mathcal{T}'[??] \cap (\mathcal{T}'[?] \times \text{Prop}) = \mathcal{T}[??] \cap (\mathcal{T}'[?] \times \text{Prop})$.

Again we ammend the notion of 'open tableau', akin to how we added truth schemas for our new operators to the semantics.

**Definition 6.5.** Let the predicate «open» be as before, except for following added conditions:

1. If $\mathcal{T}n[¡\phi]\psi$ then there are no open saturated tableaux $\mathcal{T}'$ for $\langle n\neg\psi \rangle$ such that $\mathcal{T} \xrightarrow{¡\phi} \mathcal{T}'$.

2. If $\mathcal{T}n\neg[¡\phi]\psi$ then there is an open saturated tableau $\mathcal{T}'$ for $\langle n\neg\psi \rangle$ such that $\mathcal{T} \xrightarrow{¡\phi} \mathcal{T}'$.

3. If $\mathcal{T}n[\sharp\phi]\psi$ then there are no open saturated tableaux $\mathcal{T}'$ for $\langle n\neg\psi\rangle$ such that $\mathcal{T} \xrightarrow{\sharp\phi} \mathcal{T}'$.

4. If $\mathcal{T}n\neg[\sharp\phi]\psi$ then there is an open saturated tableau $\mathcal{T}'$ for $\langle n\neg\psi\rangle$ such that $\mathcal{T} \xrightarrow{\sharp\phi} \mathcal{T}'$.

5. If $\mathcal{T}n[\Uparrow\phi]\psi$ then there are no open saturated tableaux $\mathcal{T}'$ for $\langle n\neg\psi\rangle$ such that $\mathcal{T} \xrightarrow{\Uparrow\phi} \mathcal{T}'$.

6. If $\mathcal{T}n\neg[\Uparrow\phi]\psi$ then there is an open saturated tableau $\mathcal{T}'$ for $\langle n\neg\psi\rangle$ such that $\mathcal{T} \xrightarrow{\Uparrow\phi} \mathcal{T}'$.

As before we prove that $|_{¡\phi}$, $|_{\sharp\phi}$, and $|_{\Uparrow\phi}$ on the one hand and $\xrightarrow{¡\phi}$, $\xrightarrow{\sharp\phi}$, and $\xrightarrow{\Uparrow\phi}$ on the other hand are symmetric for $\phi$-declarative tableaux and the models that natively satisfy them.

**Lemma 6.1.** If $\mathcal{M}$ is the stock model of a $\phi$-declarative tableau $\mathcal{T}$ such that $\mathcal{M}$ natively satisfies $\mathcal{T}$ up to $\neg\phi$ then it holds that

1. $\mathcal{M}|_{¡\phi}$ is the stock model for $\mathcal{T}'$ if and only if $\mathcal{T} \xrightarrow{¡\phi} \mathcal{T}'$.

2. $\mathcal{M}|_{\sharp\phi}$ is the stock model for $\mathcal{T}'$ if and only if $\mathcal{T} \xrightarrow{\sharp\phi} \mathcal{T}'$.

3. $\mathcal{M}|_{\Uparrow\phi}$ is the stock model for $\mathcal{T}'$ if and only if $\mathcal{T} \xrightarrow{\Uparrow\phi} \mathcal{T}'$.

**Proof.** The three cases are similar. Below we provide the proof for the first case. We leave the rest as an exercise for the reader

Proof for the left-to-right part of the lemma. Assume that $\mathcal{M}|_{¡\phi}$ is the stock model of $\mathcal{T}'$. It follows that $\mathcal{T}'[?] = \mathcal{T}[?]$ and $\mathcal{T}'[??] \cap (\mathcal{T}'[?] \times \text{Prop}) = \mathcal{T}[??] \cap (\mathcal{T}'[?] \times \text{Prop})$ since (i) the operation $|_{¡\phi}$ doesn't delete worlds or labels, and (ii) $\mathcal{M}$ is the stock model of $\mathcal{T}$. We also have that $\mathcal{T}'[???] = \mathcal{M}|_{¡\phi}[???]$. From this it follows that $\mathcal{T}'[???] = \mathcal{T}[???] - \{\langle ann'\rangle \mid a \in \text{Ind}^P$, and $\mathcal{T}n\phi$ xor $\mathcal{T}n'\phi\}$ by definition of $|_{¡\phi}$ and because $\mathcal{M}$ natively satisfies $\mathcal{T}$ up to $\neg\phi$. This proves that $\mathcal{T} \xrightarrow{¡\phi} \mathcal{T}'$.

Proof from right to left. Assume that $\mathcal{T} \xrightarrow{¡\phi} \mathcal{T}'$. It follows that $\mathcal{T}'[?] = \mathcal{T}[?]$ and $\mathcal{T}'[??] \cap (\mathcal{T}'[?] \times \text{Prop}) = \mathcal{T}[??] \cap (\mathcal{T}'[?] \times \text{Prop})$ by definition of $\xrightarrow{¡\phi}$. Similarly, $\mathcal{M}|_{¡\phi}[?] = \mathcal{M}[?]$ and $\mathcal{M}|_{¡\phi}[??] = \mathcal{M}[??]$ by definition of $|_{¡\phi}$. Since $\mathcal{M}$ is the stock model of $\mathcal{T}$ it now follows that $\mathcal{M}|_{¡\phi}[?] = \mathcal{T}'[?]$ and $\mathcal{M}|_{¡\phi}[??] = \mathcal{T}'[??] \cap (\mathcal{T}'[?] \times \text{Prop})$. Because $\mathcal{M}$ natively satisfies $\mathcal{T}$ up to $\neg\phi$, we have $\mathcal{T}n\phi \iff \langle \mathcal{M}n \rangle \Vdash \phi$ for all $n \in \mathcal{T}[?]$. By definition of $\xrightarrow{¡\phi}$ it now also follows that $\mathcal{M}|_{¡\phi}[???] = \mathcal{T}[???]$. This concludes the proof that $\mathcal{M}|_{¡\phi}$ is the stock model for $\mathcal{T}'$.

We extend lemma 3.7 once more and prove that our new tableau rules preserve satisfiability.

**Lemma 6.2.** If a $\sigma$-model $\mathcal{M}$ satisfies a tableau $\mathcal{T}$ via a function $f$ then it is the case that

1. If $\mathcal{T}$ is not $\mathbf{R}_{¡}$-saturated then there is a tableau $\mathcal{T}' \in \mathbf{R}_{¡}[\mathcal{T}] - \{\mathcal{T}\}$ and a function $f'$ such that $\mathcal{M}$ satisfies $\mathcal{T}'$ via $f'$.
2. If $\mathcal{T}$ is not $\mathbf{R}_{\sharp}$-saturated then there is a tableau $\mathcal{T}' \in \mathbf{R}_{\sharp}[\mathcal{T}] - \{\mathcal{T}\}$ and a function $f'$ such that $\mathcal{M}$ satisfies $\mathcal{T}'$ via $f'$.
3. If $\mathcal{T}$ is not $\mathbf{R}_{\Uparrow}$-saturated then there is a tableau $\mathcal{T}' \in \mathbf{R}_{\Uparrow}[\mathcal{T}] - \{\mathcal{T}\}$ and a function $f'$ such that $\mathcal{M}$ satisfies $\mathcal{T}'$ via $f'$.

**Proof.** The four cases are similar. We only present the proof for the first one.

Suppose $\mathbf{R}_{¡}$ is triggered by $\mathcal{T}n[¡\phi]\psi$ and $\mathcal{T}n'$. Because it is assumed that $\langle \mathcal{M}f(n) \rangle \Vdash [¡\phi]\psi$ it follows that $\langle \mathcal{M}|_{¡\phi} f(n) \rangle \Vdash \psi$. Thus, let $f' \coloneqq f$.

Similarly, if $\mathcal{T}n\neg[¡\phi]\psi$ and $\mathcal{T}n'$ then not $\langle \mathcal{M}|_{¡\phi} f(n) \rangle \Vdash \psi$. Again, let $f' \coloneqq f$.

It is now time to prove soundness and completeness. As before we use mutual induction.

**Lemma 6.3 (Lemma to Soundness and Completeness).** Given a stock model $\mathcal{M}$ for a saturated tableau $\mathcal{T}$,

1. If $\mathcal{M}$ natively satisfies $\mathcal{T}$ and $\mathcal{T}$ is a tableau for $\phi$ then $\mathcal{T}$ is open.

2. If $\mathcal{T}$ is open and $\mathcal{T} n \phi$ then $\langle \mathcal{M} n \rangle \Vdash \phi$.

**Proof.** Proof by mutual induction on the length of $\phi$. Assume that the lemma holds for all $\psi$ shorter than $\phi$.

**Part 1.**

- In lemmas 3.7 and 6.2 it is proved that $\mathcal{T}$ is free of literal contradictions.
- For all $\langle n[¡\psi]\chi \rangle \in \mathcal{T}$ it must be demonstrated that there is no open saturated tableau $\mathcal{T}'$ for $\langle n \neg \chi \rangle$ such that $\mathcal{T} \xrightarrow{¡\psi} \mathcal{T}'$.

  Proof by contradiction. Suppose such a tableau $\mathcal{T}'$ did exist. By Part 2 of the IH it is then the case that, with $\mathcal{M}'$ the stock model of $\mathcal{T}'$, $\langle \mathcal{M}' n \rangle \Vdash \neg \chi$. From $\mathbf{R}_{¡}$ and the premise that $\mathcal{T}$ is saturated it follows that $\mathcal{T}$ is $\psi$-declarative. By lemma 6.1 it now follows that $\mathcal{M}' = \mathcal{M}|_{¡\psi}$.

  However, since $\mathcal{M}$ natively satisfies $\mathcal{T}$ it is the case that $\langle \mathcal{M} n \rangle \Vdash [¡\psi]\chi$ and $\langle \mathcal{M}|_{¡\psi} n \rangle \Vdash \chi$. This ends the proof by contradiction.
- For all $\langle n \neg [¡\psi]\chi \rangle \in \mathcal{T}$ it must be demonstrated that there is an open saturated tableau $\mathcal{T}'$ for $\langle n \neg \psi \rangle$ such that $\mathcal{T} \xrightarrow{¡\psi} \mathcal{T}'$.

  As $\mathcal{M}$ natively satisfies $\mathcal{T}$ it is the case that $\langle \mathcal{M} n \rangle \Vdash \neg [¡\psi]\chi$. By definition of $[¡\psi]$ and $\neg$ it follows that $\langle \mathcal{M}|_{¡\phi} n \rangle \Vdash \neg \chi$.

  By Part 1 of the IH it follows that the stock tableau $\mathcal{T}'$ for $\langle \mathcal{M}|_{¡\phi} n \neg \psi \rangle$ is open. Moreover, $\mathcal{T}$ is $\psi$-declarative because of $\mathbf{R}_{¡}$. By lemma 6.1 it now follows that $\mathcal{T} \xrightarrow{¡\psi} \mathcal{T}'$ and this concludes our proof.
- The proof for other dynamic formulas of $\mathcal{L}^{U \Box ¡ \sharp \Uparrow}$ are entirely analogous to the proofs for formulas $[¡\psi]\chi$ and their negations.

**Part 2.** The bulk of this proof has already been covered in lemma 3.9. We only provide the proofs for the patterns specific to $\mathcal{L}^{U \Box ¡ \sharp \Uparrow}$.

- Suppose $\phi$ is $[¡\psi]\chi$. It has to be demonstrated that $\langle \mathcal{M}|_{¡\psi} n \rangle \Vdash \chi$.

  Proof by contradiction. Assume that $\langle \mathcal{M}|_{¡\psi} n \rangle \Vdash \neg \chi$.

By Part 1 of the IH it follows that any stock tableau $\mathcal{T}'$ for $\langle \mathcal{M}|_{¡\psi} n \neg\chi\rangle$ is open. Moreover, $\mathcal{T}$ is $\psi$-declarative by $\mathbf{R}_{¡}$. By Part 2 of the IH it follows that $\mathcal{M}$ natively satisfies $\mathcal{T}$ up to $\neg\psi$. By lemma 6.1 it now follows that $\mathcal{T} \xrightarrow{¡\psi} \mathcal{T}'$.

However, since $\mathcal{T}$ is an open tableau we know that there is no saturated open tableau $\mathcal{T}''$ for $\langle n \neg\chi\rangle$ such that $\mathcal{T} \xrightarrow{¡\psi} \mathcal{T}''$. This concludes the proof by contradiction.

- Suppose $\phi$ is $\neg[¡\psi]\chi$. We need to demonstrate that $\langle \mathcal{M}|_{¡\psi} n\rangle \Vdash \neg\chi$.

  First, since $\mathcal{T}$ is open it follows that there is an open saturated tableau $\mathcal{T}'$ for $\langle n\neg\chi\rangle$ such that $\mathcal{T} \xrightarrow{¡\psi} \mathcal{T}'$.

  Second, $\mathcal{T}$ is $\psi$-declarative by $\mathbf{R}_{¡}$. Hence, by Part 2 of the IH it also follows that $\mathcal{M}$ natively satisfies $\mathcal{T}$ up to $\neg\psi$. By lemma 6.1 it is now the case that $\mathcal{M}|_{¡\psi}$ is the stock model of $\mathcal{T}'$.

  Finally, again by Part 2 of the IH, it is the case that $\langle \mathcal{M}|_{¡\psi} n\rangle \Vdash \neg\chi$.

- The proofs for $[\sharp\psi]\chi$, $[\Uparrow\psi]\chi$, and their negations are similar to those of $[¡\psi]\chi$ and $\neg[¡\psi]\chi$.

## 6.4 Decidability of tableaux for $\mathcal{L}^{U\Box¡\sharp\Uparrow}$

In this section we construct a decidable proof procedure for $\mathcal{L}^{U\Box¡\sharp\Uparrow}$. We proceed along familiar lines. That is, we define priming operations for the dynamic formulas of this language and we show that priming and synchronizing operations preserve satisfiability. But first we define what satisfaction means for tableau cascades in light of our new operators.

**Definition 6.6.** A pointed tableau cascade $\langle Ct\rangle$ is «satisfied» by a model $\mathcal{M}$ via a function $f$ if and only if

- $\mathcal{M}$ satisfies $C(t?)$ via $f$.
- For every $\langle \xrightarrow{¡\phi} t'\rangle \in C[?t?]$ it is the case that $\mathcal{M}|_{¡\phi}$ satisfies $\langle Ct'\rangle$ via $f$.

- For every $\langle \xrightarrow{\sharp\phi} t' \rangle \in C[?t?]$ it is the case that $\mathcal{M}|_{\sharp\phi}$ satisfies $\langle Ct' \rangle$ via $f$.
- For every $\langle \xrightarrow{\Uparrow\phi} t' \rangle \in C[?t?]$ it is the case that $\mathcal{M}|_{\Uparrow\phi}$ satisfies $\langle Ct' \rangle$ via $f$.

We say that a tableau cascade $C$ is satisfied by a model $\mathcal{M}$ via a function $f$ if and only if $\langle C\,\mathrm{root}(C) \rangle$ is satisfied by $\mathcal{M}$ via $f$.

If desired the above definition could be made compatible with definitions 4.9 and 5.8. This would result in a decidable tableau system for the language that results from combining all previously defined operators.

We now define a priming operation for our new operators.

**Definition 6.7.** $C'$ is the result of «¡♯⇑-priming» a tableau cascade $C$ if and only if $C'$ is a minimal extension of $C$ (and the tableau therein) such that for all $\langle t\mathcal{T} \rangle \in C$,

- If $\phi = [¡\phi]\psi$ then there is a node-tableau pair $\langle t'\mathcal{T}' \rangle \in C'$ such that $\mathcal{T}'$ is a tableau for $\langle n\psi \rangle$ and $\langle \xrightarrow{¡\phi} tt' \rangle \in C'$.
- If $\phi = \neg[¡\phi]\psi$ then there is a node-tableau pair $\langle t'\mathcal{T}' \rangle \in C'$ such that $\mathcal{T}'$ is a tableau for $\langle n\neg\psi \rangle$ and $\langle \xrightarrow{¡\phi} tt' \rangle \in C'$.
- If $\phi = [\sharp\phi]\psi$ then there is a node-tableau pair $\langle t'\mathcal{T}' \rangle \in C'$ such that $\mathcal{T}'$ is a tableau for $\langle n\psi \rangle$ and $\langle \xrightarrow{\sharp\phi} tt' \rangle \in C'$.
- If $\phi = \neg[\sharp\phi]\psi$ then there is a node-tableau pair $\langle t'\mathcal{T}' \rangle \in C'$ such that $\mathcal{T}'$ is a tableau for $\langle n\neg\psi \rangle$ and $\langle \xrightarrow{\sharp\phi} tt' \rangle \in C'$.
- If $\phi = [\Uparrow\phi]\psi$ then there is a node-tableau pair $\langle t'\mathcal{T}' \rangle \in C'$ such that $\mathcal{T}'$ is a tableau for $\langle n\psi \rangle$ and $\langle \xrightarrow{\Uparrow\phi} tt' \rangle \in C'$.
- If $\phi = \neg[\Uparrow\phi]\psi$ then there is a node-tableau pair $\langle t'\mathcal{T}' \rangle \in C'$ such that $\mathcal{T}'$ is a tableau for $\langle n\neg\psi \rangle$ and $\langle \xrightarrow{\Uparrow\phi} tt' \rangle \in C'$.

The new priming operation preserves the property that if a tableau cascade $C$ is satisfiable then so is any tableau cascade that results from priming $C$.

**Lemma 6.4.** Given a tableau cascade $C$ that is satisfied by a model $\mathcal{M}^\star$ via a function $f$, if $C'$ is the result of ¡♯⇑-priming $C$ then $C'$ is satisfied by $\mathcal{M}^\star$ via $f$.

**Proof.** Suppose $\langle n[¡\phi]\psi\rangle \in \mathcal{T}$, where $\mathcal{T}$ is a tableau in $C$, and suppose $\mathcal{T}' = \{n, \langle n\phi\rangle\}$ is a tableau in $C'$ but not $C$. Let $\mathcal{M}'$ be any model and $f'$ any function such that $\langle \mathcal{M}' f'(n)\rangle \Vdash [¡\phi]\psi$. Such a model exists by assumption and we want to prove that $\langle \mathcal{M}'|_{¡\phi} f'(n)\rangle \Vdash \psi$. In fact, this immediately follows from the definition of $[¡\phi]$. It also follows that $\mathcal{M}^\star$ satisfies $C'$ via $f$.

The proofs for negated $[¡\phi]$ formulas and for the other operators are structurally identical to the above case and we omit them here.

Synchronization also preserves satisfiability.

**Lemma 6.5.** Where \$ stands for ¡, ♯, or ⇑, let $C$ be a tableau cascade such that $\langle \xrightarrow{\$\phi} tt'\rangle \in C$ and such that $C(t?)$ is satisfied by a model $\mathcal{M}$ via a function $f$ and $C(t'?)$ is satisfied by $\mathcal{M}|_{\$\phi}$ via $f$. If $C$ can be synchronized such that at most $t$ and $t'$ are affected then there's an outcome $C'$ of synchronizing $C$ such that $\mathcal{M}$ satisfies $C'(t?)$ via $f$ and $\mathcal{M}|_{\$\phi}$ satisfies $C'(t'?)$ via $f$.

**Proof.** The proofs are similar to the proof for lemma 4.10.

We have now implemented the scaffolding for using dynamic tableaux in a completely automated fashion.

**Theorem 6.6.** If there is a pointed $\sigma$-model $\langle \mathcal{M}w\rangle$ such that $\langle \mathcal{M}w\rangle \Vdash \phi$ (where $\phi \in \mathcal{L}^{U\Box¡\sharp\Uparrow}$) then an open saturated $\sigma$-tableau $\mathcal{T}$ for $\langle w\phi\rangle$ can be found in a finite number of steps.

**Proof.** Our dynamic tableaux for preorder logics introduce no difficulties that were not already covered in theorem 4.11. Therefore, we omit the decidability proof.

## 6.5 Related work

To the best of our knowledge no tableau system exists for dynamic doxastic logic. We also don't know of a tableau system that directly implements dynamic preference logic. Indirectly, however, such tableau systems exist since the update and upgrade operations discussed in this chapter can be represented by action models. Lexical upgrades cannot be modeled by the operation $|_{\otimes\phi}$. In [6] this limitation is explained and an alternative kind of action model update—named 'action-priority update'—is presented that can express lexical upgrades. Again, though, we know of no tableau system for action priority update logic.

The lexical upgrade operator was first introduced in [33]. See [5, §1] for a discussion of how it relates to the formalisms in [6] and in this chapter.

For a detailed discussion of dynamic preference logics, see [28].

# 7 Clojure Implementation

In this chapter we present a proof-of-concept computer implementation of our dynamic tableau system for public announcement logic. It is implemented in the programming language Clojure. Clojure is a functional programming language in the style of LISP. It is a young programming language but has several things going for it that make it well-suited to our purposes.

Functional programming languages are heavily inspired by lambda calculus. Unlike most other types of programming languages this means that they are organized in a way to encourage the use of immutable data structures and functions as primary means of abstraction. This makes such languages a good fit for implementing procedures that were first described using mathematical notation. Allow us to illustrate this point with an example.

Computer programs usually store ordered sequences of elements in one of two data structures—'lists' or 'vectors'. Ignoring implementation details, the key difference is that it's very fast to access, add, or remove elements at the front of a list, whereas it's very fast to access elements from a vector regardless their position in the sequence. The trade-off is that adding or removing elements at the front of a vector is slower. Suppose that we want to sort a vector. In a functional programming language we would use a procedure—called a function—that yields or 'returns' a new vector. In other (imperative) languages the original vector is typically modified and the original data is lost.

Using Clojure also has performance advantages because Clojure programs are executed on the Java Virtual Machine (JVM). Clojure programs are translated or ‘compiled’ to an intermediate representation called ‘Java bytecode’. The JVM is then asked to run this bytecode. The JVM is a mature software platform and is very efficient at this task. Another advantage is that Clojure has good abstractions for writing code that performs tasks in parallel.

Performance benefits are important because theorem provers are inherently computationally intensive applications. Code parallelism is especially important because in recent years high-end computer processors have only seen modest year-over-year improvements with respect to running sequential programs. There are, on the other hand, clear trends towards more parallel computing architectures.

Clojure programs are highly portable. First of all, because they are run on the JVM they can be executed on any computer architecture and operating system that can run Java programs. This includes Windows, OS X, Linux, and Android. Second, there is a dialect of Clojure that is compiled to JavaScript. This dialect, named ClojureScript, is very similar to Clojure and it is in fact typical that large parts of a Clojure programs are also valid ClojureScript code. Such programs could, for instance, be run on the JVM when performance is essential and in a web browser when user-friendly distribution of the program is deemed more important.

In the remainder of this chapter we implement the proof system from chapter 4 in Clojure. We present it as a proof-of-concept. We set out to create highly parallel code but also tried to be true to the core principles of our approach. This means that we refrained from trying several optimizations that we felt might work. We leave these for further research.

## 7.1 A brief overview of Clojure

In this section we briefly outline the major distinguishing features and syntax of Clojure. We refer the reader to [17] for a more thorough introduction to this programming language.

Being a variant on LISP, Clojure is homoiconic. This means that it is possible to interpret Clojure source code in terms of Clojure data structures in a straightforward way. Take the following program:

```
(defn is-even? [n]
  (if (= (mod n 2) 0)
    true
    false))
```

The above code is represented by the following list:

- The ‘symbol’ ‹defn›.
- The symbol ‹is-even?›.
- The following vector:
  - The symbol ‹n›.
- The following list:
  - The symbol ‹if›.
  - The following list:
    - The symbol ‹=›.
    - The following list:
      - The symbol ‹mod›.
      - The symbol ‹n›.
      - The integer ‹2›.
    - The integer ‹0›.

This is also called an abstract syntax tree (AST).

In a Clojure AST, symbols are usually names or ‘bindings’. Bindings are primarily lexically scoped. This means they behave like lambda abstraction.

The code above defines a function ‹is-even?› that takes one argument, bound to ‹n›. The function itself is in fact bound by ‹defn› to the variable name ‹is-even?› in the current ‘namespace’. It can also be accessed from other namespaces or files. We’ll look into how to do this in a bit. For now we merely want to point out that the code above already uses this feature: The functions ‹=› and ‹mod› are actually defined in the namespace ‹clojure.core›.

In Clojure programs, lists are usually interpreted as invocations of functions, 'special forms', or 'macros'. Special forms are used for fundamental constructs like if-then-else branching, variable declarations, and function definitions. We will discuss macros later on. The first element in a list is the name of the function, special form, or macro. It is executed with the remaining elements in the list serving as arguments.

Other data structures—including vectors, numbers, and booleans—are not evaluated further. Therefore the Clojure code ‹[1 2 3]› corresponds to a vector of the numbers 1, 2, and 3 in the AST and in the program. Observe that we cannot include a list of the numbers 1, 2, 3 in our program by typing ‹(1 2 3)›. Instead we are expected to type ‹(list 1 2 3)›.

With the very basics out of the way, let's look at those functions, special forms, and macros built-in to Clojure that we will be using. We will also introduce some extra syntax along the way.

We start by giving some more examples of ‹if› statements, and of the special form ‹cond›. Note that semicolons start a comment and absorb everything up to the end of the line. Strings (sequences of characters) are surrounded by double quotes.

```
(if false 1  2) ; 2
(if nil 1 2) ; 2
(if "for sure" 1 2) ; 1 (only false and nil are 'falsy')
(if false 1) ; nil
(if-not false 1 2) ; 1

; a single cond-statement can replace multiple if-statements
(cond (= 1 3) :one
      (= 2 3) :two
      (= 3 3) :three
      :else :i-give-up)
```

To define a variable the 'special form' ‹def› is used.

```
(def x "text")
; the variable x is bound to "text" from this point on
```

There are several ways to create functions.

```
(defn f [x y]
  (+ x y))
; from here on (f 3 4) evaluates to 7

(def f (fn [x y] (+ x y))) ; this has the same effect

((fn [x] (* x x)) 3) ; 9
```

Functions that are not assigned a variable name are often called lambdas. There's also a special form for creating functions with exactly one argument.

```
#(* % %) ; equivalent to (fn [x] (* x x))
```

The following syntax enables us to create functions that take a variable number of arguments.

```
(defn count-args [x & xs] (+ 1 (count xs)))
; (count-args "a" 10 5.2 [1 2 3 4 5]) evaluates to 4
```

Above, in the function body of ‹count−args›, ‹x› evaluates to ‹*"a"*› and ‹xs› evaluates to the list ‹(10 5.2 [1 2 3 4 5])›.

Functions can dispatch to different bodies of code depending on the number of arguments passed.

```
(defn f
  ([] "zero") ; (f) -> "zero"
  ([x] "one") ; (f nil) -> "one"
  ([x y] "two") ; (f "a" "b") -> "two"
  ([x y & xs] "more")) ; (f [] [] []) -> "more"
```

Function arguments that are vectors can also be 'destructured'. That is to say, the elements of the vector can directly be assigned to individual variables. Here's an example:

```
(defn str-birthday [name [[year month day] place]]
  (str name " was born in " place
         " on " day " "
```

```
        (get {1 "January" ; etc.
              6 "June" ; etc.
              12 "December"}
             month)
        " " year "."))
; e.g. (str-birthday "Rousseau" [[1712 6 28] "Geneva"])
```

The above example also illustrates the use of ‘maps’ (or dictionaries). We discuss more examples of maps further down. The function ‹str› converts each of its arguments to a string and appends the results.

Bindings can also be introduced using the special form ‹let›.

```
(defn f [n]
  (let [x (* n n)
        y (* x x)]
    (+ y 1)))
; equivalent to (+ (* n n n n) 1)
```

Note that it’s also possible to use destructuring in combination with ‹let›. We also want to use this opportunity to mention an advanced destructuring feature. It is possible to destructure a sequence—that is, a list or a vector—and create a binding for the entire sequence at the same time

```
(let [[a b & xs :as l] [1 2 3 4 5]]
  l)
; [1 2 3 4 5]
```

Here’s a variant on ‹let› for defining functions:

```
(letfn [(f [x] (inc x))]
  (f 3))
```

Note that the functions ‹inc› and ‹dec› increment and decrement their argument by one.

Clojure has a unique syntax for loops.

```
(defn repeat-string [n s]
  (loop [n n
          acc ""]
    (if (= n 0)
      acc
      (recur (dec n) (str acc s)))))
; (repeat-string 5 "ab") evaluates to "ababababab"
```

Every time ‹recur› is used, new values are bound to ‹n› and ‹acc› and the loop body is executed once more.

The first thing at the top of a Clojure source file is normally a namespace declaration. For this, ‹ns› is used; ‹ns› is also used to make outside namespaces accessible.

```
(ns myproject.this
    (:require [clojure.core.reducers :as r]
              [clojure.set :refer :all]
              [myproject.that]))
```

The above instruction makes the function ‹map› from the namespace ‹clojure.core.reducers› accessible as ‹r/map›. It also binds the ‹intersection› to the intersection function from the namespace ‹clojure.set›. Functions and macros from ‹clojure.core› are made available without a prefix by default. Since the namespace of this file is called ‹myproject.this›, the file should be stored in file named 'this.clj' in a folder 'myproject'. Similarly the last line loads the file 'that.clj' in the same folder.

The syntax for ‹ns› is a bit peculiar as ‹:require› is not a function but a keyword. All terms starting with a colon are keywords. You can think of keywords as strings that are not intended to be manipulated and are not intended to be ever shown to the user. Still, the question remains why ‹:require› is at the start of a list. Without disclosing the details we want to use this case as a segue to one feature typical of homoiconic languages.

Macros allow powerful transformations on parts of the Clojure AST. When a macro is invoked, the expressions passed to it are rewritten into a new expression. This happens at compile time. The program is only run after all macros have expanded into macro-free code. Macros are a great feature because they allow Clojure to have a simple yet extensible syntax.

There are two important macros that we will use throughout this chapter. They are called the 'threading' macros. The macro ‹->› takes a variable number of arguments. Given two expressions for arguments, the second one being a list, it inserts the first expression as the second element of the second expression. Given a third argument—also a list—it inserts the resulting expression as the second element of the third expression. And so on.

```
(-> foo
    (bar baz))
; expands to (bar foo baz)

(-> foo
    (bar baz)
    (qux quux quuux))
; expands to (qux (bar foo baz) quux quuux)
```

The second threading macro ‹->>› is similar to ‹->› but inserts results as the last, rather than second, element of the next expression.

```
(->> foo
     (bar baz))
; expands to (bar baz foo)

(->> foo
     (bar baz)
     (qux quux quuux))
; expands to (qux quux quuux (bar baz foo))
```

Relatedly, the function ‹apply› can be used to change how functions are called. The special form ‹partial› can be used for partial application of functions.

```
(apply + [1 2 3 4]) ; equivalent to (+ 1 2 3 4)

(def f (partial + 1 2 3))
(f 4 5) ; (+ 1 2 3 4 5)
```

Let's now turn our attention to functions that operate on collections—lists, vectors, sets, and maps. Once more we illustrate their usage by example.

```
(list 1 2 3 4) ; list
(conj (list 1 2 3 4) 5) ; (list 5 1 2 3 4) (prepends)
(cons 1 (list 2 3 4 5)) ; (list 1 2 3 4 5)

[1 2 3 4] ; vector
(vector 1 2 3 4) ; same
(conj [1 2 3 4] 5) ; [1 2 3 4 5] (appends)
(cons 1 [2 3 4 5]) ; (list 1 2 3 4 5)

(first [1 2 3]) ; 1
(second (list 1 2 3)) ; 2
(nth [1 2 3] 2) ; 3
(next [1 2 3]) ; [2 3]

#{1 2 3 4} ; set
(hash−set 1 2 3 4) ; same
(contains? #{1 2 3} 3) ; true
(conj #{1 2 3 4} 5) ; #{1 2 3 4 5}

{:a 1 :b 2 :c 3} ; map
(hash−map :a 1 :b 2 :c 3) ; same
(get {:a 1 :b 2 :c 3} :b) ; 2
(conj {:a 1 :b 2} [:c 3]) ; adds a key−value pair
(assoc {:a 1 :b 2} :c 3) ; same

(count [1 2 3]) ; 3

(into [1 2] [3 4]) ; same as (conj (conj [1 2] 3) 4)
```

We need to do a lot of computations on collections. For this purpose we use Clojure's 'combine-reduce' infrastructure. It allows us to perform such operations efficiently.

Every collection we have mentioned is ‘reducible’. This means they can be transformed using the functions ‹map›, ‹mapcat›, and ‹ filter › from the namespace ‹clojure.core.reducers›. This yields a new but abstract collection. This abstract collection keeps a reference to the original collection and to the operations that have to be performed. Notice that ‹map› and the other functions we just mentioned don’t actually perform any transformations on the original collection. Afterwards the function ‹reduce› can be used to materialize the modified collection as, for example, a vector.

Some collections are also ‘foldable’. For these collections the function ‹fold› can be used instead of ‹reduce› to perform the scheduled transformations in parallel. When ‹fold› is applied to a non-foldable but reducible collection it simply calls ‹reduce›.

We illustrate by example. Assume the namespace ‹clojure.core.reducers› has been aliased to ‹r›.

```
(->> [1 10 100 1000] ; original collection
     (r/map #(* 2 %)) ; multiply every element by 2
     (r/reduce conj #{})) ; conj every element into a set
; #{2 20 200 2000}

(->> [1 10 100 1000]
     (r/mapcat (fn [x] [x (* 2 x)]))
     ; 1, 2, 10, 20, 100, 200, 1000, 2000
     (r/filter #(> % 100))
     ; 200, 1000, 2000
     (r/reduce conj [5]))
; [5 200 1000 2000]

(->> (range 1 5000) ; create a non-foldable collection
     (apply vector) ; convert the list to a (foldable) vector
     (r/map -)
     (r/fold (fn
               ([] #{})
               ([left right] (into left right)))
```

```
                 conj))
; #{-1 -2 -3 -4 -5 -6 -7 -8 -9 -10 ... }
```

The first type of argument passed to ‹fold› is called a monoid in Clojure parlance. Clojure monoids are functions that, when passed zero arguments, return an identity element. When passed two arguments, both of the same type as the identity element, it combines them. This latter operation must be associative. This also explains why they are called monoids—their namesake algebraic structures are structures that contain an identity element and an associative operation.

When ‹fold› is applied to a foldable collection, the collection is partitioned into large classes of elements. The previously discussed function ‹reduce› is applied to every one of these classes. Its first argument is the second argument passed to ‹fold› (in the example above this function is ‹conj›) and its second argument is the monoid's identity element.

It is also possible to apply ‹map› (the function) to maps (the data structure). Simply pass a function to ‹map› that takes two arguments—a key and a value.

```
(->> {:a 1 :b 2}
     (r/map (fn [key val] [key (- val)]))
     (r/fold (r/monoid into (constantly {})) conj))
; {:a -1 :b -2}
```

The function ‹constantly› creates a function that takes any number of arguments and always returns the same value. The function ‹monoid› creates a monoid, where the first argument is the combining function and the second argument is a function that returns the identity element.

Notice that, as a matter of fact, the original map in the previous example is small and would therefore not actually be processed in parallel.

A naive way to fold a collection ‹coll› would be to write the following code:

```
(r/fold (r/monoid into vector) conj coll)
```

This, however, involves a lot of unnecessary copying. For first the collection is divided into several parts—provided it is foldable and contains many elements—and each of these parts is reduced to a vector. Afterwards the elements of these vectors, starting from the second vector, are appended to the first one. A better solution is to write ‹(r/foldcat coll)›. This will first reduce different parts of ‹coll› to a vector-like structure (technically, Java's ‹ArrayList›) and append them in a list-like structure. The result is *usually* foldable—we omit the specifics but will come back to this at the end of section 7.6.

‹foldcat› uses the monoid ‹cat› internally. This monoid can also be used to concatenate two foldable collections.

```
(r/cat coll1 coll2)
; similar to (into coll1 coll2), but faster
```

Finally, we want to mention that it's sometimes possible to omit the second argument—the function to use in the reducing phase—to ‹fold›. In this case the monoid is used instead.

```
(r/fold (r/monoid + (constantly 0))
        (apply vector (range 0 10000)))
; equals (apply + (range 0 10000))
```

We have now discussed the basics of Clojure. We use more features of Clojure than have been explained so far, but we will explain these as we discuss our Clojure program.

## 7.2 Utility functions

We start by defining a namespace ‹dyntab.util›. Here we define functions that will prove useful, but are more broad in scope than our current enterprise—creating a theorem prover.

First, we register the namespace and load two required Clojure package. We already discussed the package ‹clojure.core.reducers› in the previous section. The functions in ‹clojure.set› have self-explaining names.

```
(ns dyntab.util
  (:require [clojure.core.reducers :as r]
            [clojure.set :as set]))
```

We will make a lot of use of multi-maps. We represent them using maps that have sets for values.

```
(defn mmap-get [m k]
  (get m k #{}))

(defn mmap-conj [m [k v]]
  (assoc m k (conj (mmap-get m k)
                   v)))

(defn mmap-merge [& ms]
  (apply merge-with into ms))

(defn map-merge-overwrite [& ms]
  (apply merge-with (fn [x y] y) ms))

(defn mmap-get-unique [m k]
  (let [v (mmap-get m k)]
    (if-not (= (count v) 1)
      (throw (str "No unique value for " k " in " m))
    (first v))))

(defn mmap-diff [m1 m2]
  (->> m1
       (r/map (fn [[k v]]
                [k (set/difference v (get m2 k #{}))]))
       (r/filter (fn [[k v]]
                   (not (= v #{}))))
       (r/reduce conj {})))
```

The first function here, ‹mmap−get› is straightforward. When querying a multi-map, if there's no value that corresponds to our key ‹k› then we prefer to get an empty set rather than the value ‹nil›. The second function, ‹mmap−conj›, defines the preferred way of adding a single element to a multi-map—viz. by adding it to the set currently associated with its key.

Next we have two functions for merging (multi-)maps. The first merge function assigns to every key ‹k› the union of all sets associated with ‹k› in the multi-maps ‹ms›. The second merge function represents a different way of 'merging' multi-sets. To every key ‹k› it assigns the value associated with ‹k› in the rightmost multi-map ‹m› (in the order that they are passed to ‹mmap−merge−overwrite›) that assigns a nonempty set to ‹k›. ‹mmap−merge−overwrite› assumes that whenever a multi-map assigns a nonempty set to a key ‹k›, it internally stores no value at all for ‹k›; this is a difference that is 'hidden' by ‹mmap−get›.

The call ‹(mmap−get−unique m k)› is a shorthand for the expression ‹(first (mmap−get m k))›, except that it raises an error if ‹m› does not associate a singleton with the key ‹k›.

Finally, the function ‹mmap−diff› returns the multi-map that assigns to every ‹k› the set that results from subtracting the set associated with ‹k› in the second argument from the set associated with ‹k› in the first argument.

The following function is an efficient way to check if a foldable collection that results from applying ‹mapcat› or ‹filter› is empty. (For other collections, ‹count› can be used.)

```
(defn fold−empty? [coll]
  (r/fold (fn
            ([] true)
            ([left right] (and left right)))
          (constantly false)
          coll))
```

Maps and sets will be our preferred data types for storing collections of objects. The following function folds a foldable collection into a set.

```
(defn foldset [coll]
  (r/fold set/union conj coll))
```

Next up is the function ‹rjuxt›. Given a collection ‹coll› and a function ‹f› that maps every element of ‹coll› onto a set, ‹rjuxt› returns a foldable collection of all lists ‹[x y]› such that ‹x› is an element of ‹coll› and ‹y› is an element of ‹(f x)›.

```
(defn rjuxt [f coll]
  (r/mapcat (fn [x] (r/map (fn [y] [x y])
                           (f x)))
            coll))
```

Given a sequence of sets, the following function first removes the empty sets and then returns the Cartesian product of the remaining sets.

```
(defn one-from-each [coll-of-colls]
  (letfn [[f [acc rem]
           (cond (empty? rem) [acc]
                 (empty? (first rem)) (f acc (next rem))
                 :else (->> (first rem)
                            (r/mapcat #(f (conj acc %)
                                          (next rem)))))]]
    (f #{} coll-of-colls)))
```

With the above definitions we have enough tools to build a theorem prover.

## 7.3 Syntax

We start by defining the syntax. We interpret the keywords ‹:p›–‹:t› as atomic propositions and the keywords ‹:a›–‹:e› as indices.

```
(ns dyntab.syntax)

(def Prop #{:p :q :r :s :t})

(def Ind #{:a :b :c :d :e})
```

Complex formulas are represented by vectors. Thus we write the negation of ‹A› as ‹[:not A]›, the conjunction of ‹A› and ‹B› as ‹[:and A B]›, box-‹x› ‹A› as ‹[:box x A]›, and '‹B› holds after announcing ‹A›' as ‹[:! A B]›.

The function ‹wff?› tests if its argument is a well-formed formula.

```
(defn wff? [form]
  (try
    (case [(count form) (nth form 0)]
      [2 :not] (wff? (nth form 1))
      [3 :and] (and (wff? (nth form 1))
                    (wff? (nth form 2)))
      [3 :box] (and (contains? Ind (nth form 1))
                    (wff? (nth form 2)))
      [3 :!] (and (wff? (nth form 1))
                  (wff? (nth form 2)))
      false)
    (catch Exception x ; not a coll or empty coll
      (contains? Prop form))))
```

The ‹case› construct matches the result of clj(count form) and ‹nth form 0› against four possibilities. If matching fails then ‹false› is returned.

If ‹form› is a proposition (or otherwise not a collection) then ‹(count form)› raises an exception or error. Similarly, if ‹form› is an empty collection then ‹(nth form 0)› raises an exception. In these case we return ‹true› if and only if ‹form› is an atomic proposition.

## 7.4 Tuple bags

In this section we define the data type that we will use to store lgraphs. We call this data type a ‘tuple bag’.

We start by defining the namespace ‹tableaux.bag› and importing two familiar packages.

```
(ns dyntab.bag
  (:require [clojure.core.reducers :as r]
            [dyntab.util :as u]))
```

Next, we define the ‘interface’ ‹ITupleBag›.

```
(defprotocol ITupleBag
  (post [this coll])
  (query [this k])
  (newest [this k])
  (mark [this marker])
  (since [this marker])
  (process [this marker rules]))
```

In a sense, what we say is that to be a tuple bag is to implement the following functions:

- Given a tuple bag and a collection of 'messages', ‹post› sends every one of these messages to a 'indexer' (internal to the tuple bag). This indexer assigns zero or more keys to the message. The updated tuple bag is returned.

- Given a tuple bag and a key, ‹query› returns the set of messages associated with that key.

- Given a tuple bag and a key, ‹newest› looks up the last time a ‹post› invocation assigned one or more messages to that key. It returns the set of those messages (or the empty set if no such messages were posted).

- Given a tuple bag and a keyword, ‹mark› returns a new tuple bag. The keyword serves as an annotation in the history of posts.

- Given a tuple bag and a keyword, ‹since› returns a multi-map of those messages—and the keys they were assigned to—added since the tuple bag was marked with the given keyword. If a message is posted twice, only the first time it was posted is taken into account.

- Given a tuple bag, a keyword, and a collection of rules, the function ‹process› looks up what messages were posted ‹since› the tuple bag was marked with the given keyword. These messages (and their keys) are processed by the given rules.

  Rules are functions. When called with no arguments, a rule return a key indicating that it can process messages that are associated with that key.

‹process› calls these rules with two arguments—viz. the tuple-bag (after marking it) and a new message. Rules return foldable collections.

‹process› returns a list of the following elements: (i) the tuple bag marked with the given keyword and (ii) the union of the foldable collections returned by the rules.

These are the functions through which we interact with tuple bags. We explain what indexers and rules are further down.

First we define the data type ‹TupleBag›. Here we specify how we will, as a matter of fact, store tuple bags in memory. We also define a function for creating tuple bags with a given collection of indexers but no messages initially.

```
(defrecord TupleBag [indexers everything newest history])

(defn tuple-bag [indexers]
  (TupleBag. indexers {} {} (list)))
```

Notice that our tuple bag initially contains the following elements or ‘fields’:

- ‹indexers› A foldable collection of indexers. Each indexer is a function that takes a message as an argument and returns a collection of keys.
- ‹everyting› is a multi-map that maps keys (as returned by the indexers) to sets of messages. It's monotone in the sense that posting messages to the tuple bag only ever increases this multi-map.
- ‹newest› is also a multi-map of keys to sets of messages. It maps every key ‹k› to the set of messages assigned to ‹k› the last time ‹post› assigned a message to ‹k›.
- ‹history› is a list of two kind of arguments. The first kind of elements are multi-maps, where each multi-map maps keys to sets of messages. The function ‹post› prepends a multi-map to this list everytime it is invoked. This multi-map contains the key-value pairs that were newly added to

‹everything›. The second elements are keywords, which serve as markers in the history stream. This keywords are added by the function ‹mark›.

Our implementation is as follows.

```
(extend-type TupleBag
  ITupleBag
  (post [this coll]
        (let [aggr-result (->> (.indexers this)
                               (u/rjuxt (constantly coll))
                               (r/mapcat
                                 (fn [[f v]]
                                   (r/map (fn [k] [k v])
                                          (f v))))
                               (r/fold u/mmap-merge
                                       u/mmap-conj))]
          (TupleBag. (.indexers this)
                     (u/mmap-merge (.everything this)
                                   aggr-result)
                     (u/map-merge-overwrite (.newest this)
                                            aggr-result)
                     (cons (u/mmap-diff aggr-result
                                        (.everything this))
                           (.history this)))))
  (query [this k] (u/mmap-get (.everything this) k))
  (newest [this k] (u/mmap-get-unique (.newest this) k))
  (mark [this marker]
        (assert (keyword? marker))
        (TupleBag. (.indexers this)
                   (.everything this)
                   (.newest this)
                   (cons marker (.history this))))
  (since [this marker]
         (->> (.history this)
              (r/take-while #(not= % marker))
              (r/filter coll?)
              (r/fold u/mmap-merge)))
  (process [this marker rules]
               (let [new-bags (since this marker)]
                 [(mark this marker)
              (->> rules
                   (u/rjuxt #(get new-bags (%)))
                   (r/mapcat (fn [[f v]] (f this v)))
                   (u/foldset))])))
```

The function ‹assert› causes the program to terminate abruptly if passed ‹false› or ‹nil›. In ‹mark› we used it to make sure ‹marker› is a keyword because passing a collection here would be confusing.

The function ‹take-while› from ‹clojure.core.reducers› does what its name suggests. Here we use it to keep only the first items in the tuple-bag history; we ignore everything starting at ‹marker›.

Notice that the functions that start with a period, access the members of ‹TupleBag› instances directly. The function ‹TupleBag.› (notice the trailing dot) instantiates such instances. It takes four arguments, corresponding to the desired values for the fields of the tuple bag.

Our next task is to define the indexers that we need. But first we define a little utility function. It works like ‹nth›, but returns ‹nil› instead of raising an exception when the requested element could not be found, even when ‹nth› is applied to a non-sequential object.

```
(defn nth-or-nil [obj n]
  (try
    (nth obj n)
    (catch Exception x nil)))
```

There's one indexer that indexes all messages (assuming they are indeed tuples). It files them under the key ‹[:by-arity x]›, with ‹x› the number of elements in the message.

```
(defn index-by-arity [x]
  [[:by-arity (count x)]])
```

All other indexers either only index pairs or only index triples. We first discuss the ones that index pairs.

```
(defn index-pairs-by-first [x]
  (if (= 2 (count x))
    [[:pairs-by-first (first x)]]
    nil))

(defn index-pairs-by-second [x]
  (if (= 2 (count x))
    [[:pairs-by-second (second x)]]
    nil))

(defn index-pairs-by-fsecond [x]
  (if (= 2 (count x))
    [[:pairs-by-fsecond (nth-or-nil (second x) 0)]]
    nil))

(defn index-pairs-by-first-fsecond-ssecond [x]
  (if (= 2 (count x))
    (let [form (second x)]
      [[:pairs-by-first-fsecond-ssecond
        (first x)
        (nth-or-nil form 0)
        (nth-or-nil form 1)]])
    nil))

(defn index-pairs-by-fsecond-fssecond [x]
  (if (= 2 (count x))
    (let [outer-form (second x)
          inner-form (nth-or-nil outer-form 1)]
      [[:pairs-by-fsecond-fssecond
```

```
          (nth-or-nil outer-form 0)
          (nth-or-nil inner-form 0)]])
    nil))
```

The first of the above indexers indexes pairs by their first element. The other indexers index by the second element of the pair. Thus the indexer ‹index-pairs-by-second› indexes pairs by the entirety of their second element. The next function indexes by the result of applying ‹first› to the second element of the pair. The function ‹index-pairs-by-first-fsecond-ssecond› indexes pairs ‹x› by their first element, and by ‹(first (second x))›, or ‹nil› if no such element exists, and ‹(second (second x))› (or ‹nil›). The fifth and final pair indexer, ‹index-pairs-by-fsecond-fssecond›, indexes pairs ‹x› by ‹(first (second x))› (or ‹nil›) and ‹(first (second (second x)))› (or ‹nil›). Finally, we list the indexers for triples. They are straightforward so we don't discuss them any further.

```
(defn index-triples-by-first-second [x]
  (if (= 3 (count x))
    [[:triples-by-first-second (first x) (second x)]]
    nil))

(defn index-triples-by-first-third [x]
  (if (= 3 (count x))
    [[:triples-by-first-third (first x) (nth x 2)]]
    nil))

(defn index-triples-by-second [x]
  (if (= 3 (count x))
    [[:triples-by-second (second x)]]
    nil))

(defn index-triples-by-third [x]
  (if (= 3 (count x))
    [[:triples-by-third (nth x 2)]]
    nil))
```

## 7.5 Tableau system

We have everything we need to implement the tableau system for PAL. First we set up a new namespace and then we define the tableau rules.

```
(ns dyntab.tableau
  (:require [clojure.core.reducers :as r]
            [clojure.set :as set]
            [dyntab.bag :as bag]
            [dyntab.util :as u]
            [dyntab.syntax :as syntax]))
```

There are two kinds of rules that we need to take into account: deterministic rules and nondeterministic rules. Deterministic rules present us with a collection of tuples that have to be added to the tableau. Nondeterministic rules, on the other hand, give us a collection of collections of tuples. From each of these collections of tuples we are expected to choose one tuple. Of course, in practice we want to try all of our options anyway, but only as branches that we investigate independently.

Deterministic rules are easy. We can process them in parallel and simply merge the collection of formulas that they return. We then ‹post› the merged collection.

Let us first look at the deterministic rules. Recall that when a rule is called with no arguments it returns the key that should trigger the rule. Whenever a message is associated with said key, the rule is called with the tuple bag (after marking it) and the message for arguments.

```
(defn rule-not-not
  ([] [:pairs-by-fsecond-fssecond :not :not])
  ([tb [n
        [not1 [not2 form]]]]
   [[n form]]))

(defn rule-and
  ([] [:pairs-by-fsecond :and])
  ([tb [n
        [and1 form1 form2]]]
   [[n form1]
    [n form2]]))

(defn rule-box1
  ([] [:pairs-by-fsecond :box])
  ([tb [n
        [box1 idx form]]]
   (->> (bag/query tb [:triples-by-first-second idx n])
        (r/map (fn [m] [(nth m 2) form])))))

(defn rule-box2
  ([] [:by-arity 3])
  ([tb [idx src dest]]
   (->> (bag/query
          tb
          [:pairs-by-first-fsecond-ssecond
           src :box idx])
        (r/map (fn [m] [dest (nth (second m) 2)])))))

(defn rule-not-box
  ([] [:pairs-by-fsecond-fssecond :not :box])
  ([tb [n
        [not1 [box1 idx form]]]]
   (if (->> (bag/query tb [:triples-by-first-second idx n])
            (r/filter (fn [[idx2 src2 dest2]]
                        (get (bag/query tb [:by-arity 2])
                             [dest2 [:not form]])))
            (u/fold-empty?))
     (let [m (gensym "node")]
       [[m]
        [idx n m]
```

```
          [m [:not form]]])
        [])))

(defn rule-T
  ([] [:by-arity 1])
  ([tb [n]] (->> syntax/Ind
                 (r/map (fn [x] [x n n])))))

(defn rule-B
  ([] [:by-arity 3])
  ([tb [idx src dest]] [[idx dest src]]))

(defn rule-4
  ([] [:by-arity 3])
  ([tb [idx src dest]]
   (->> [(r/map (fn [[idx2 src2 dest2]] [idx src dest2])
                (bag/query
                  tb
                  [:triples-by-first-second idx dest]))
         (r/map (fn [[idx2 src2 dest2]] [idx src2 dest])
                (bag/query
                  tb
                  [:triples-by-first-third idx src]))]
        (r/mapcat identity))))
```

Notice that there are two rules for (non-negated) box formulas. One is triggered by the addition of a box formula and the other rule is triggered by the addition of edges.

We use the function ‹gensym› to create previously unused names for new nodes. Predictably, ‹identity› is a function that takes one argument and returns said argument unmodified.

As explained above, nondeterministic rules return collections of collections of tuples. To aid readability we let the names of the nondeterministic rules end in an asterisk.

```
(defn rule-not-and*
  ([] [:pairs-by-fsecond-fssecond :not :and])
  ([tb [n
        [not1 [and1 form1 form2]]]]
   [[[n [:not form1]]
     [n [:not form2]]]]))

(defn rule-precond1*
  ([] [:pairs-by-fsecond :!])
  ([tb [n
        [bang1 ann-form post-form]]]
   (->> (bag/query tb [:by-arity 1])
        (r/map (fn [[n]]
                 [[n ann-form] [n [:not ann-form]]]))))))

(defn rule-precond2*
  ([] [:pairs-by-fsecond-fssecond :not :!])
  ([tb [n
        [not1 [bang1 ann-form post-form]]]]
   (->> (bag/query tb [:by-arity 1])
        (r/map (fn [[n]]
                 [[n ann-form] [n [:not ann-form]]]))))))

(defn rule-precond3*
  ([] [:by-arity 1])
  ([tb [n]]
   (->> (bag/query tb [:pairs-by-fsecond :!])
        (r/map (fn [[x [ann1 ann-form post-form]]]
                 [[n ann-form] [n [:not ann-form]]]))))))

(defn rule-precond4*
  ([] [:by-arity 1])
  ([tb [n]]
   (->> (bag/query
             tb
             [:pairs-by-fsecond-fssecond :not :!])
        (r/map (fn [[x [not1 [ann1 ann-form post-form]]]]
                 [[n ann-form] [n [:not ann-form]]]))))))
```

There are four precondition rules. The first two rules add ‹ann−form› or ‹[:not ann−form]› to every node when a formula ‹[ :! ann−form post−form]› or ‹[:not [ :! ann−form post−form]]› is added to any node. The other two rules are activated when a node is added. They look up all dynamic formulas ‹[ :! ann−form post−form]› (and negations thereof) from any node and add ‹ann−form› or ‹[:not ann−form]› to the current node.

The function ‹post› that we discussed earlier takes as arguments a tuple bag and a collection of messages (tuples). It returns an updated tuple bag. For nondeterministic rules we need slightly different behavior. We start from a collection of collections of tuples. From each of these collections of tuples we take one tuple and we put the chosen tuples in a new collection. This gives us $n$ collections of tuples. We independently post these collections to the tuple bag and thus end up with $n$ tuple bags. Each representing a different branch.

```
(defn disjunctive-post [tb coll-of-colls]
  (r/map (partial bag/post tb)
         (u/one-from-each coll-of-colls)))
```

We now define a function for constructing a new tableau. Given a node and a formula, ‹tableau› constructs a new tuple bag with all the indexes required by the rules we discussed earlier and posts the node and formula. If ‹tableau› is given only a formula then the node ‹:start−node› is used.

```
(defn tableau
  ([form] (tableau :start-node form))
  ([node form]
   (if (not (syntax/wff? form))
     (throw (IllegalArgumentException.
              (str "Not a wff: " form))))
```

```
    (-> (bag/tuple-bag
              [bag/index-by-arity
               bag/index-pairs-by-second
               bag/index-pairs-by-fsecond
               bag/index-pairs-by-first-fsecond-ssecond
               bag/index-pairs-by-fsecond-fssecond
               bag/index-triples-by-first-second
               bag/index-triples-by-first-third
               bag/index-triples-by-second
               bag/index-triples-by-third])
        (bag/post [[node] [node form]]))))
```

Given a tableau, the following operation returns the limit of applying all the tableau rules.

```
(defn saturate [tableaux]
  (letfn [(phase1 [inner-tabs]
            (->> inner-tabs
                 (r/map #(bag/process %
                                      :tab-sat-mark
                                      [rule-not-not
                                       rule-and
                                       rule-box1
                                       rule-box2
                                       rule-T
                                       rule-B
                                       rule-4]))
                 (r/map (partial apply bag/post))
                 (r/foldcat)))
          (phase2 [inner-tabs]
            (->> inner-tabs
                 (r/map #(bag/process %
                                      :tab-sat-mark2
                                      [rule-not-box]))
                 (r/map (partial apply bag/post))
                 (r/map #(bag/process %
                                      :tab-sat-mark3
                                      [rule-not-and*
                                       rule-precond1*
                                       rule-precond2*
                                       rule-precond3*
                                       rule-precond4*]))
                 (r/mapcat (partial apply disjunctive-post))
                 (r/foldcat))))]
    (loop [tableaux tableaux
           cur-phase phase1
           changed false]
      (let [new-tableaux (cur-phase tableaux)]
        (if-not (->> new-tableaux
                     (r/mapcat #(bag/since % :tab-sat-mark))
                     (u/fold-empty?))
          (recur new-tableaux phase1 true)
          (if (= cur-phase phase1)
            (recur new-tableaux phase2 changed)
            [changed new-tableaux]))))))
```

It is very noticeable that there are two phases to the saturation process. In the first phase all of the deterministic rules except ‹rule-not-box› are applied. It is only when these rules are saturated that the second phase commences. If changes are made to the tableau in the second phase then the process immediately starts over from the first phase. The reason for this is to ensure that, via ‹rule-T›, ‹rule-B›, and ‹rule-4›, for all $a$ the set of $a$-links is an equivalence relation before ‹rule-not-box› is given the opportunity to create new nodes. This ensures that the function terminates.

Finally, we define a function for checking if a tableau has literal contradictions.

```
(defn consistent? [tab]
  (let [not-forms (bag/query
                        tab
                        [:pairs-by-fsecond :not])]
    (->> (bag/query tab [:pairs-by-fsecond nil])
         (r/filter (fn [[n p]] (get not-forms [n [:not p]])))
         (u/fold-empty?))))
```

We now have a tableau system that can be used to check the satisfiability of non-dynamic formulas. To check the satisfiability of public announcements we need tableau cascades.

## 7.6 Tableau cascades

As before we start by configuring the namespace and defining a function for creating tableau cascades.

```
(ns dyntab.cascade
  (:require [clojure.core.reducers :as r]
            [clojure.set :as set]
            [dyntab.util :as u]
            [dyntab.bag :as bag]
            [dyntab.syntax :as syntax]
            [dyntab.tableau :as tab]))

(defn tableau-cascade [form]
  (bag/post (bag/tuple-bag [bag/index-by-arity
                            bag/index-pairs-by-first
                            bag/index-triples-by-first-second
                            bag/index-triples-by-second
                            bag/index-triples-by-third])
            [[:start-tableau]
             [:start-tableau (tab/tableau form)]]))
```

Our tableau structures are monotone in the sense that we only ever add tuples to them. We never remove anything. Tableau cascades, as presented in previous chapters, are not monotone. Their labels (tableaux) are repeatedly replaced by new labels—although the new labels are always supersets of the labels they replace. Below we take a slightly different approach. We will not remove deprecated labels. Instead we simply provide a function that allows us to easily retrieve the most up-to-date label.

```
(defn newest-tableau [casc t]
  (-> (bag/newest casc [:pairs-by-first t])
      (second)))
```

Recall that saturating tableau cascades involves three operations—namely, developing the tableaux therein, priming, and synchronizing.

We start by providing a rule for developing a tableau in a tableau cascade until the tableau is saturated. The new tableau is then posted to the tableau cascade (unless it was already saturated from the start).

```
(defn saturate-tableau
  ([] [:by-arity 2])
  ([casc [t tab]]
   (->> [tab]
        (tab/saturate)
        ((fn [[changed tableaux]]
           (if changed
             (r/map (fn [new-tab] [t new-tab])
                    tableaux))))
        (r/foldcat)
        (vector))))
```

Priming and synchronizing may trigger updates on tableaux within the tableau cascade. Upon encountering a pattern in a first tableau a second tableau may need to be updated and vice versa.

To detect patterns in a tableau we apply rules to it. These rules return a collection of multi-maps. These multi-maps are interpreted as follows:

- The set associated with the key ‹nil› is a collection of messages to post to the tableau cascade.

- Any key that is not ‹nil› is assumed to be a tableau cascade node ‹t›. The set associated with this key is a collection of messages to post to ‹(newest−tableau casc t)›. The updated tableau becomes the new label for ‹t›.

The following function takes a tableau and a collection of rules as arguments. It applies the rules and merges the vectors of multi-maps that are returned.

```
(defn meta−process [tab rules]
  [(r/fold
     u/mmap−merge
     (second (bag/process tab :meta−post−mark rules)))])
```

Given a tableau cascades and a vector of multi-maps the following function posts the messages in the multi-map and returns an updated tableau cascade.

```
(defn meta−post [casc tables]
  (−>> (apply u/mmap−merge tables)
       (r/mapcat (fn [t coll]
                   (if t
                     [[t (−> (newest−tableau casc t)
                             (bag/mark :meta−post−mark)
                             (bag/post coll))]]
                     coll)))
       (r/foldcat)
       (bag/post casc)))
```

Notice that when a tableau is updated it is marked with the keyword ‹:meta−post−mark›. This marker is also used in the function ‹meta−process› above.

We use the above conventions in our priming operation. The rule ‹prime› is triggered when a tableau is changed in a tableau cascade. When triggered by ‹[t tab]› it invokes ‹meta−process› on the tableau. The first two rules given to ‹meta−process› ensure that if a tableau node contains the formulas ‹[ :! form1 form2]› and ‹form1› then a new tableau is created. Specifically, the first rule is triggered by the presence of formulas ‹[ :! form1 form2]› and then checks if ‹form1› is present in the tableau node; the second rule is triggered by the addition of any formula ‹form1› and checks if ‹[ :! form1 form2]› is also present. The third rule is triggered by negated public announcements. It adds the precondition to the current tableau node and adds a new tableau to the tableau cascade.

```
(defn prime
  ([] [:by-arity 2])
  ([casc [t tab]]
   (meta-process
     tab
     [(fn
         ([] [:pairs-by-fsecond :! ])
         ([cur-tab [n [op form1 form2] :as node-label]]
          (let [new-t (gensym "tableau")]
            (if (-> (bag/query cur-tab [:by-arity 2])
                    (contains? [n form1]))
              [{nil #{[new-t]
                      [new-t (tab/tableau n form2)]
                      [form1 t new-t]}
               t #{node-label}}]
              []))))
      (fn
         ([] [:by-arity 2])
         ([cur-tab [n form :as node-label]]
          [{nil (->> (bag/query
                      cur-tab
                      [:pairs-by-first-fsecond-fssecond
                       n :! form])
                    (r/mapcat
                      (fn [[n2 [op form2 form3]]]
                        (let [new-t (gensym "tableau")]
                          [[new-t]
```

```
                           [new-t (tab/tableau n form3)]
                           [form t new-t]])))
                    (u/foldset))
           t #{node-label}}]))
      (fn
        ([] [:pairs-by-fsecond-fssecond :not :!])
        ([cur-tab [n [op1 [op2 form1 form2]] :as node-label]]
         (let [new-t (gensym "tableau")]
           [{nil #{[new-t]
                   [new-t (tab/tableau n [:not form2])]
                   [form1 t new-t]}
             t #{node-label}}
            {t #{[n form1]}}]))))])))
```

We now turn our attention to synchronization. We start by defining a function that, given a tableau cascade ‹casc› and a tableau cascade node ‹t›, returns a multi-map from tableau nodes ‹n› to the tableau cascade nodes ‹t2› such that (i) ‹(newest-tableau casc t2)› contains ‹n› and (ii) there is an edge in ‹casc› between ‹t› and ‹t2›.

```
(defn synchronization-range [casc t]
  (->> (bag/query casc [:triples-by-second t])
       (r/map #(nth % 2))
       (r/foldcat)
       (r/cat (->> (bag/query casc [:triples-by-third t])
                   (r/map second)
                   (r/foldcat)))
       (r/mapcat (fn [t2]
                   (->> (newest-tableau casc t2)
                        (#(bag/query % [:by-arity 1]))
                        (r/map (fn [[n]] {n #{t2}}))))))
       (r/fold u/mmap-merge)))
```

Definition 4.12 frames the synchronizing operation in set-theoretic terms. It does not give us an algorithm for synchronizing tableaux, but fortunately it's not difficult to create such an algorithm. For any two tableau cascade nodes $t_1$ and $t2$ of a tableau cascade $C$, all we have to do is decide which tuples of $C(t_1?)$ should be added to the label of $t_2$ (and vice versa).

When a new tableau ‹t2› is added we start by copying over all nodes from the tableau ‹t1› that spawned it. The rule for this is triggered by the addition of a tableau cascade edge, which is only ever added when a new tableau is created.

```
(defn synchronize-init
  ([] [:by-arity 3])
  ([casc [precond t1 t2]]
   [(r/fold u/mmap-merge
            [(->> (bag/query (newest-tableau casc t1)
                             [:pairs-by-second precond])
                  (r/map (fn [[n form]] {t2 #{[n]}}))
                  (r/fold u/mmap-merge))])]))
```

Other triggers for copying tableau nodes are the result of changes within ‹t1› or ‹t2›. Every node that is added to ‹t2› is added to ‹t1›. If the formula ‹form› is added to a node in ‹t1› then this node is copied to ‹t2›. The rule ‹synchronize› takes care of these conditions. It also deals with atoms and edges.

```
(defn synchronize
  ([] [:by-arity 2])
  ([casc [t tab]]
   (let [sync-range (synchronization-range casc t)]
     (meta-process
       tab
       [(fn ; copy node to left-hand tableaux
           ([] [:by-arity 1])
           ([cur-tab [n]]
            (->> (bag/query casc [:triples-by-third t])
                 (r/map (fn [[precond t1 t2]]
                            {t1 #{[n]
```

```
                              [n precond]}})))))
        (fn ; copy node to right-hand tableaux
          ([] [:by-arity 2])
          ([cur-tab [n form]]
           (->> (bag/query casc
                           [:triples-by-first-second
                            form t])
                (r/map (fn [[precond t1 t2]]
                             {t2 #{[n]}}))))))
        (fn ; copy atoms to this tableau
          ([] [:by-arity 1])
          ([cur-tab [n]]
           (->> (get sync-range n)
                (r/map #(bag/newest casc
                                    [:pairs-by-first %]))
                (r/mapcat
                  #(bag/query
                     (second %)
                     [:pairs-by-fsecond nil]))
                (r/filter #(= (first %) n))
                (r/map (fn [x] {t #{x}})))))
        (fn ; copy atoms to neighboring tableaux
          ([] [:pairs-by-fsecond nil])
          ([cur-tab [n atom]]
           (r/map (fn [t2] {t2 #{[n atom]}})
                  (get sync-range n))))
        (fn ; copy incoming edges to this tableau
          ([] [:by-arity 1])
          ([cur-tab [n]]
           (->> (get sync-range n)
                (r/map #(bag/newest casc
                                    [:pairs-by-first %]))
                (r/mapcat #(bag/query (second %)
                                        [:triples-by-third n]))
                 (r/filter
                   #(contains? (bag/query cur-tab
                                            [:by-arity 1])
                                [(second %)]))
                 (r/map (fn [x] {t #{x}})))))
        (fn ; copy outgoing edges to this tableau
          ([] [:by-arity 1])
          ([cur-tab [n]]
           (->> (get sync-range n)
                 (r/map #(bag/newest casc
                                      [:pairs-by-first %]))
                 (r/mapcat
                   #(bag/query (second %)
                                [:triples-by-second n]))
                 (r/filter
                   #(contains? (bag/query cur-tab
                                            [:by-arity 1])
                                [(nth % 2)]))
                 (r/map (fn [x] {t #{x}})))))
        (fn ; copy edges to neighboring tableaux
          ([] [:by-arity 3])
          ([cur-tab [idx n m]]
           (r/map (fn [t2] {t2 #{[idx n m]}})
                   (set/intersection
                     (get sync-range n)
                     (get sync-range m))))])))
```

Whenever a node is added to ‹t1› or ‹t2› the atoms of the other tableau must be copied over. Similarly, when an atom is added to ‹t1› or ‹t2› then this same atom must be added to the other tableau. Copying edges follows similar reasoning.

We can now define the algorithm for saturating tableau cascades. The following function develops, primes, and synchronizes a given collection of tableau cascades. It returns the limit of all branches resulting from these operations.

```
(defn saturate [cascades]
  (let [next-casc
         (->> cascades
              (r/map #(bag/process
                        %
                        :casc-sat-mark
                        [saturate-tableau]))
              (r/mapcat (partial apply tab/disjunctive-post))
              (r/map #(bag/process
                        %
                        :casc-sat-mark2
                        [prime
                         synchronize
                         synchronize-init]))
              (r/map (partial apply meta-post))
              (r/fold 8 (r/monoid r/cat vector) conj))]
    (if-not (->> next-casc
                 (r/mapcat #(bag/since % :casc-sat-mark))
                 (u/fold-empty?))
      (recur next-casc)
      next-casc)))
```

The fragment ‹(r/fold 8 (r/monoid r/cat vector) conj)› behaves similarly to ‹r/foldcat›. However, ‹r/foldcat› only returns a foldable collection if the initial collection is foldable and large enough to make it worthwhile to process in parallel. Our replacement code always returns a foldable collection. Moreover, it does parallel processing in batches of 8 tableau cascades. This is more sensible for our purposes than using batches of 512 items—as is the default for ‹r/fold›.

With the system that we developed in this chapter we can check whether a formula of $\mathcal{L}^{\Box!}$ is satisfiable. We start by constructing an initial tableau cascade and applying ‹saturate›. The formula is satisfiable if and only if this yields us a tableau cascade in which all tableaux are consistent.

```
(defn consistent? [casc]
  (->> (bag/query casc [:by-arity 1])
       (r/map #(bag/newest casc [:pairs-by-first (first %)]))
       (r/filter #(not (tab/consistent? (second %))))
       (u/fold-empty?)))

(defn satisfiable? [form]
  (->> (saturate [(tableau-cascade form)])
       (r/filter consistent?)
       (u/fold-empty?)
       (not)))

(defn valid? [form]
  (not (satisfiable? [:not form])))
```

The source code that was presented in this chapter is also available online at https://github.com/jdevuyst/tableaux.

## 7.7 Remarks

We set out to create a proof of concept theorem prover for public announcement logic. On this we believe to have delivered.

We do have some comments with respect to possible expansions on our codebase. First, we want to make a few retrospective comments on the overall design of our theorem prover. Afterwards we would like to make some comments on its performance in practice.

We are satisfied with the overall structure of our code. Nevertheless, the ‹TupleBag› data structure could use some rethinking. The system of installing ad hoc indexers provides good performance and is easy to implement. However, it comes at a cost. That cost is the complexity of the rules. A better tuple bag would allow us to rewrite ‹rule−box1› and ‹rule−box2› into one rule. Informally this rule would be as follows: Given two tuples ‹[n [:box idx form]]› and ‹[idx n m]›, add ‹[m form]›.

The RETE algorithm provides functionality as described above [18]. We know of two RETE implementations for Clojure, Mímir [32] and Clara [8].

RETE relies on heuristics to provide good performance. Because our application was expected to be performance sensitive from the start we decided not to use RETE at this point. Whether or not these performance concerns are justified is something we leave to future research. Indeed, our present tuple bag implementation provides a useful performance baseline for possible future efforts to refactor ‹TupleBag›.

If RETE, or any other forward chaining system, were to be added in a future update then ideally it should be implemented in a such a way that it can also match inter-tableau patterns. For instance, it should be possible to have a tableau cascade rule such that, given a tableau cascade edge ‹[form t1 t2]› in ‹casc›, an edge ‹[idx n m]› in ‹(newest−tableau casc t1)›, and two singles ‹[n]› and ‹[m]› in ‹(newest−tableau casc t2)› the edge ‹[idx n m]› is added to the tableau for ‹t2›. Currently ‹synchronize› contains three rules for this!

Given a forward chaining system it would also be worth considering to implement a domain specific language for specifying rules. This would make it easier to extend our software. Knowledge of Clojure would no longer be required, thereby significantly reducing the barrier to entry.

Finally, we want to make some remarks about performance. We used the Oracle Java 7 virtual machine. The first thing we noticed was that for complex formulas we had to manually increase the memory available to the JVM. We used the command line option ‹−Xmxn› to increase the available memory to 6 GB (out of 8 GB total memory). Still, we found that for sufficiently complex formulas our theorem prover ran out of memory after a few minutes. Specifically, the garbage collector would start ‘trashing’. This means that most of the time spent executing the program was spent on garbage collection. Each garbage collection cycle would take tens of seconds and a new ‘full’ cycle was started right after the previous one ended. We did attempt to use the incremental garbage collection (available using the option ‹−Xincgc›) but this did not help.

Because the ‘full’ garbage collection cycles were started one after another we conjecture that the problem is that not enough memory was available. The JVM has a generational garbage collector, meaning that it tries to recycle recently allocated memory before searching for unreferenced parts in old data. Hence, had the problem been that the garbage collector could not handle the intermediate data generated then we would not expect to see so many full cycles being performed.

The above results suggest that tuple bags should be refactored to use less memory or to offload parts of their content to disk. On the other hand, it might be the case that a linear decrease of memory used by ‹TupleBag› would be immediately defeated due to the theoretic space complexity of our algorithm. Analysis of the space complexity of our program is left for further reasearch. Furthermore, refactoring ‹TupleBag› to use less memory might involve performance tradeoffs that create new bottlenecks.

We ran our theorem prover on a laptop with an 1.7 GHz Intel i7 Haswell processor. This processor has four cores. We found that we could easily attain CPU utilization of 370% and more by processing tableau cascades in batches of around 8 items at a time. However, we also found that this decreased the execution time by slightly less than 50%. More research is needed to determine the cause of this somewhat disappointing speed up.

## 7.8 Related work

LoTREC is the only software program that we know of that contains a tableau-based theorem prover for public announcement logic. LoTREC is an extensible system that comes with a domain specific language and a graphical user interface. Evidently our Clojure program does not have the same level of maturity. Nevertheless, the theoretic foundations of our program are simple enough that, at this point, the lack of a domain specific language for tableau rules (and tableau cascade rules) does not strike us as a major shortcoming. Additionally, the graphical user interface of LoTREC is rather complicated.

# 8
# Conclusion

We set out to create proof systems for dynamic modal logics that are conceptually simple, easy to use, modular, and extensible. Let's review these items one by one and evaluate if we succeeded.

- Conceptual simplicity. From the get go we modeled our tableau systems closely after the semantics they target. We believe this has resulted in transparent proof systems.

- Ease of use. Because the rules of the tableau systems are based on the semantics of the different dynamic languages, they are easy to understand and apply.

  Determining whether two tableau stand in relations $\xrightarrow{!\phi}$, $\xrightarrow{\otimes\phi}$, $\xrightarrow{¡\phi}$, $\xrightarrow{\sharp\phi}$, or $\xrightarrow{\Uparrow\phi}$ is no more difficult than computing the outcome of an operation $|_{!\phi}$, $|_{\otimes\phi}$, $|_{¡\phi}$, $|_{\sharp\phi}$, or $|_{\Uparrow\phi}$.

  The workings of tableau cascades and the operations of priming and synchronizing are fairly straightforward. We reckon that the operation of $\otimes$-correcting is somewhat inelegant, but that the benefit of having the tableau rules be independent of tableau cascades is worth the cost.

  Finally, folding is not the easiest concept to understand but (i) it is only required for certain combinations of frame conditions and (ii) we feel it is preferable to have a folding operation than to include frame condition properties in the rule $\mathbf{R}_{\Box}$ (as is common in the literature).

- Modularity. For every truth schema in the different languages we have at most two tableau rules. Every dynamic operator also has two corresponding clauses that constrain what it means for a tableau to be 'open'. Finally, these clauses depend on a relation between tableaux that is closely related to the dynamic operation on models. Consequently, our tableau systems are highly modular.

  The way the relations $\xrightarrow{!\phi}$, $\xrightarrow{\otimes\phi}$, $\xrightarrow{¡\phi}$, $\xrightarrow{\sharp\phi}$, and $\xrightarrow{\Uparrow\phi}$ are related to the operations $|_{!\phi}$, $|_{\otimes\phi}$, $|_{¡\phi}$, $|_{\sharp\phi}$, and $|_{\Uparrow\phi}$ is also modular in the sense that lemmas 4.1, 5.1, and 6.1 were established before the soundness and completeness proofs.

  Similarly, we would tout it as a benefit with respect to modularity that the operations of folding, priming, and synchronizing are not needed for the soundness and completeness proofs. Indeed, we only introduced tableau cascades after proving soundness and completeness.

  Our tableau cascades themselves are also modular. Most dynamic operators require only that a priming operation is defined and that the definition of satisfaction of tableau cascades is amended.

- Extensibility. Throughout this dissertation we have to a large extent been able to add constructs to our languages without having to revisit earlier results. To wit, we did not have to create a proof system from scratch for every logic and it would be straightforward to create a proof system for the language that would result from combining all dynamic operators in one language.

To summarize, we hold that we have succeeded in creating an alternative proof theory for several popular dynamic modal logics that has pedagogic and esthetic advantages.

In section 6.1 we encountered one belief revision operator that we did not (yet) manage to translate to our style of tableau systems. This is a notable limitation. It's presently not known what changes would need to be made to our general approach to make it compatible with 'conservative upgrades'.

For future research we would be interested in applying the techniques discussed in this volume to more expressive logics. For instance, presently no tableau systems appear to exist for dynamic epistemic logics with common knowledge [22]. Beyond this, the minimal revision logic in [13] has dynamic operators that might fit in well with our approach to tableau systems.

Our Clojure implementation demonstrates that our tableau system for public announcement logic can readily be turned into an automatic theorem prover. A synchronization algorithm has to be devised but this is an easy enough task. No complicated new algorithms are required. For future research we think the primary focus should be on adding a forward chaining engine and defining a domain specific language for defining tableau rules and tableau cascade rules. Afterwards, adding support for $\mathcal{L}^{\Box\otimes}$ and $\mathcal{L}^{U\Box¡\sharp\Uparrow}$ should require only a fairly straightforward translation of the results from chapters 5 and 6.

# Bibliography

[1] C.E. Alchourrón, P. Gärdenfors, and D. Makinson. On the logic of theory change: Partial meet contraction and revision functions. *Journal of Symbolic Logic*, 50:510–530, 1985.

[2] P. Balbiani, H.P. van Ditmarsch, A. Herzig, and T. de Lima. Tableaux for public announcement logic. *Journal of Logic and Computation*, 20:55–76, 2010.

[3] A. Baltag and L.S. Moss. Logics for epistemic programs. *Synthese*, 139:165–224, 2004.

[4] A. Baltag, L.S. Moss, and S. Solecki. The logic of public announcements, common knowledge, and private suspicions. Technical Report SEN-R9922, University of Amsterdam, 1999.

[5] A. Baltag and S. Smets. Conditional doxastic models: A qualitative approach to dynamic belief revision. In G. Mints and R. de Queiroz, editors, *Proceedings of WOLLIC 2006*, volume 165 of *Electronic Notes in Theoretical Computer Science*, pages 5–21, 2006.

[6] A. Baltag and S. Smets. A qualitative theory of dynamic interactive belief revision. In *Logic and the Foundations of Game and Decision Theory (LOFT 7)*, Texts in Logic and Games 3, pages 13–60. Amsterdam University Press, 2008.

[7] P. Blackburn, M. de Rijke, and Y. Venema. *Modal Logic*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2002.

[8] R. Brush. Clara: Forward-chaining rules in clojure. https://github.com/rbrush/clara-rules, 2013.

[9] M. Castilho, L. Fariñas del Cerro, O. Gasquet, and A. Herzig. Modal tableaux with propagation rules and structural rules. *Fundamenta Informaticae*, 32(3/4):281–297, 1997.

[10] M. D'Agostino, Dov M. Gabbay, H. Reiner, and J. Posegga, editors. *Handbook of Tableau Methods*. Springer, 1999.

[11] R. Davies and F. Pfenning. A modal analysis of staged computation. *Journal of the ACM*, pages 258–270, 1996.

[12] M.S. de Boer. KE tableaux for public announcement logic, 2007.

[13] J. De Vuyst. Minimal revision and classical Kripke models: First results. In H.P. van Ditmarsch, J. Lang, and S. Ju, editors, *LORI 2011*, volume 6953 of *Lecture Notes in Artificial Intelligence*, pages 300–313. Springer, 2011.

[14] L. Fariñas del Cerro, D. Fauthoux, O. Gasquet, A. Herzig, D. Longin, and F. Massacci. Lotrec: The generic tableau prover for modal and description logics. In *Proceedings of the First International Joint Conference on Automated Reasoning*, LNCS, pages 453–458. Springer, 2001.

[15] L. Fariñas del Cerro and O. Gasquet. A general framework for pattern-driven modal tableaux. *Logic Journal of the IGPL*, 10(1):51–83, 2002.

[16] M. Fitting. *Proof methods for modal and intuitionistic logics*, volume 169. Springer, 1983.

[17] M. Fogus and C. Houser. *The Joy of Clojure*. Manning Publications, 2nd edition, forthcoming.

[18] C. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19:17–37, 1982.

[19] O. Gasquet, A. Herzig, D. Longin, and M. Sahade. Lotrec: Logical tableaux research engineering companion. In B. Beckert, editor, *Automated Reasoning with Analytic Tableaux and Related Methods*, volume 3702 of *Lecture Notes in Computer Science*, pages 318–322. Springer, 2005.

[20] J.D. Gerbrandy and W. Groeneveld. Reasoning about informatino change. *Journal of Logic, Language, and Information*, 6:147–169, 1997.

[21] R. Goldblatt. Mathematical modal logic: A view of its evolution. *Journal of Applied Logic*, 1(5-6):309–392, 2003.

[22] J.U. Hansen. Terminating tableaux for dynamic epistemic logics. *Electronic Notes in Theoretical Computer Science*, 262:141–156, 2010.

[23] A. Herzig, S. Konieczny, and L. Perrussel. On iterated revision in the AGM framework. In *Symbolic and Quantitative Approaches to Reasoning with Uncertainty*, pages 477–488. Springer, 2003.

[24] J. Hintikka. *Knowledge and belief an introduction to the logic of the two notions*. Contemporary philosophy series. Cornell University Press, 4th edition, 1969.

[25] G.E. Hughes and M.J. Cresswell. *A New Introduction to Modal Logic*. Routledge, 1996.

[26] S.A. Kripke. A completeness theorem in modal logic. *Journal of Symbolic Logic*, 24:1–14, 1959.

[27] S.A. Kripke. Semantical analysis of modal logic I: Normal modal propositional calculi. *Mathematical Logic Quarterly*, 9:67–96, 1963.

[28] F. Liu. *Reasoning about Preference Dynamics*. Springer Library, 2011.

[29] M. McGrath. Propositions. In E.N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Summer 2012 edition, 2012.

[30] J.A. Plaza. Logics of public communications. In *Proceedings of the 4th International Symposium on Methodologies for Intelligent Systems*, pages 201–216, 1989.

[31] G. Primiero. A multi-modal type system and its procedural semantics for safe distributed programming. In *Intuitionistic Modal Logic and Applications Workshop (IMLA11), Nancy*, 2011.

[32] H. Råberg. Mímir: An experimental rule engine written in clojure. https://github.com/hraberg/mimir, 2012-2013.

[33] J. Van Benthem. Dynamic logic for belief revision. *Journal of Applied Non-Classical Logics*, 17(2), 2007.

[34] J. van Benthem. *Logical Dynamics of Information and Interaction*. Cambridge University Press, 2011.

[35] J. Van Benthem and F. Liu. Dynamic logic of preference upgrade. *Journal of Applied Non-Classical Logics*, 14(2), 2004.

[36] H.P. van Ditmarsch. *Knowledge games*. PhD thesis, University of Groningen, 2000.

[37] H.P. van Ditmarsch, W. van der Hoek, and B. Kooi. *Dynamic epistemic logic*, volume 337 of *Synthese Library*. Springer, 2007.

[38] Y. Wang and Q. Cao. On axiomatizations of public announcement logic. *Synthese*, pages 1–32, 2013.

[39] T. Williamson. *Knowledge and Its Limits*. Oxford University Press, 2002.

# Index of Definitions