Satisfiability of Temporal Formulas in Models of Linear Time

Marcus Mathioudakis Supervisor: Ian Hodkinson Second Marker: Philippa Gardner

29th June 2013

Abstract

We consider the problem of determining satisfiability of temporal formulas in general models of linear time. We start by giving a model expression language for describing general models of linear time. We describe a new algorithm for determining satisfiability of temporal formulas of the logics $L(\mathcal{F}, \mathcal{P})$ and $L(\mathcal{U}, \mathcal{S})$ in these models. We provide an implementation of the algorithm, with a sophisticated user interface. Finally we prove that the algorithm operates in polynomial space, implying a new result for the complexity of the satisfiability problem itself.

Acknowledgments

First and foremost, I would like to thank Ian Hodkinson for supervising me. His support throughout the project kept me motivated, and the direction he provided made the project challenging and rewarding in equal measure. I would also like to thank Philippa Gardner for her advice and feedback.

I am immensely grateful to my parents Kostas and Simone for their love and support, and for giving me the opportunity to study at Imperial College. Lastly, I would like to thank my friends Adam, Danny, Justin and Maynard, my discussions with whom opened me up to lateral approaches to the various challenges I faced at university.

Contents

Chapter 1

Introduction

1.1 Motivation

Temporal Logic is an area of logic which is concerned with reasoning about time and change through time. Since its inception by Arthur Prior in the late 1950's , it has found many applications in the field of computer science, ranging from databases to A.I. (see [11] for a survey of such applications). In particular as a result of Amir Pnueli's pioneering paper on "The Temporal Logic of Programs"([6]) in which he introduced a temporal logic for reasoning about the correctness of sequential and concurrent programs, temporal logic has found an important application to the field of formal program verification. The general idea is that of building a temporal model of a program, and constructing a specification of the program's desired behaviour as a set of formulas in some propositional language. The verification of the program is then reduced to the problem of determining that the formulas of the specification are satisfied in the model of the program. This form of temporal-propositional system specification and verification forms a large sub-area of the field of Model Checking, which is concerned with verification of finite state systems.

When choosing a temporal logic for specifying a system's desired behaviour, or for any other purpose, one of the key considerations is the view of time we take. We might for example take a branching view, where a given point in time may have more than one point directly after it, or a linear view. Furthermore we might view time as being set of discrete points, or as having a continuous flow, whereby between any two point in time there are infinitely many more points. Much of the work within the field of temporal logic itself adopts a natural numbers view of time, whereby time has a first point and no endpoint, and every point has exactly one direct successor. Subsequently the aforementioned problem of determining whether a formula is satisfied in a given model (referred to here as satisfiability) has been thoroughly investigated for propositional temporal logics over such views of time. For example the complexity of determining satisfiability of a formula in the logic of Until and Since over a natural numbers view of time has been shown in [8] to be P-SPACE complete. Such logics have also been used for practical applications, a prime example being LTL (Linear Temporal Logic introduced in [6]) which has been extensively used in Model Checking, and for which an EXPTIME[1] procedure for deciding satisfiability was given in [12].

However until recently, the area of temporal logic taking a real-numbers view of time has been left largely unexplored. Thus it is on such logics that we will focus on in this project. Specifically we focus on the problem of determining if a given formula is satisfied in a given model, as it is (in light of the discussion above) of both theoretical and potentially practical interest. To

[1] specifically its runtime is exponential in the length of the formula but linear in the size of the model

the best of our knowledge this problem has previously only been considered by Reynolds et al. in [1].We will consider two temporal languages over such models of time; firstly the logic with connectives $\mathcal{F}$ and $\mathcal{P}$ (referred to as $L(\mathcal{F},\mathcal{P})$), and subsequently that with connectives $\mathcal{U}$ and $\mathcal{S}$ ($L(\mathcal{U},\mathcal{S})$). The first of these is the "original" temporal language, as introduced by Prior. The second is an expressively superior language, which was shown by Kamp([13]) to be expressively complete over $\mathbb{R}$ and $\mathbb{N}$[2]. The formulas of these languages are by definition finite. Thus the first problem we are faced with if we are to effectively compute satisfiability for a given model is that of finitely specifying infinite models. To this end we will introduce a model expression language similar to that introduced in [1] [3] for describing general models of time. The language is sufficiently expressive that for any formula F satisfied in a model based on a real-numbers view of time, F is satisfied in a model described by the language. We proceed to describe an algorithm for determining satisfiability of formulas in said models, first for $L(\mathcal{F},\mathcal{P})$ and then for $L(\mathcal{U},\mathcal{S})$. We provide an implementation of the algorithm, together with an intuitive user interface. Finally we consider the complexity of our algorithm, proving that for a significant sub-problem of the original problem it runs in polynomial time. We then prove that in general it operates in PSPACE, a result which gives us a new classification (in terms of its complexity) of the satisfiability problem for the class of models describable by the model expression language.

1.2 Contributions

The key contributions of the project can be outlined as follows:

- A new algorithm for determining satisfiability of temporal formulas of the logics $L(\mathcal{F},\mathcal{P})$ and $L(\mathcal{U},\mathcal{S})$ in general linear models of time, as described by model expressions.
- An implementation of the algorithm, with a sophisticated user interface. This is to the best of our knowledge the first implementation of an algorithm solving this problem.
- A complexity analysis of our algorithm, implying a new result for the complexity of the problem itself.

1.3 Structure of the Report

- Chapter 2: In this chapter give the background knowledge required to understand the rest of the project. We start by describing the logics $L(\mathcal{F},\mathcal{P})$ and $L(\mathcal{U},\mathcal{S})$, after which we introduce an expression language for describing linear models of time. Finally we briefly give some background relevant to complexity theory.
- Chapter 3: We describe our algorithm for determining satisfiability of a given formula $A \in L(\mathcal{F},\mathcal{P})$ in a model described by a model expression. We then extend the algorithm to $L(\mathcal{U},\mathcal{S})$.
- Chapter 4: Here we discuss our implementation of the algorithm described in chapter 3.
- Chapter 5: In this chapter we analyse the complexity of the algorithm, and subsequently of the problem of determining satisfiability itself.

[2] for every formula $a(t)$ of first order logic, there is a formula F in the logic of $\mathcal{U}$ and $\mathcal{S}$, such that F and $a(t)$ are equivalent over both $\mathbb{R}$ and $\mathbb{N}$

[3] this is quite a natural choice, as the expression language itself originates from the theory of linear orders

- Chapter 6: We summarise and evaluate the results of the project, and discuss the questions it raises, outlining future work.

Chapter 2

Background

In this chapter we present the background knowledge necessary to understand the rest of this report. Thus instead of reading it in full, the reader is free to use it as a reference when reading subsequent chapters.

2.1 The Temporal Logics $L(\mathcal{F}, \mathcal{P})$, and $L(\mathcal{U}, \mathcal{S})$

We start by fixing a countably infinite[1] set $\mathcal{L}$ of propositional atoms. Temporal formulas are defined inductively in terms of these atoms and various operators, the syntax and semantics of which are defined below. Time is modelled using a linear ordering on a set T of points in time.

Definition 2.1. (Linear Ordering): a linear ordering[2] of the set T is a binary relation $<$ on T (we write $t < u$ to mean $(t, u) \in <$) such that for any elements t, u, v in T:

1. $t < u \wedge u < v \rightarrow t < v$
2. if $t \neq u$, then either $t < u$ or $u < t$ but not both
3. $t \not< t$

Linear orderings prove suitable for modeling time as they capture well some of our intuitions about time: if t is in the past of u and u is in the past of v, then t should be in the past of v;every point t in time is either in the future or the past of every other point v ;a time t is not in the past or future of itself. Temporal models can now be defined as follows:

Definition 2.2. (temporal model): A temporal model $\mathcal{M}$ is a triple $(T, <, h)$ where

- T is a non-empty set (whose elements correspond to points in time)
- $<$ is a linear ordering of T
- h is an assignment over T, (i.e. a function $h : \mathcal{L} \rightarrow \wp(T)$[3] mapping each atom in $\mathcal{L}$ to the set of points in T at which we consider that the atom is true)

We will consider the formulas of two logics over such models, namely $L(\mathcal{F}, \mathcal{P})$ and $L(\mathcal{U}, \mathcal{S})$.

[1] intuitively, a countably infinite set is one for which there is a one-to-one correspondence with the natural numbers

[2] Note that when a relation $<$ is a linear ordering of a set T, we will often directly refer to the pair $(T, <)$ as a linear ordering.

[3] where $\wp(T)$ denotes the powerset of T

2.1.1 $L(\mathcal{F}, \mathcal{P})$

Syntax

Definition 2.3. ($L(\mathcal{F}, \mathcal{P})$ formula):The formulas of $L(\mathcal{F}, \mathcal{P})$ are defined as:

- $\top$
- The atoms in $\mathcal{L}$
- if A and B are formulas of $L(\mathcal{F}, \mathcal{P})$, then so are $\neg A$, $A \wedge B$, $\mathcal{F}A$, and $\mathcal{P}A$.

***Abbreviations* :** In addition to the above operators, we will also use the following abbreviations:

- $A \vee B = \neg(\neg A \wedge \neg B)$
- $A \rightarrow B = \neg(A \wedge \neg B)$

***Binding Conventions* :** All unary operators are right associative. All binary operators are left associative. operators are listed below in order of decreasing precedence:

1. $\neg, \mathcal{F}, \mathcal{P}$ (these are unary, so mutual order is immaterial)
2. $\wedge$
3. $\vee$
4. $\rightarrow$

Definition 2.4. (subformulas for $L(\mathcal{F}, \mathcal{P})$): We define the subformulas of a formula $A \in L(\mathcal{F}, \mathcal{P})$ (denoted $subformulas(A)$) inductively on the structure of A:

- if A is an atom $p \in \mathcal{L}$ or $\top$ then A is the only subformula of A.
- if $A = \neg B$, $A = \mathcal{F}B$ or $A = \mathcal{P}B$ then the subformulas of A are the subformulas of B, and A itself.
- if $A = B \wedge C$ then the subformulas of A are the subformulas of B, the subformulas of C, and A itself.

Semantics

We evaluate these formulas at points of temporal models $\mathcal{M} = (T, <, h)$, writing $\mathcal{M}, t \vDash A$ for a formula A in $L(\mathcal{F}, \mathcal{P})$ to mean that A is true at the point t of the model $\mathcal{M}$. Given a model $\mathcal{M} = (T, <, h)$, a world $t \in T$ and formulas A,B we define $\vDash$ as follows:

- $\mathcal{M}, t \vDash \top$
- For any atom $p \in \mathcal{L}$, $\mathcal{M}, t \vDash p \iff t \in h(p)$
- $\mathcal{M}, t \vDash \neg A \iff \mathcal{M}, t \nvDash A$
- $\mathcal{M}, t \vDash A \wedge B \iff \mathcal{M}, t \vDash A \; and \; \mathcal{M}, t \vDash B$
- $\mathcal{M}, t \vDash \mathcal{F}A \iff \exists u.u > t \wedge \mathcal{M}, u \vDash A$
  (the Future operator: A is true at some point in the future)
- $\mathcal{M}, t \vDash \mathcal{P}A \iff \exists u.u < t \wedge \mathcal{M}, u \vDash A$
  (the Past operator: A was true at some point in the past)

2.1.2 $L(\mathcal{U}, \mathcal{S})$

Syntax

Similarly, the formulas of $L(\mathcal{U}, \mathcal{S})$ are defined as:

- $\top$
- The atoms in $\mathcal{L}$
- if A and B are formulas of $L(\mathcal{U}, \mathcal{S})$, then so are $\neg A$, $A \wedge B$, $A\mathcal{U}B$, $A\mathcal{W}B$, $A\mathcal{S}B$ and $A\mathcal{Z}B$.

Abbreviations: In addition to the above operators, we will also use the following abbreviations:

- $A \vee B = \neg(\neg A \wedge \neg B)$
- $A \rightarrow B = \neg(A \wedge \neg B)$
- $\mathcal{G}A = A\mathcal{W}\bot$
  (A will be true at all times in the future)
- $\mathcal{H}A = A\mathcal{Z}\bot$
  (A was true at all times in the past)

Binding Conventions: All unary operators are right associative. All binary operators are left associative. operators are listed below in order of decreasing precedence:

1. $\mathcal{G}, \mathcal{H}$ (these are unary, so mutual order is immaterial)
2. $\mathcal{U}, \mathcal{S}, \mathcal{W}, \mathcal{Z}$ (these all have the same precedence)
3. $\wedge$
4. $\vee$
5. $\rightarrow$

Definition 2.5. (subformulas for $L(\mathcal{U}, \mathcal{S})$): We define the subformulas of a formula $A \in L(\mathcal{U}, \mathcal{S})$ (denoted $subformulas(A)$) inductively on the structure of A:

- if A is an atom $p \in \mathcal{L}$ or $\top$ then A is the only subformula of A.
- if $A = \neg B$ then the subformulas of A are the subformulas of B, and A.
- if $A = B \wedge C$, $A = B\mathcal{U}C$, $A = B\mathcal{W}C$, $A = B\mathcal{S}C$ or $A = B\mathcal{Z}C$ then the subformulas of A are the subformulas of B, the subformulas of C, and A itself.

Semantics

Given a model $\mathcal{M} = (T, <, h)$, a world $t \in T$ and formulas A,B of $L(\mathcal{U}, \mathcal{S})$ we define the relation $\vDash$ as above for atoms, $\top$, and formulas built using $\neg$ and $\wedge$. The semantics for the operators $\mathcal{U}$,$\mathcal{W}$,$\mathcal{S}$,$\mathcal{Z}$ are as follows:

- $\mathcal{M}, t \vDash A\mathcal{U}B \iff \exists u.(u > t \wedge \mathcal{M}, u \vDash B \wedge \forall v.(t < v < u \rightarrow \mathcal{M}, v \vDash A))$
  (the Until operator: A will true until B becomes true)

- $\mathcal{M}, t \vDash A\mathcal{W}B \iff \exists u.(u > t \wedge \mathcal{M}, u \vDash B \wedge \forall v.(t < v < u \rightarrow \mathcal{M}, v \vDash A)) \vee \forall v > t.\mathcal{M}, v \vDash A$
(the Weak Until operator: A will be true until B becomes true, or A will be true forever)
- $\mathcal{M}, t \vDash A\mathcal{S}B \iff \exists u.(u < t \wedge \mathcal{M}, u \vDash B \wedge \forall v.(u < v < t \rightarrow \mathcal{M}, v \vDash A))$
(the Since operator: A has been true since B was true)
- $\mathcal{M}, t \vDash A\mathcal{Z}B \iff \exists u.(u < t \wedge \mathcal{M}, u \vDash B \wedge \forall v.(u < v < t \rightarrow \mathcal{M}, v \vDash A)) \vee \forall v < t.\mathcal{M}, v \vDash A$
(the Weak Since (Zince) operator: A has been true since B was true, or A has always been true)

It is worth noting that we adopt here what are referred to in the literature as the 'strict' versions of the temporal connectives, as opposed to the often used non-strict ones. An example of the non-strict connectives are $\mathcal{F}^{\leq}$ and $\mathcal{U}^{\leq}$:

- $\mathcal{M}, t \vDash \mathcal{F}^{\leq}A \iff \exists u.u \geq t \wedge \mathcal{M}, u \vDash A$
- $\mathcal{M}, t \vDash A\mathcal{U}^{\leq}B \iff \exists u.(u \geq t \wedge \mathcal{M}, u \vDash B \wedge \forall v.(t \leq v < u \rightarrow \mathcal{M}, v \vDash A))$

For technical reasons, non-strict versions work better with model checking using automata, but expressively, the strict versions are superior (See [4] p.103).

Finally, we introduce some definition of terms we will use for both $L(\mathcal{F}, \mathcal{P})$ and $L(\mathcal{U}, \mathcal{S})$

Definition 2.6. (atomic formula): A formula F is atomic if it is a propositional atom $a \in \mathcal{L}$, or $\top$.

Definition 2.7. (boolean formula): A formula F is boolean if it is atomic, or if it is of the form $A \wedge B$, or $\neg A$.

Relationship between $L(\mathcal{F}, \mathcal{P})$ and $L(\mathcal{U}, \mathcal{S})$

$L(\mathcal{F}, \mathcal{P})$ is in fact a subset of $L(\mathcal{U}, \mathcal{S})$, as $\mathcal{F}A$ and $\mathcal{P}A$ are expressible in $L(\mathcal{U}, \mathcal{S})$ as $\top\mathcal{U}A$ and $\top\mathcal{S}A$ respectively, so any results we prove in the subsequent sections for $L(\mathcal{U}, \mathcal{S})$ will also hold for $L(\mathcal{F}, \mathcal{P})$. However $L(\mathcal{F}, \mathcal{P})$ although less expressive than $L(\mathcal{U}, \mathcal{S})$ may be sufficient for certain applications. Thus although an algorithm for determining satisfiability in $L(\mathcal{U}, \mathcal{S})$ would also be capable of handling $L(\mathcal{F}, \mathcal{P})$, we will start by defining a (hopefully simpler) algorithm for $L(\mathcal{F}, \mathcal{P})$ initially and then extending it to $L(\mathcal{U}, \mathcal{S})$.

Concluding remarks

Definition 2.8. (satisfied): We say that a formula A (of $L(\mathcal{U}, \mathcal{S})$ or $L(\mathcal{F}, \mathcal{P})$) is satisfied in a model $\mathcal{M} = (T, <, h)$ (denoted $\sigma(A, M)$) if and only if there exists $t \in T$ such that $M, t \vDash A$.

Remark 2.9. In chapter 5, we will write formulas in *polish notation.*This simply consists in writing operators as prefixes. So for example the formula $a\mathcal{S}(b \wedge \neg(c \wedge d))$ would be written in polish notation as $\mathcal{S}a \wedge b\neg \wedge cd$.

2.2 Linear Models of time

As mentioned in the previous section, we evaluate temporal formulas at points of models of the form $\mathcal{M} = (T, <, h)$ where $<$ is a linear order on T. Our starting point for constructing general linear orders will be single point models.

Definition 2.10. (single point model): a single point model is a model of the form $(\{0\}, \{\}, h)$ for some assignment h.

Linear orders can be composed together by means of a lexicographic sum operation in order to form larger orders. Thus in order to build general temporal models starting from single point models, we first define the lexicographic sum of models.

Definition 2.11. (lexicographic sum of models): given a finite set of atoms $\mathcal{L}$, a non empty set I, a linear order $<_I$on I, and for each $i \in I$ a model $\mathcal{M}_i = (T_i, <_i, h_i)$, the lexicographic sum $\sum_{i \in I} \mathcal{M}_i$ is defined as the model $\mathcal{M} = (T, <, h)$ where:

- $T = \{\langle t, i\rangle | i \in I,\ t \in T_i\}$
- $\langle t, i\rangle < \langle t', j\rangle \iff (i <_I j \vee i = j \wedge t <_i t')$
- $h(p) = \{\langle t, i\rangle | i \in I,\ t \in h_i(p)\}$

In the theory of linear orderings, there are four well-known operators which are used to build general linear orders, each of which is an instance of a lexicographic sum. Having defined lexicographic sums for temporal models, we are now in a position to define these operators, which we will use to construct finite model expressions describing temporal models.

2.2.1 Model expressions

We start by giving the syntax of model expressions

Definition 2.12. (model expression M): $M := a | M + M' | \omega M | \omega^* M | < M_1, ..., M_n >$ where a is a finite subset of $\mathcal{L}$.

The semantics of model expressions are as follows

Definition 2.13. (semantics of model expressions): We define the model $\mathcal{M}$ described by the model expression M by induction on the structure of M:

- The expression a describes the single point model $(\{0\}, \{\}, h)$ where $\forall p \in a.h(p) = \{0\}$ and $\forall q \notin a.h(q) = \{\}$

Let M_0, M_1 describe models $\mathcal{M}_0$, $\mathcal{M}_1$. Then

- The expression $M_0 + M_1$ describes the model $\sum_{i \in \{0,1\}} \mathcal{M}_i$, where $0 < 1$.
- The expression ωM_0 describes the model $\sum_{i \in \mathbb{N}} \mathcal{M}_i$, where $\mathcal{M}_i = \mathcal{M}_0$ for all $i \in \mathbb{N}$ (and $<$ is the ordering of the natural numbers). It is illustrated in figure 2.1.

- The expression $\omega^* M_0$ describes the model $\sum_{i \in I} \mathcal{M}_i$, where $I = (\mathbb{Z} \backslash \mathbb{N}) \cup \{0\}$, and $\mathcal{M}_i = \mathcal{M}_0$ for all $i \in I$ (and $<$is the ordering of $(\mathbb{Z} \backslash \mathbb{N}) \cup \{0\}$). It is illustrated in figure 2.2.

The final operator is the shuffle operator. In order to define it we must first give some definitions from the theory of linear orders.

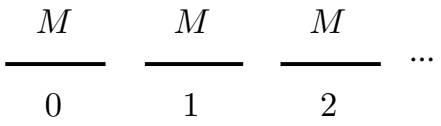

Figure 2.1: The model ωM

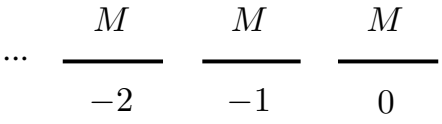

Figure 2.2: The model $\omega^* M$

Definition 2.14. (subordering):Let $<_{\mathrm{R}}$ be a linear ordering of A, and let $<_S$ be a linear ordering of B, and suppose that $A \subseteq B$.We say that $(A, <_{\mathrm{R}})$ is a subordering of $(B, <_S)$ if for every $t, u \in A$, $t <_R u$ iff $t <_S u$. Thus intuitively, $(A, <_{\mathrm{R}})$ is a subordering of $(B, <_S)$ if any two elements of A are ordered by R in the same way that they are ordered by S.

Definition 2.15. (dense in a linear ordering): Let A be a linear ordering and let D be a subordering of A. We say that D is dense in A if between any two elements of A there is an element of D.

We are now in a position to define the semantics of the shuffle operator:

Definition 2.16. (semantics of shuffle operator): Let M_1,...,M_n be model expressions describing models $\mathcal{M}_1$,...,$\mathcal{M}_n$ respectively. For $n \in \mathbb{N}\backslash 0$ let $\mathbb{Q}$ be partitioned into n subsets $\{\mathbb{Q}_j | j < n\}$ each of which is dense in $\mathbb{Q}$. For each $i \in \mathbb{Q}$ define $\mathcal{M}_i$ to be the model $\mathcal{M}_j$ such that $i \in Q_j$. The shuffle $< M_1, ..., M_n >$ of expressions $M_1, ..., M_n$ corresponds to the model $\sum_{i \in \mathbb{Q}} \mathcal{M}_i$.So the shuffle is essentially a dense mixture of the input models, i.e. between any two points in $\mathbb{Q}$, there will be a copy of every input model. The idea is illustrated in figure 2.3, which is from [1].

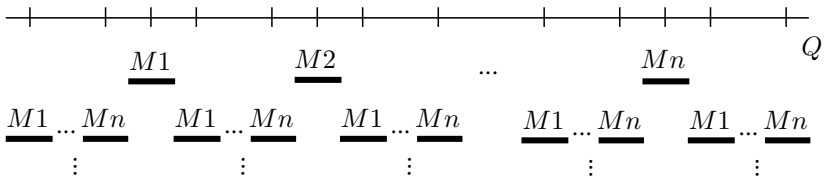

Figure 2.3: The model $< M_1, ..., M_n >$

The usage of the operators is illustrated by example 2.17.

Example 2.17. (example model): Assume we want to describe a model $\mathcal{M}$ isomorphic to $\mathbb{Q}$ such that:

- The atom z is true only at the members of $\mathbb{Z}$
- The atom q is true only at the members of $\mathbb{Q}\backslash\mathbb{Z}$

Then $\mathcal{M}$ can be described (up to isomorphism) by means of the model expression

$$M = \omega^*(\{z\} + < \{q\} >) + \{z\} + \omega(< \{q\} > + \{z\})$$

Finally we introduce some definitions which will be used in the chapters that follow.

Definition 2.18. (isomorphic): We say that two models $\mathcal{M} = (T, <, h)$, $\mathcal{M}' = (T', <', h')$ are isomorphic (denoted $\mathcal{M} \cong \mathcal{M}'$) if and only if there is a bijection $f : T \to T'$ such that:

- for all x, y in T, $x < y \Leftrightarrow f(x) <' f(y)$
- for all $x \in T$, for all $p \in \mathcal{L}$, $x \in h(p) \Leftrightarrow f(x) \in h'(p)$

Isomorphisms preserve truth of temporal formulas ,i.e for any formula A (in $L(\mathcal{F}, \mathcal{P})$ or $L(\mathcal{U}, \mathcal{S})$),any $t \in T$, if $\mathcal{M}' \cong \mathcal{M}$ then $\mathcal{M}, t \vDash A \iff \mathcal{M}', f(t) \vDash A$. For a proof of this see [1],p.7.

Definition 2.19. (constructible model): A model $\mathcal{M}'$ is said to be constructible if there is a model expression M describing a model$\mathcal{M}$ such that $\mathcal{M} \cong \mathcal{M}'$.

Remark. The distinction between a model expression M and the model $\mathcal{M}$ it describes should be clear. Given that in this report we will be primarily concerned with models described by model expressions, for the sake of brevity (where there is no danger of confusion) we will refer to a model expression M when in fact we are referring to the underlying model $\mathcal{M}$ it describes. So for example we will say "The formula F satisfied in the model ωM" to mean "The formula F satisfied in the model which ωM *describes*", we will write $M, t \vDash F$ to mean $\mathcal{M}, t \vDash F$, or $\sum_{i \in I} M_i$ to mean $\sum_{i \in I} \mathcal{M}_i$.

2.2.2 The power of the operators: modelling general flows of time

A question which arises at this point is why did we choose these operators? how expressive are they? In other words what kind of models can we build with them? To answer this question we consider here some relevant results.

Definition 2.20. ($\equiv_k$): for linear models M, N, and $k \in \mathbb{N}$, we define $M \equiv_k N$ to mean that for any formula $A \in L(\mathcal{U}, \mathcal{S})$ with $depth(A) \leq k$ (where depth(A) is maximum nesting of $\mathcal{U}$, $\mathcal{S}$, $\mathcal{W}$, $\mathcal{Z}$ in A)

$$\sigma(A, M) \Longleftrightarrow \sigma(A, N)$$

From the definition of the operators we have the following:

Lemma 2.21. *(linearity of models): Any model M constructed using the four operators is linear.*

Theorem 2.22. *(constructibilty): Given $k \in \mathbb{N}$, for any linear model N there exists a model M constructible using the four operators such that $N \equiv_k M$ ([3]).*

Definition 2.23. ($\mathbb{R} - restricted$ constructible model): We say that a constructible model is $\mathbb{R} - restricted$ if it is constructible by a model expression satisfying the following restrictions:

- the operation $M_0 + M_1$ is only defined if M_0 has a greatest point and M_1 has no least point, or if M_1 has a least point and M_0 has no greatest point.
- the operation ωM_0 is only defined if M_0 has a greatest point and no least point.

- the operation $\omega^* M_0$ is only defined if M_0 has a least point and no greatest point.
- the operation $< M_1, ..., M_n >$ is only defined if at least one of the input models $M_1 ... M_n$ is a single point model, and all remaining input models have endpoints (both a least and greatest point).

Theorem 2.24. *For any $\mathbb{R}-$ restricted constructible model M there exists a model N based on a linear order $T \cong \mathbb{R}$ such that $\forall k \in \mathbb{N}. M \equiv_k N$ ([1])*

Remark 2.25. *:*M is by construction countable, so it is not isomorphic to N ($M \ncong N$)

Theorem 2.26. *For any $k \in \mathbb{N}$, for any model N based on a linear order $T \cong \mathbb{R}$, there exists an $\mathbb{R}-$ restricted constructible model M such that $M \equiv_k N$.([3])*

A direct consequence of the above is that model expressions are sufficiently expressive that for any formula F satisfied in a model based on a real-numbers view of time, F is satisfied in a model described by the language. The theorems illustrate the suitability of model expressions for describing general linear models, and particularly for our purpose of describing models based on real flows of time.

2.3 Computational Complexity

The field of Computational Complexity is concerned with analysing the resources required by algorithms in terms of time and space, which is characterised as their time/space complexity. Furthermore it focuses on the complexity intrinsic to the problems themselves, in other words the resources required by any optimal algorithm to solve a particular problem.Based on this characterisation problems are put into different complexity classes.

In order to analyse the complexity of an algorithm, we create a computational model of that algorithm and analyse the resources it uses. Similarly to analyse the complexity of a particular problem, we create a computational model of an algorithm solving the problem, and analyse it's resource usage. The fundamental model of computation is the Turing Machine. We assume the readers familiarity with the definition of Turing Machines. Subsequently we consider here only a few definitions which will be necessary to understand the results in chapters 5 and 6. The definitions are taken from [10], and should the reader have further queries regarding complexity theory, we refer them to [10].

Definition 2.27. (Language L): A language L is a set of strings (or words) over a given alphabet (i.e. set of symbols).

Definition 2.28. (k-tape I/O TM): An input/output k-tape TM ($k \geq 2$) has:

- input tape: read only head can move freely, but no change of symbol
- $k-2$ work tapes
- output tape:write only head can only move to right

Definition 2.29. $(f(n))SPACE$: An i/o TM operates within space $f(n)$ if on every input of length n it uses $\leq f(n)$ squares of each work tape. L is in $(f(n))SPACE$ if L is decided by an i/o TM operating within space $f(n)$.

Definition 2.30. (PSPACE): L is in PSPACE if L is in $(f(n))SPACE$ for some polynomial $f(n)$.

Remark 2.31. We will write $poly(x)$ to mean "polynomial in x", and $exp(x)$ to mean "exponential in x"

Chapter 3

Determining satisfiability of temporal formulas in linear models

In this chapter we describe our algorithm for determining if a given formula is satisfied in a model described by a model expression (recall definition 2.8). Initially we outline an algorithm for formulas of $L(\mathcal{F}, \mathcal{P})$. We then generalise various parts of the algorithm, so that it can be extended to handle formulas of $L(\mathcal{U}, \mathcal{S})$.

3.1 Determining satisfiability for $L(\mathcal{F}, \mathcal{P})$

Our problem is given a model M described by some model expression, and a finite formula A in $L(\mathcal{F}, \mathcal{P})$, to determine whether A is satisfied in M.

As mentioned previously, M is a lexicographic sum $\sum_{i \in I} M_i$ for some linear order I, and atomic Models M_i. Thus the most obvious approach is that of looking at each world t of the model and checking if $M, t \vDash A$. This clearly may not terminate in the general case, as models may have infinitely many worlds. A follow up idea is that of focusing on each sub-model instead of focusing on each world. Specifically to recursively determine what formulas are satisfied in each of the sub-models M_i, based on this determine what formulas are satisfied in the model $M = \sum_{i \in I} M_i$, and finally check if A is one of them. Evidently as there are infinitely many formulas satisfied in any model, and we are only interested in those relevant to the truth value of A, namely its subformulas, we restrict our focus to these. Thus the idea is to start at the leaves (the 1-point models) of the tree of operations describing our model, determine which subformulas of A are satisfied at each of these, then work our way up the tree determining which subformulas are satisfied at each level until we get to the root. At this point we will have the subformulas of A satisfied in the model M, and can just check if A is one of them.

However we need to consider the fact that certain subformulas which are not satisfied in any of the arguments of a lexicographic sum will be satisfied in the sum itself. For example if we consider the formula $A = \mathcal{F}p$ and the one point models $M_1 = \{\}$, $M_2 = \{p\}$, A is clearly not satisfied in M_1 or M_2, but is satisfied in $M_1 + M_2($ illustrated in figure 3.1). To handle this we introduce the notion of formula *localization*.

Definition 3.1. (localization for $L(\mathcal{F}, \mathcal{P})$): Given a model $M = \sum_{i \in I} M_i$ for some linear order $(I, <)$ and a formula F in the logic of $\mathcal{F}, \mathcal{P}$, we define the localization F_i of F at model M_i by induction on the structure of F:

- If F is atomic then $\forall i \in I . F_i = F$
- $(\neg A)_i = \neg A_i$

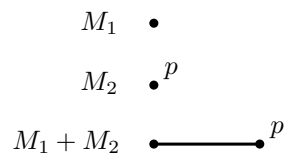

Figure 3.1: why we need localisation

- $(A \wedge B)_i = A_i \wedge B_i$
- $(\mathcal{F}A)_i = \begin{cases} \top & if\ \exists j >_I i. A_j\ satisfied\ in\ M_j \\ \mathcal{F}(A_i) & otherwise \end{cases}$
- $(\mathcal{P}A)_i = \begin{cases} \top & if\ \exists j <_I i. A_j\ satisfied\ in\ M_j \\ P(A_i) & otherwise \end{cases}$

Intuitively, Given a model $M = \sum_{i \in I} M_i$, the localization A_i of a formula A at a sub model M_i of M is a rewritten version of the formula incorporating information about the other models in the sum $M = \sum_{i \in I} M_i$. Thus evaluating the localization A_i at the worlds of M_i (considering M_i in isolation) will be equivalent to evaluating the unlocalized formula A at the worlds of the model M_i in the context of the model M. This idea is formalised by the lemma that follows.

Lemma 3.2. *(localization lemma for $L(\mathcal{F}, \mathcal{P})$): Given a linear order $(I, <_I)$, a model $M = \sum_{i \in I} M_i$, and a formula $A \in L(\mathcal{F}, \mathcal{P})$:*

$$\forall i \in I. \forall t \in T_i. (M, \langle t, i \rangle \models A \Leftrightarrow M_i, t \models A_i)$$

where $\langle t, i \rangle$ denotes the world of M corresponding to the world t of M_i

Proof. Pick arbitrary $i \in I$,$t \in T_i$. we prove the lemma by induction on the structure of A:

- (Base Case) A atomic:
  1. $M, \langle t, i \rangle \models A \Leftrightarrow M_i, t \models A$ (by definition of $M = \sum_{i \in I} M_i$ for atomic A)
  2. $A_i = A$ (by definition of localization for atomic A)
  3. $M, \langle t, i \rangle \models A \Leftrightarrow M_i, t \models A_i$(by 1,2)
- Inductive Hypothesis: property holds for A,B
  - $M, \langle t, i \rangle \models \neg A$
    $\Leftrightarrow M, \langle t, i \rangle \not\models A$
    $\Leftrightarrow M_i, t \not\models A_i$(by IH)
    $\Leftrightarrow M_i, t \models \neg A_i$
    $\Leftrightarrow M_i, t \models (\neg A)_i$(by def. of localization)
  - $M, \langle t, i \rangle \models A \wedge B$
    $\Leftrightarrow M, \langle t, i \rangle \models A$and $M, \langle t, i \rangle \models B$
    $\Leftrightarrow M_i, t \models A_i$and $M_i, t \models B_i$(by IH)
    $\Leftrightarrow M_i, t \models A_i \wedge B_i$
    $\Leftrightarrow M_i, t \models (A \wedge B)_i$(by def. of localization)
  - to show $M, \langle t, i \rangle \models \mathcal{F}A \Leftrightarrow M_i, t \models (\mathcal{F}A)_i$ we have two cases:

* If $\exists j >_I i.A_j$ satisfied in M_j(assumption):
  $\Rightarrow \exists t' \in T_j.M_j, t' \models A_j$ (by def. 2.8-satisfied)
  $\Rightarrow \exists t' \in T_j.M, \langle t', j\rangle \models A$(by IH)
  $\Rightarrow \exists \langle t', j\rangle.\langle t, i\rangle < \langle t', j\rangle \wedge M, \langle t', j\rangle \models A$ (by assumption, def. of $<$)
  $\Rightarrow M, \langle t, i\rangle \models \mathcal{F}A$
  Also by definition of localization and assumption $(\mathcal{F}A)_i = \top$, so trivially $M_i, t \models (\mathcal{F}A)_i$. Thus $M, \langle t, i\rangle \models \mathcal{F}A \Leftrightarrow M_i, t \models (\mathcal{F}A)_i$ as both terms are true.
* Otherwise $\neg\exists j >_I i.A_j$satisfied in M_j (assumption):
  $\Rightarrow \neg\exists j >_I i.\exists t' \in T_j.M_j, t' \models A_j$ (by def. of satisfied)
  $\Rightarrow \neg\exists j >_I i.\exists t' \in T_j.M, \langle t', j\rangle \models A$(by IH) (A)
  Subsequently
  $M, \langle t, i\rangle \models \mathcal{F}A$
  $\Leftrightarrow \exists t'.t <_i t' \wedge M, \langle t', i\rangle \models A$ (by A, semantics of $\mathcal{F}$ and def. of$<_I$)
  $\Leftrightarrow \exists t'.t <_i t' \wedge M_i, t' \models A_i$ (by IH)
  $\Leftrightarrow M_i, t \models \mathcal{F}A_i$ (by defintion of $\mathcal{F}$)
  $\Leftrightarrow M_i, t \models (\mathcal{F}A)_i$ (by definition of localization and assumption)

– The case for $M_i, t \models (\mathcal{P}A)_i \Leftrightarrow M, \langle t, i\rangle \models \mathcal{P}A$ is entirely temporally symmetrical (replace $\mathcal{F}$ with $\mathcal{P}$ and $<$ with $>$).

□

From this lemma we can prove the following:

Theorem 3.3. *(localization Theorem for $L(\mathcal{F}, \mathcal{P})$): Given a linear order $(I, <_I)$, a model $M = \sum_{i\in I} M_i$, and a formula $A \in L(\mathcal{F}, \mathcal{P})$:*

$$A \text{ is satisfied in } M = \textstyle\sum_{i\in I} M_i \Longleftrightarrow \exists i \in I.A_i \text{is satisfied in } M_i$$

Proof. A is satisfied in $M = \sum_{i\in I} M_i$
$\Leftrightarrow \exists t \in T.M, t \models A$ (by def. of satisfied)
$\Leftrightarrow \exists i \in I.\exists t \in T_i.M, \langle t, i\rangle \models A$ (by def. of $\sum_{i\in I} M_i$)
$\Leftrightarrow \exists i \in I.\exists t \in T_i.M_i, t \models A_i$ (by Lemma 3.2)
$\Leftrightarrow \exists i \in I.A_i$ satisfied in M_i(by def. of satisfied) □

The localization theorem is the first step towards our goal of using localization to determine satisfiability. Eventually we will use it to formalise the idea given at the start of this chapter into an algorithm for solving the model checking problem for $L(\mathcal{F}, \mathcal{P})$, i.e. deciding whether a given formula A of $L(\mathcal{F}, \mathcal{P})$ is satisfied in a given model M. In order to do this we give some more definitions.

Definition 3.4. $(S(A))$: for a formula $A \in L(\mathcal{F}, \mathcal{P})$, $S(A)$ is defined as the minimal closure of A under subformulas, and localization. I.e. it is the smallest set such that:

- $A \in S(A)$
- $subformulas(A) \subseteq S(A)$
- for any model $\sum_{i\in I} M_i$, linear order $(I, <_I)$, $\forall B \in S(A).\forall i \in I.B_i \in S(A)$

$S(A)$ can be constructed as the set of every possible localization of every subformula B of A, where a possible localization of a formula B is the result of replacing any number of its subformulas of the form $\mathcal{F}C$ or $\mathcal{P}C$ with $\top$. $S(A)$ is clearly both finite, and finitely computable.

Definition 3.5. ($\Sigma_A(M)$): Given a formula $A \in L(\mathcal{F}, \mathcal{P})$ and a model M, we define

$$\Sigma_A(M) = \{B \in S(A) : B \text{ is satisfied in } M\}$$

Using these definitions, we can rephrase Theorem 3.3 by defining $\Sigma_A(\sum_{i\in I} M_i)$ in terms of $\Sigma_A(M_i)$.

Proposition 3.6. (definition of $\Sigma_A(\sum_{i\in I} M_i)$*): for any lexicographic sum* $\sum_{i\in I} M_i$

$$\Sigma_A(\sum_{i\in I} M_i) = \{B \in S(A) : \exists i \in I.B_i \in \Sigma_A(M_i)\}$$

Proof. :

$$\begin{aligned}
\Sigma_A(\sum_{i\in I} M_i) &= \{B \in S(A) : B \text{ is satisfied in } M\} \text{ (by definition)} \\
&= \{B \in S(A) : \exists i \in I.B_i \text{ is satisfied in } M_i\} \text{ (by Localization Theorem-3.3)} \\
&= \{B \in S(A) : \exists i \in I.B_i \in \Sigma_A(M_i)\} \text{ (by definition of } \Sigma_A(M_i), \text{ as } B_i \in S(A) \\
&\quad \text{by definition of } S(A))
\end{aligned}$$

□

The algorithm

The significance of proposition 3.6 for the purpose of determining satisfiability of a formula A in a model $\sum_{i\in I} M_i$, is that we have moved from considering potentially infinite structures M_i to finite sets $\Sigma_A(M_i)$ ($\Sigma_A(M_i)$ is by definition finite, as it is a subset of $S(A)$ which is finite). However to construct $\Sigma_A(\sum_{i\in I} M_i)$ we still have to construct $\Sigma_A(M_i)$ for each $i \in I$, and I could be infinite. As such, this does not give us an algorithm. In the following section, we will show that if $M = \sum_{i\in I} M_i$ is a constructible model , we can adapt proposition 3.6 into an algorithm for computing $\Sigma_A(M)$, and subsequently determining satisfiability by checking if $A \in \Sigma_A(M)$.

Specifically we will use proposition 3.6 together with some intermediary lemmas on localizations in constructible models to show that for any constructible model $M = \_(M_0...M_n)$ where $\_$ is one of the four operators and $M_0...M_n$ are the input models, we can construct $\Sigma_A(M)$ purely in terms of $\Sigma_A(M_i)$ for $i \in \{0...n\}$, where n is by definition finite. The result will be that for any constructible model M, $\Sigma_A(M)$ can be computed *finitely* by induction over the model tree of M. Firstly we note that

Lemma 3.7. *($\Sigma_A(a)$):There is an algorithm which given any formula $A \in L(\mathcal{F}, \mathcal{P})$, and any single point model a computes $\Sigma_A(a)$.*

Proof. : Recall that a is a finite subset of $\mathcal{L}$, denoting the atoms true at the world 0 of a single point model. initially $\Sigma_A(a) = \{\}$. The algorithm first constructs $S(A)$ by generating the subformulas of A , and for each one generating every possible version of it where arbitrarily many of its subformulas of the form $\mathcal{F}C$ or $\mathcal{P}C$ are replaced with $\top$. Then, for each formula $B \in S(A)$ it evaluates it at 0:

- if B is $\top$ return true
- if B is an atom $p \in \mathcal{L}$ then return true if $p \in a$ and false otherwise.

- if $B = \neg C$ then recursively call the algorithm on C, returning the negation of its result.
- if $B = C \wedge D$ recursively call the algorithm on C and D, returning true if both recursive calls return true and false otherwise.
- if $B = \mathcal{F}C$ or $B = \mathcal{P}C$ return false

If the formula evaluates to true add it to $\Sigma_A(a)$. When there are no more formulas in $S(A)$ return $\Sigma_A(a)$.

□

In the sections that follow, we will describe how to compute $\Sigma_A(\_(M_0...M_n))$ in terms of $\Sigma_A(M_0)...\Sigma_A(M_n)$ for each of the four operators.

Computing $\Sigma_A(M_0 + M_1)$

By proposition 3.6 and definition of $M_0 + M_1$ we have $\Sigma_A(M_0 + M_1) = \{B \in S(A) : \ B_0 \in \Sigma_A(M_0) \vee B_1 \in \Sigma_A(M_1)\}$.

Furthermore, for every $B \in S(A)$ we can compute the localizations B_0, B_1 of B inductively, by specialising the definition of localisation (definition 3.1) to the model $M = M_0 + M_1$:

- If B is atomic then $B_0 = B_1 = B$
- for $i \in \{0, 1\}$, $(\neg C)_i = \neg C_i$
- for $i \in \{0, 1\}$, $(C \wedge D)_i = C_i \wedge D_i$
- $(\mathcal{F}C)_0 = \begin{cases} \top & \text{if } C_1 \in \Sigma_A(M_1) \\ \mathcal{F}(C_0) & \text{otherwise} \end{cases}$
- $(\mathcal{F}C)_1 = \mathcal{F}(C_1)$
- $(\mathcal{P}C)_0 = \mathcal{P}(C_0)$
- $(\mathcal{P}C)_1 = \begin{cases} \top & \text{if } C_0 \in \Sigma_A(M_0) \\ \mathcal{P}(C_1) & \text{otherwise} \end{cases}$

Notice that the localisation context conditions "C_i is satisfied in M_i" become $C_i \in \Sigma_A(M_i)$ by definition of $\Sigma_A(M_i)$, because $C_i \in S(A)$ by definition of $S(A)$ (C_i is a possible localisation of a subformula of B).

From the above it follows that

Lemma 3.8. *($\Sigma_A(M_0 + M_1)$): There is an algorithm which for any models M_0, M_1 given as input a formula $A \in L(\mathcal{F}, \mathcal{P})$ and $\Sigma_A(M_0)$, $\Sigma_A(M_1)$ computes $\Sigma_A(M_0 + M_1)$.*

Computing $\Sigma_A(< M_1, ..., M_n >)$

By the definition of shuffle as a lexicographic sum where $I = \mathbb{Q}$, and proposition 3.6 we have

$$\Sigma_A(< M_1, ..., M_n >) = \{B \in S(A) : \ \exists i \in \mathbb{Q}.B_i \in \Sigma_A(M_i)\}$$

and thus should in theory compute B_i for every $i \in \mathbb{Q}$. This is in fact unnecessary as a result of the following Lemma.

Lemma 3.9. *(Shuffle localization lemma for $L(\mathcal{F},\mathcal{P})$): For any formula $B \in L(\mathcal{F},P)$:*

$$\forall i,j \in \mathbb{Q}.(B_i = B_j)$$

Proof. : Pick arbitrary $B \in L(\mathcal{F},P)$, $i,j \in \mathbb{Q}$ Then:

- If B is atomic then by definition of localization$B_i = B_j = B$

(IH): assume the lemma for formulas C, D.

- $(\neg C)_i$
  $= \neg C_i$ (by def. of loc.)
  $= \neg C_j$ (by IH)
  $= (\neg C)_j$ (by def. of loc.)
- $(C \wedge D)_i$
  $= C_i \wedge D_i$ (by def. of loc.)
  $= C_j \wedge D_j$ (by IH)
  $= (C \wedge D)_j$ (by def. of loc.)
- Assume $\exists h >_I i$ such that C_h is satisfied in M_h(assumption)

  1. $(\mathcal{F}C)_i = \top$ (by def. of loc., assumption)
  2. $\exists x \in \{0,1,...,n\}.M_h = M_x$ (by assumption, def. of shuffle)
  3. $\exists k >_I j.M_k = M_x$(by definition of shuffle)
  4. $C_k = C_h$ (by IH)
  5. $\exists k >_I j$ such that C_k is satisfied in M_k (by assumption, 2,3,4)
  6. $(\mathcal{F}C)_j = \top$ (by 5, def. of loc.)
  7. $(\mathcal{F}C)_i = (\mathcal{F}C)_j$(by 1,6)

  Otherwise $\neg\exists h >_I i$ such that C_h is satisfied in M_h(assumption) in which case:
  $(\mathcal{F}C)_i$
  $= \mathcal{F}C_i$(by assumption, def. of loc.)
  $= \mathcal{F}C_j$(by IH)
  $= (\mathcal{F}C)_j$(by assumption, def. of shuffle, def. of loc.)

- The case for $(\mathcal{P}C)_i = (\mathcal{P}C)_j$ is symmetric

□

As a result of the Lemma we can simplify the definition of localization for formulas $B \in S(A)$, in terms of the single localization B_λ that we need to compute:

- If B is atomic then $B_\lambda = B$
- $(\neg B)_\lambda = \neg B_\lambda$
- $(C \wedge D)_\lambda = C_\lambda \wedge D_\lambda$
- $(\mathcal{F}C)_\lambda = \begin{cases} \top & if\ C_\lambda \in \bigcup_{i=0}^{n} \Sigma_A(M_i) \\ \mathcal{F}(C_\lambda) & otherwise \end{cases}$

- $(\mathcal{P}C)_\lambda = \begin{cases} \top & if\ C_\lambda \in \bigcup_{i=0}^{n} \Sigma_A(M_i) \\ \mathcal{P}(C_\lambda) & otherwise \end{cases}$

Note that we consider $\bigcup_{i=0}^{n} \Sigma_A(M_i)$ for the localization conditions as by the definition of shuffle for any $i \in \mathbb{Q}$ we have $\bigcup_{j>i} \Sigma_A(M_j) = \bigcup_{j<i} \Sigma_A(M_j) = \bigcup_{i\in\mathbb{Q}} \Sigma_A(M_i) = \bigcup_{i=0}^{n} \Sigma_A(M_i)$. Subsequently

$$\begin{aligned}
\Sigma_A(< M_0, M_1 ... M_n >) &= \{B \in S(A) :\ \exists i \in \mathbb{Q}.B_i \in \Sigma_A(M_i)\} \\
&= \{B \in S(A) :\ \exists i \in \mathbb{Q}.B_\lambda \in \Sigma_A(M_i)\} \text{ (by Lemma 3.9)} \\
&= \{B \in S(A) :\ B_\lambda \in \bigcup_{i\in\mathbb{Q}} \Sigma_A(M_i)\} \\
&= \{B \in S(A) :\ B_\lambda \in \bigcup_{i=0}^{n} \Sigma_A(M_i)\} \text{ (as } \bigcup_{i\in\mathbb{Q}} \Sigma_A(M_i) = \bigcup_{i=0}^{n} \Sigma_A(M_i) \\
&\quad \text{by definition of shuffle)}
\end{aligned}$$

and clearly it follows that:

Lemma 3.10. *($\Sigma_A(< M_1, ..., M_n >)$): There is an algorithm which for any models M_1,...,M_n given as input a formula $A \in L(\mathcal{F}, \mathcal{P})$ and $\Sigma_A(M_1)$,...,$\Sigma_A(M_n)$ computes $\Sigma_A(< M_1 ... M_n >)$.*

Computing $\Sigma_A(\omega M_0)$

We start by specialising the definition of localisation to the model ωM_0. Recall that for ωM_0 the index ordering is $I = \mathbb{N}$. Thus for $B \in S(A)$, according to definition 3.1 B_n is defined for $n \geq 0$:

- If B is atomic then $\forall n.B_n = B$
- $(\neg B)_n = \neg B_n$
- $(A \wedge B)_n = A_n \wedge B_n$
- $(\mathcal{F}B)_n = \begin{cases} \top & if\ \exists m.m > n \wedge B_m \in \Sigma_A(M_0) \\ \mathcal{F}(B_n) & otherwise \end{cases}$
- $(\mathcal{P}B)_n = \begin{cases} \top & if\ \exists m.0 \leq m < n \wedge B_m \in \Sigma_A(M_0) \\ P(B_n) & otherwise \end{cases}$

By proposition 3.6 we have $\Sigma_A(\omega M_0) = \{B \in S(A) :\ \exists n \in \mathbb{N}.B_n \in \Sigma_A(M_0)\}$. According to this definition in order to compute $\Sigma_A(\omega M_0)$ we potentially need to compute the localization B_n of B for all natural numbers. This is in fact not necessary, as a result of the following lemma.

Definition 3.11. ($d(F)$ for $F \in L(\mathcal{F}, P)$):For a formula $B \in L(\mathcal{F}, P)$, we define $d(B)$ to be the maximum depth of nesting of $\mathcal{P}$ in B:

- If B is atomic then $d(B) = 0$
- $d(\neg B) = d(B)$
- $d(A \wedge B) = max(d(A), d(B))$

- $d(\mathcal{F}B) = d(B)$
- $d(\mathcal{P}B) = d(B) + 1$

Lemma 3.12. *(ω localization lemma for $L(\mathcal{F}, \mathcal{P})$): given a model M, a formula $A \in L(\mathcal{F}, \mathcal{P})$, and $B \in S(A)$, $\forall n \geq d(B).B_n = B_{d(B)}$, i.e. the localizations of B converge at the sub-model $M_{d(B)}$.*

Proof. :

- If B is atomic then $d(B) = 0$, and $\forall n \geq 0.B_n = B_0 = B$

(IH): assume the lemma for formulas C, D.

- $B = \neg C$: pick $n \geq d(\neg C) = d(C)$.

  $(\neg C)_n$
  $= \neg(C_n)$ (by def. of loc.)
  $= \neg(C_{d(C)})$ (by IH)
  $= (\neg C)_{d(C)}$ (by def. of loc.)
  $= (\neg C)_{d(\neg C)}$ (by def. of $d()$)

- $B = C \wedge D$: pick $n \geq d(C \wedge D) = max(d(C), d(D))$

  $(C \wedge D)_n$
  $= C_n \wedge D_n$ (by def. of loc.)
  $= C_{d(C)} \wedge D_{d(D)}$(by IH)
  $= C_{d(C \wedge D)} \wedge D_{d(C \wedge D)}$ (by IH, as $d(C \wedge D) \geq d(C)$, $d(C \wedge D) \geq d(D)$)
  $= (C \wedge D)_{d(C \wedge D)}$(by def. of loc.)

- $B = \mathcal{F}C$: pick $n \geq d(\mathcal{F}C) = d(C)$
  Recall:
  $$(\mathcal{F}C)_n = \begin{cases} \top & if\ \exists m.m > n \wedge C_m \in \Sigma_A(M_0) \\ \mathcal{F}(C_n) & otherwise \end{cases}$$
  $$(\mathcal{F}C)_{d(\mathcal{F}C)} = \begin{cases} \top & if\ \exists m.m > d(\mathcal{F}C) \wedge C_m \in \Sigma_A(M_0) \\ \mathcal{F}(C_{d(\mathcal{F}C)}) & otherwise \end{cases}$$
  However
  $C_n = C_{d(C)}$ (by IH, as $n \geq d(C)$)
  $= C_{d(\mathcal{F}C)}$ (by def. of $d()$)
  Thus $\mathcal{F}(C_n) = \mathcal{F}(C_{d(\mathcal{F}C)})$. Additionally, as $n \geq d(\mathcal{F}C) = d(C)$ the conditions for the $\top$ cases are both equivalent to $C_{d(C)} \in \Sigma_A(M_0)$ by IH. It follows that $(\mathcal{F}C)_n = (\mathcal{F}C)_{d(\mathcal{F}C)}$.

- $B = \mathcal{P}C$: pick $n \geq d(\mathcal{P}C) = d(C) + 1$ ($\Rightarrow n > d(C)$)
  Recall:
  $$(\mathcal{P}C)_n = \begin{cases} \top & if\ \exists m.0 \leq m < n \wedge C_m \in \Sigma_A(M_0) \\ P(C_n) & otherwise \end{cases}$$
  $$(\mathcal{P}C)_{d(\mathcal{P}C)} = \begin{cases} \top & if\ \exists m.0 \leq m < d(\mathcal{P}C) \wedge C_m \in \Sigma_A(M_0) \\ P(C_{d(\mathcal{P}C)}) & otherwise \end{cases}$$
  We first show equivalence of the conditions for the $\top$cases:
  $\exists m.0 \leq m < n \wedge C_m \in \Sigma_A(M_0)$
  $\Leftrightarrow \exists m.(0 \leq m \leq d(C) \vee d(C) < m < n) \wedge C_m \in \Sigma_A(M_0)$ (as $n > d(B)$)
  $\Leftrightarrow \exists m.(0 \leq m \leq d(C)) \wedge C_m \in \Sigma_A(M_0)$ (as by IH $m > d(C) \Rightarrow C_m = C_{d(C)}$)

$\Leftrightarrow \exists m.(0 \leq m < d(C)+1) \wedge C_m \in \Sigma_A(M_0)$
$\Leftrightarrow \exists m.(0 \leq m < d(\mathcal{P}C)) \wedge C_m \in \Sigma_A(M_0)$ (by def. of $d()$)
Additionally:
$P(C_n)$
$= P(C_{d(C)})$ (by IH, as $n > d(C)$)
$= P(C_{d(C)+1})$ (by IH, as $d(C)+1 > d(C)$)
$= P(C_{d(\mathcal{P}C)})$ (by def. of $d()$)

$\mathrm{So}(\mathcal{P}C)_n = (\mathcal{P}C)_{d(\mathcal{P}C)}$.

□

As a result of Lemma 3.12 we have $\Sigma_A(\omega M_0) = \{B \in S(A) : \exists n \in \mathbb{N}.B_n \in \Sigma_A(M_0)\} = \{B \in S(A) : \exists n \leq d(B).B_n \in \Sigma_A(M_0)\}$. Furthermore, as a result of the Lemma we have that for $n \leq d(B)$ the context condition $\exists m.m > n \wedge B_m \in \Sigma_A(M_0)$ which is required for computing $(\mathcal{F}B)_n$ is equivalent to the finitely computable condition $(\exists m.n < m \leq d(B) \wedge B_m \in \Sigma_A(M_0)) \vee (n = d(B) \wedge B_n \in \Sigma_A(M_0))$.

Claim 3.13. $n \leq d(B) \wedge \exists m.m > n \wedge B_m \in \Sigma_A(M_0) \iff (\exists m.n < m \leq d(B) \wedge B_m \in \Sigma_A(M_0)) \vee (n = d(B) \wedge B_n \in \Sigma_A(M_0))$

Proof. :
$n \leq d(B) \wedge \exists m.m > n \wedge B_m \in \Sigma_A(M_0)$
$\Leftrightarrow (n = d(B) \wedge \exists m.m > n \wedge B_m \in \Sigma_A(M_0)) \vee (n < d(B) \wedge \exists m.m > n \wedge B_m \in \Sigma_A(M_0))$ (def. of $\leq$ distributivity of $\wedge$)
$\Leftrightarrow (n = d(B) \wedge B_n \in \Sigma_A(M_0)) \vee (n < d(B) \wedge \exists m.m > n \wedge B_m \in \Sigma_A(M_0))$ (by Lemma 3.12)
$\Leftrightarrow (n = d(B) \wedge B_n \in \Sigma_A(M_0)) \vee \exists m.((n < m < d(B) \vee m \geq d(B)) \wedge B_m \in \Sigma_A(M_0))$
$\Leftrightarrow (n = d(B) \wedge B_n \in \Sigma_A(M_0)) \vee \exists m.((n < m < d(B) \wedge B_m \in \Sigma_A(M_0)) \vee (m \geq d(B)) \wedge B_m \in \Sigma_A(M_0)))$ (distributivity of $\wedge$)
$\Leftrightarrow (n = d(B) \wedge B_n \in \Sigma_A(M_0)) \vee \exists m.((n < m < d(B) \wedge B_m \in \Sigma_A(M_0)) \vee (m = d(B)) \wedge B_m \in \Sigma_A(M_0)))$ (by Lemma 3.12)
$\Leftrightarrow (n = d(B) \wedge B_n \in \Sigma_A(M_0)) \vee \exists m.((n < m \leq d(B) \wedge B_m \in \Sigma_A(M_0)))$ □

As a result of this we get a finitely (inductively) computable definition of localization for $B \in S(A)$, $n \leq d(B)$:

- If B is atomic then $\forall n.B_n = B$
- $(\neg B)_n = \neg B_n$
- $(A \wedge B)_n = A_n \wedge B_n$
- $(\mathcal{F}B)_n = \begin{cases} \top & \text{if } \exists m.n < m \leq d(B) \wedge B_m \in \Sigma_A(M_0) \\ & \text{or } n = d(B) \wedge B_n \in \Sigma_A(M_0) \\ \mathcal{F}(B_n) & \text{otherwise} \end{cases}$
- $(\mathcal{P}B)_n = \begin{cases} \top & \text{if } \exists m.0 \leq m < n \wedge B_m \in \Sigma_A(M_0) \\ P(B_n) & \text{otherwise} \end{cases}$

As we have shown $\Sigma_A(\omega M_0) = \{B \in S(A) : \exists n \leq d(B).B_n \in \Sigma_A(M_0)\}$ from the above it follows that

Lemma 3.14. *($\Sigma_A(\omega M_0)$): There is an algorithm which for any model M_0 given as input a formula $A \in L(\mathcal{F}, \mathcal{P})$ and $\Sigma_A(M_0)$ computes $\Sigma_A(\omega M_0)$.*

Computing $\Sigma_A(\omega^* M_0)$

as the ω^* operator is the mirror image of the ω operator, this whole case will be the mirror image of the case above, so we will only outline it.

Recall that for $\omega^* M_0$ the index ordering is $I = (\mathbb{Z}\backslash\mathbb{N}) \cup \{0\}$. As in the case for $\Sigma_A(\omega M_0)$, we have a similar localization lemma.

Definition 3.15. ($d^*(F)$ for $F \in L(\mathcal{F}, P)$):For a formula $B \in L(\mathcal{F}, P)$, we define $d^*(B)$ to be the maximum depth of nesting of $\mathcal{F}$ in B:

- If B is atomic then $d(B) = 0$
- $d(\neg B) = d(B)$
- $d(A \wedge B) = max(d(A), d(B))$
- $d(\mathcal{P}B) = d(B)$
- $d(\mathcal{F}B) = d(B) + 1$

Lemma 3.16. *(ω^* localization lemma for $L(\mathcal{F}, \mathcal{P})$): given a model M, a formula $A \in L(\mathcal{F}, \mathcal{P})$, and $B \in S(A)$, $\forall n \leq -d^*(B).B_n = B_{-d^*(B)}$, i.e. the localizations of B converge at the sub-model $M_{-d^*(B)}$.*

Proof. : the mirror image of the proof of Lemma 3.12. □

As a result of this we get a finitely (inductively) computable definition of localization for $B \in S(A)$, $-d^*(B) \leq n \leq 0$:

- If B is atomic then $\forall n.B_n = B$
- $(\neg B)_n = \neg B_n$
- $(A \wedge B)_n = A_n \wedge B_n$
- $(\mathcal{F}B)_n = \begin{cases} \top & \text{if } \exists m.n < m \leq 0 \wedge B_m \in \Sigma_A(M_0) \\ & \text{or } n = d(B) \wedge B_n \in \Sigma_A(M_0) \\ \mathcal{F}(B_n) & \text{otherwise} \end{cases}$
- $(\mathcal{P}B)_n = \begin{cases} \top & \text{if } \exists m. -d^*(B) \leq m < n \wedge B_m \in \Sigma_A(M_0) \\ & \text{or } n = -d^*(B) \wedge B_n \in \Sigma_A(M_0) \\ P(B_n) & \text{otherwise} \end{cases}$

As a result of the lemma we also have $\Sigma_A(\omega^* M_0) = \{B \in S(A) : \exists n. -d^*(B) \leq n \leq 0.B_n \in \Sigma_A(M_0)\}$. So unsurprisingly, as in the $\Sigma_A(\omega M_0)$ case we get

Lemma 3.17. *($\Sigma_A(\omega^* M_0)$): There is an algorithm which for any model M_0 given as input a formula $A \in L(\mathcal{F}, \mathcal{P})$ and $\Sigma_A(M_0)$ computes $\Sigma_A(\omega^* M_0)$.*

Summary

Proposition 3.18. *There is an algorithm which given any formula $A \in L(\mathcal{F}, \mathcal{P})$, and any model expression M, computes $\Sigma_A(M)$.*

Proof. : By induction on the structure of M. Pick arbitrary $A \in L(\mathcal{F}, \mathcal{P})$ and model M:

- (Base Case) $M = a$: can compute $\Sigma_A(M)$ by lemma 3.7

(IH) Assume lemma holds for M_0,...,M_n, i.e. we can compute $\Sigma_A(M_i)$ for $i \in \{0..n\}$

- $M = M_0 + M_1$: can compute $\Sigma_A(M)$ by IH and lemma 3.8
- $M =< M_1, ..., M_n >$: can compute $\Sigma_A(M)$ by IH and lemma 3.10
- $M = \omega M_0$: can compute $\Sigma_A(M)$ by IH and lemma 3.14
- $M = \omega^* M_0$: can compute $\Sigma_A(M)$ by IH and lemma 3.17

□

The algorithm in question works as follows: Consider the syntax tree of the model expression M, the leaves of which are the one point models a. We start by computing $\Sigma_A(a)$ for each of these models. Starting from these models, work up the tree, at each stage computing $\Sigma_A(M_i)$ of a node M_i in the tree, in terms of $\Sigma_A(M_1)...\Sigma_A(M_n)$ where $M_1, ...M_n$ are its children (using the definitions given in the preceding four sections). Iterate this procedure all the way to the root of M. At this point we will have computed $\Sigma_A(M)$.

Determining if A is satisfied in M is then reduced to just checking if $A \in \Sigma_A(M)$, as by definition of $\Sigma_A(M)$, $A \in \Sigma_A(M) \Leftrightarrow A$ is sastisfied in M.

3.2 Determining satisfiability for $L(\mathcal{U}, \mathcal{S})$

In this section we will extend the algorithm described in the previous section to decide satisfiability of formulas of $L(\mathcal{U}, \mathcal{S})$ in constructible models M. As noted in Chapter 2, $L(\mathcal{F}, \mathcal{P})$ is in fact a subset of $L(\mathcal{U}, \mathcal{S})$ meaning that this new algorithm is a more powerful extension of the one described in the previous section. In order to generalise the algorithm to $L(\mathcal{U}, \mathcal{S})$ we must first generalise the definitions and results of the previous section to $L(\mathcal{U}, \mathcal{S})$. The starting point is the key concept of the algorithm, namely that of localization.

Definition 3.19. (localization for $L(\mathcal{U}, \mathcal{S})$): Given a model $M = \sum_{i \in I} M_i$ for some linear order $(I, <)$ and a formula C in $L(\mathcal{U}, \mathcal{S})$, we define the localization C_i of C at model M_i by induction on the structure of C:

- If C is atomic, $\forall i \in I. C_i = C$
- $(\neg A)_i = \neg A_i$
- $(A \wedge B)_i = A_i \wedge B_i$
- $(A\mathcal{U}B)_i = \begin{cases} A_i \mathcal{W} B_i & \text{if } \exists j >_I i. (\sigma(B_j \wedge \mathcal{H}A_j, M_j) \wedge \forall k. (i <_I k <_I j) \rightarrow \neg\sigma(\neg A_k, M_k)) \\ A_i \mathcal{U} B_i & \text{otherwise} \end{cases}$
- $(A\mathcal{W}B)_i = \begin{cases} A_i \mathcal{W} B_i & \text{if } \exists j >_I i. (\sigma(B_j \wedge \mathcal{H}A_j, M_j) \wedge \forall k. (i <_I k <_I j) \rightarrow \neg\sigma(\neg A_k, M_k)) \\ & \text{or } \forall k >_I i. \neg\sigma(\neg A_k, M_k) \\ A_i \mathcal{U} B_i & \text{otherwise} \end{cases}$
- $(A\mathcal{S}B)_i = \begin{cases} A_i \mathcal{Z} B_i & \text{if } \exists j <_I i. (\sigma(B_j \wedge \mathcal{G}A_j, M_j) \wedge \forall k. (j <_I k <_I i) \rightarrow \neg\sigma(\neg A_k, M_k)) \\ A_i \mathcal{S} B_i & \text{otherwise} \end{cases}$
- $(A\mathcal{Z}B)_i = \begin{cases} A_i \mathcal{Z} B_i & \text{if } \exists j <_I i. (\sigma(B_j \wedge \mathcal{G}A_j, M_j) \wedge \forall k. (j <_I k <_I i) \rightarrow \neg\sigma(\neg A_k, M_k)) \\ & \text{or } \forall k <_I i. \neg\sigma(\neg A_k, M_k) \\ A_i \mathcal{S} B_i & \text{otherwise} \end{cases}$

We can now generalise the localization lemma of $L(\mathcal{F}, \mathcal{P})$ (lemma 3.2) to $L(\mathcal{U}, \mathcal{S})$

Lemma 3.20. *(localization lemma for $L(\mathcal{U}, \mathcal{S})$): Given a linear order $(I, <_I)$, a model $M = \sum_{i \in I} M_i$, and a formula $C \in L(\mathcal{U}, \mathcal{S})$:*

$$\forall i \in I. \forall t \in T_i. (M, \langle t, i \rangle \models C \Leftrightarrow M_i, t \models C_i)$$

where $\langle t, i \rangle$ denotes the world of M corresponding to the world t of M_i

Proof. Pick arbitrary $i \in I$,$t \in T_i$. we prove the lemma by induction on the structure of C. The cases where C is boolean are identical to those in the proof of the localization lemma for $L(\mathcal{F}, \mathcal{P})$ (lemma 3.2), so we consider here only the cases for $C = A\mathcal{U}B$, $C = A\mathcal{W}B$, $C = A\mathcal{S}B$, $C = A\mathcal{Z}B$. Assume inductively that the lemma holds for A, B (IH).

- to show $M, \langle t, i \rangle \models A\mathcal{U}B \Leftrightarrow M_i, t \models (A\mathcal{U}B)_i$ we have two cases:
  - If $\exists j >_I i. (\sigma(B_j \wedge \mathcal{H}A_j, M_j) \wedge \forall k. (i <_I k <_I j) \rightarrow \neg\sigma(\neg A_k, M_k))$ (assumption) then $\exists j >_I i$ such that $\forall k. (i <_I k <_I j) \rightarrow \neg\sigma(\neg A_k, M_k)$

$\Rightarrow \forall k.(i <_I k <_I j) \rightarrow \neg\exists t \in T_k.M_k, t \models \neg A_k$ (by def. 2.8-satisfied)
$\Rightarrow \forall k.(i <_I k <_I j) \rightarrow \forall t \in T_k.M_k, t \models A_k$
$\Rightarrow \forall k.(i <_I k <_I j) \rightarrow \forall t \in T_k.M, \langle t, k\rangle \models A$ (by IH) (1)
and
$\sigma(B_j \wedge \mathcal{H}A_j, M_j)$
$\Rightarrow \exists w \in T_j.M_j, w \models B_j \wedge \mathcal{H}A_j$ (by def. 2.8-satisfied) (2)

* $\Rightarrow M_j, w \models B_j$ (by 2)
  $\Rightarrow M, \langle w, j\rangle \models B$ (by IH) (3)
* $\Rightarrow M_j, w \models \mathcal{H}A_j$ (by 2)
  $\Rightarrow \forall t <_j w.M_j, t \models A_j$
  $\Rightarrow \forall t.\langle t, j\rangle < \langle w, j\rangle \rightarrow M, \langle t, j\rangle \models A$(by IH, def. of lexicographic sum) (4)

$\Rightarrow \exists w \in T_j.(\forall t.\langle t, j\rangle < \langle w, j\rangle \rightarrow M, \langle t, j\rangle \models A \bigwedge M, \langle w, j\rangle \models B)$ (3,4) (5)
Subsequently we have
$M_i, t \models (A\mathcal{U}B)_i$
$\Leftrightarrow M_i, t \models A_i\mathcal{W}B_i$ (by assumption, def. 3.19-localization)
$\Leftrightarrow \exists u.(u >_i t \wedge M_i, u \vDash B_i \wedge \forall v.(t <_i v <_i u \rightarrow M_i, v \vDash A_i)) \vee \forall v >_i t.M_i, v \vDash A_i$ (by semantics of $\mathcal{W}$)
$\Leftrightarrow$

* $\exists u.(u >_i t \wedge M_i, u \vDash B_i \wedge \forall v.(t <_i v <_i u \rightarrow M_i, v \vDash A_i))$
  $\Leftrightarrow \exists u.(\langle u, i\rangle > \langle t, i\rangle \wedge M, \langle u, i\rangle \vDash B \wedge \forall v.(\langle t, i\rangle < \langle v, i\rangle < \langle u, i\rangle \rightarrow M, \langle v, i\rangle \vDash A))$ (by IH, def. 2.11-lexicographic sum)
  $\Leftrightarrow M, \langle t, i\rangle \models A\mathcal{U}B$ (by semantics of $\mathcal{U}$, def. of lexicographic sum)

or

* $\forall v >_i t.M_i, v \vDash A_i$
  $\Leftrightarrow \forall v.(\langle v, i\rangle > \langle t, i\rangle \rightarrow M, \langle v, i\rangle \vDash A)$ (by IH,def. 2.11-lexicographic sum)
  $\Leftrightarrow \forall v.(\langle v, i\rangle > \langle t, i\rangle \rightarrow M, \langle v, i\rangle \vDash A) \bigwedge \forall k.(i <_I k <_I j \rightarrow \forall t \in T_k.M, \langle t, k\rangle \models A) \bigwedge \exists w \in T_j.(\forall t.(\langle t, j\rangle < \langle w, j\rangle \rightarrow M, \langle t, j\rangle \models A) \bigwedge M, \langle w, j\rangle \models B)$ (by 1 , 5, assumption)
  $\Leftrightarrow \exists j \in I.\exists w \in T_j.(\langle w, j\rangle > \langle t, i\rangle \bigwedge M, \langle w, j\rangle \models B \bigwedge \forall\langle v, k\rangle.(\langle t, i\rangle < \langle v, k\rangle < \langle w, j\rangle \rightarrow M, \langle v, k\rangle \models A))$ (by def. of lexicographic sum, assumption)
  $\Leftrightarrow M, \langle t, i\rangle \models A\mathcal{U}B$ (by semantics of $\mathcal{U}$, def. of lexicographic sum)

– otherwise $\neg\exists j >_I i.(\sigma(B_j \wedge \mathcal{H}A_j, M_j) \wedge \forall k.(i <_I k <_I j) \rightarrow \neg\sigma(\neg A_k, M_k))$ (assumption)
Subsequently we have
$M, \langle t, i\rangle \models A\mathcal{U}B$
$\Leftrightarrow \exists u.(u >_i t \wedge M, \langle u, i\rangle \vDash B \wedge \forall v.(t <_i v <_i u \rightarrow M, \langle v, i\rangle \vDash A))$ (by semantics of $\mathcal{U}$, def. of lexicographic sum, assumption)
$\Leftrightarrow \exists u.(u >_i t \wedge M_i, u \vDash B_i \wedge \forall v.(t <_i v <_i u \rightarrow M_i, v \vDash A_i))$ (by IH)
$\Leftrightarrow M_i, t \models A_i\mathcal{U}B_i$ (by semantics of $\mathcal{U}$)
$\Leftrightarrow M_i, t \models (A\mathcal{U}B)_i$ (by assumption and def. of localization)

- The case for $M, \langle t, i\rangle \models A\mathcal{S}B \Leftrightarrow M_i, t \models (A\mathcal{S}B)_i$ is entirely temporally symmetrical to (replace $\mathcal{U}$ with $\mathcal{S}$ and $<$ with $>$).
- Showing $M, \langle t, i\rangle \models A\mathcal{W}B \Leftrightarrow M_i, t \models (A\mathcal{W}B)_i$ is similar to the $\mathcal{U}$ case.
- The case for $M, \langle t, i\rangle \models A\mathcal{Z}B \Leftrightarrow M_i, t \models (A\mathcal{Z}B)_i$ is entirely temporally symmetrical.

□

From this lemma we have:

Theorem 3.21. *(localization Theorem for $L(\mathcal{U},\mathcal{S})$): Given a linear order $(I,<_I)$, a model $M=\sum_{i\in I}M_i$, and a formula $A\in L(\mathcal{U},\mathcal{S})$:*

$$A \text{ is satisfied in } M=\sum_{i\in I} M_i \Longleftrightarrow \exists i\in I. A_i \text{is satisfied in } M_i$$

Proof. : identical to the proof of theorem 3.3, but using lemma 3.20. □

Having generalised the localization theorem to $L(\mathcal{U},\mathcal{S})$, the next step is to generalise the definition of $S(A)$ (definition 3.4). Unfortunately definition 3.4 will not work for $L(\mathcal{U},\mathcal{S})$, because the minimal closure of A under subformulas and localisation does not contain the formulas necessary to determine the context for a given formula. Consider for example the formula $a\mathcal{U}b$. The minimal closure of $a\mathcal{U}b$ under subformulas and localisation is $\{a,b,a\mathcal{U}b,a\mathcal{W}b\}$. However in order to localise $a\mathcal{U}b$ at any model M_i in the context of $\sum_{i\in I}M_i$ we may need to know $\sigma(b\wedge\mathcal{H}a,M_j)$ for some $j\in I$. Thus if we want this information to get inductively computed for each submodel we need to amend the definition of $S(A)$ to include these additional context formulas.

Definition 3.22. $(S(A))$: for a formula $A\in L(\mathcal{U},\mathcal{S})$, $S(A)$ is defined as the smallest set such that:

- $A\in S(A)$
- if $B\in S(A)$ then $subformulas(B)\subseteq S(A)$
- for any model $\sum_{i\in I}M_i$, linear order $(I,<_I)$, $\forall B\in S(A).\forall i\in I. B_i\in S(A)$
- if $C\mathcal{U}D\in S(A)$ or $C\mathcal{W}D\in S(A)$, then $D\wedge\mathcal{H}C\in S(A)$ and $\neg C\in S(A)$
- if $C\mathcal{S}D\in S(A)$ or $C\mathcal{Z}D\in S(A)$, then $D\wedge\mathcal{G}C\in S(A)$ and $\neg C\in S(A)$

We do not modify the definition of $\Sigma_A(M)$ beyond generalising it to apply to $A\in L(\mathcal{U},\mathcal{S})$, and to use the set $S(A)$ of definition 3.22.

Definition 3.23. $(\Sigma_A(M))$: Given a formula $A\in L(\mathcal{U},\mathcal{S})$ and a model M, we define

$$\Sigma_A(M)=\{B\in S(A): B \text{ is satisfied in } M\}$$

As for $L(\mathcal{F},\mathcal{P})$, using these definitions, we can rephrase theorem 3.21 by defining $\Sigma_A(\sum_{i\in I}M_i)$ in terms of $\Sigma_A(M_i)$.

Proposition 3.24. (definition of $\Sigma_A(\sum_{i\in I}M_i)$*): for any lexicographic sum $\sum_{i\in I}M_i$*

$$\Sigma_A(\sum_{i\in I}M_i)=\{B\in S(A):\exists i\in I. B_i\in\Sigma_A(M_i)\}$$

Proof. :

$$\begin{aligned}
\Sigma_A(\sum_{i\in I}M_i) &= \{B\in S(A): B \text{ is satisfied in } M\} \text{ (by definition)}\\
&= \{B\in S(A):\exists i\in I. B_i \text{ is satisfied in } M_i\} \text{ (by Localization Theorem-3.21)}\\
&= \{B\in S(A):\exists i\in I. B_i\in\Sigma_A(M_i)\} \text{ (by definition of } \Sigma_A(M_i), \text{ as } B_i\in S(A)\\
&\quad \text{by definition of } S(A))
\end{aligned}$$

□

The algorithm

Having generalised the previous definitions to $L(\mathcal{U},\mathcal{S})$, the idea of the algorithm for inductively computing $\Sigma_A(M)$ for any model expression M will be exactly the same as for $L(\mathcal{F},\mathcal{P})$. Thus we need to specify an algorithm for each model expression $M = \_(M_0...M_n)$ (where $\_$ is one of the four operators and $M_0...M_n$ are the input models) which given a formula $A \in L(\mathcal{F},\mathcal{P})$ and $\Sigma_A(M_i)$ for $i \in \{0...n\}$can construct $\Sigma_A(M)$.

Computing $\Sigma_A(a)$

Lemma 3.25. *($\Sigma_A(a)$)There is an algorithm which given any formula $A \in L(\mathcal{U},\mathcal{S})$, and any single point model a computes $\Sigma_A(a)$.*

Proof. : Firstly we need to compute $S(A)$ according to definition 3.22. Recall that $\mathcal{G}A$, $\mathcal{H}A$ abbreviate $A\mathcal{W}\bot$, $A\mathcal{Z}\bot$ respectively. Here, a possible localization of a formula is an edited version of it where arbitrarily many of it's subformulas of the form $A\mathcal{U}B$ are replaced with $A\mathcal{W}B$ (and vice versa), and arbitrarily many of those of the form $A\mathcal{S}B$ are replaced with $A\mathcal{Z}B$ (and vice versa). Initialise $S(A)$ to just be the subformulas of A. Then iterate through each formula of $S(A)$, adding formulas to it according to the definition. As soon as an iteration adds no formulas $S(A)$ has been computed. The process is bound to terminate as $S(A)$ is by definition finite. Then, for each formula $B \in S(A)$ evaluate it at 0:

- if B is $\top$ return true
- if B is an atom $p \in \mathcal{L}$ then return true if $p \in a$ and false otherwise.
- if $B = \neg C$ then recursively call the algorithm on C, returning the negation of its result.
- if $B = C \wedge D$ recursively call the algorithm on C and D, returning true if both recursive calls return true and false otherwise.
- if $B = C\mathcal{S}D$ or $B = C\mathcal{U}D$ return false
- if $B = C\mathcal{Z}D$ or $B = C\mathcal{W}D$ return true

If the formula evaluates to true add it to $\Sigma_A(a)$. When there are no more formulas in $S(A)$ return $\Sigma_A(a)$.

□

In the sections that follow, we will describe how to compute $\Sigma_A(\_(M_0...M_n))$ in terms of $\Sigma_A(M_0)...\Sigma_A(M_n)$ for each of the four operators.

Remark 3.26. **IMPORTANT NOTE TO READER**: The following sections (through to the end of the chapter 3) contain a typographical error which was discovered very late in the writing process, and has as such been left uncorrected for fear that correcting it may introduce more errors. The error is that although we are implicitly assuming that the overall formula for which we are trying to determine satisfiability is called A (as indicated by the references to $\Sigma_A(M)$ in the localisation conditions) we also refer to one of the current subformulas as A (e.g. in $(A \wedge B)_i$). Thus all references to $\Sigma_A(M)$ and $S(A)$ are referring to the overall formula A, not the unfortunately named subformula. We apologise in advance for this error, and hope that it does not lead to confusion.

Computing $\Sigma_A(M_0 + M_1)$

By proposition 3.24 and definition of $M_0 + M_1$ we have $\Sigma_A(M_0 + M_1) = \{C \in S(A) : \ C_0 \in \Sigma_A(M_0) \vee C_1 \in \Sigma_A(M_1)\}$.

Furthermore, for every $C \in S(A)$ we can compute the localizations C_0, C_1 of C inductively, by specialising the definition of localisation (definition 3.19) to the model $M = M_0 + M_1$:

- If B is atomic then $C_0 = C_1 = C$
- for $i \in \{0, 1\}$, $(\neg A)_i = \neg A_i$
- for $i \in \{0, 1\}$, $(A \wedge B)_i = A_i \wedge B_i$
- $(A\mathcal{U}B)_0 = \begin{cases} A_0\mathcal{W}B_0 & \text{if } B_1 \wedge \mathcal{H}A_1 \in \Sigma_A(M_1) \\ A_0\mathcal{U}B_0 & \text{otherwise} \end{cases}$
- $(A\mathcal{U}B)_1 = A_1\mathcal{U}B_1$
- $(A\mathcal{W}B)_0 = \begin{cases} A_i\mathcal{W}B_i & \text{if } B_1 \wedge \mathcal{H}A_1 \in \Sigma_A(M_1) \\ & \text{or } \neg A_1 \notin \Sigma_A(M_1) \\ A_i\mathcal{U}B_i & \text{otherwise} \end{cases}$
- $(A\mathcal{W}B)_1 = A_1\mathcal{W}B_1$
- The definition of $(A\mathcal{S}B)_i$ for $i \in \{0, 1\}$ is symmetric to that of $(A\mathcal{U}B)_i$
- The definition of $(A\mathcal{Z}B)_i$ for $i \in \{0, 1\}$ is symmetric to that of $(A\mathcal{W}B)_i$

Notice that the localisation context conditions $\sigma(B_i \wedge \mathcal{H}A_i, M_i)$ become $B_i \wedge \mathcal{H}A_i \in \Sigma_A(M_i)$, as $\sigma(B_i \wedge \mathcal{H}A_i, M_i) \iff B_i \wedge \mathcal{H}A_i \in \Sigma_A(M_i)$ by definition of $\Sigma_A(M_i)$, because $B_i \wedge \mathcal{H}A_i \in S(A)$ by definition of $S(A)$ ($A_i\mathcal{U}B_i$ is a possible localisation of a formula in $S(A)$). By the same argument we have $\neg\sigma(\neg A_i, M_i) \iff \neg A_i \notin \Sigma_A(M_i)$

From the above it follows that

Lemma 3.27. *($\Sigma_A(M_0 + M_1)$): There is an algorithm which for any models M_0, M_1 given as input a formula $A \in L(\mathcal{U}, \mathcal{S})$ and $\Sigma_A(M_0)$, $\Sigma_A(M_1)$ computes $\Sigma_A(M_0 + M_1)$.*

Computing $\Sigma_A(< M_1, ..., M_n >)$

By the definition of shuffle as a lexicographic sum where $I = \mathbb{Q}$, and proposition 3.24 we have

$$\Sigma_A(< M_1, ..., M_n >) = \{B \in S(A) : \ \exists i \in \mathbb{Q}.B_i \in \Sigma_A(M_i)\}$$

and thus should in theory compute B_i for every $i \in \mathbb{Q}$. This is in fact unnecessary as a result of the following Lemma.

Lemma 3.28. *(Shuffle localization lemma for $L(\mathcal{U}, \mathcal{S})$): For any formula $C \in L(\mathcal{U}, \mathcal{S})$:*

$$\forall i, j \in \mathbb{Q}.(C_i = C_j)$$

Proof. : Pick arbitrary $C \in L(\mathcal{F}, P)$, $i, j \in \mathbb{Q}$ Then:

- If B is atomic then by definition of localization$C_i = C_j = C$

(IH): assume the lemma for formulas A, B.

- $(\neg A)_i$
  $= \neg A_i$ (by def. 3.19-loc.)
  $= \neg A_j$ (by IH)
  $= (\neg A)_j$ (by def. of loc.)

- $(A \wedge B)_i$
  $= A_i \wedge B_i$ (by def. of loc.)
  $= A_j \wedge B_j$ (by IH)
  $= (A \wedge B)_j$ (by def. of loc.)

- To show $(A\mathcal{U}B)_i = (A\mathcal{U}B)_j$we have two cases:
  Assume $\exists h >_I i.(\sigma(B_h \wedge \mathcal{H}A_h, M_h) \wedge \forall k.(i <_I k <_I h) \rightarrow \neg\sigma(\neg A_k, M_k))$ (assumption)

  1. $\exists x \in \{0, 1, ..., n\}.M_x = M_h$ (by def. of shuffle)
  2. $\exists r >_I j.M_r = M_x$(by definition of shuffle)
  3. $\sigma(B_h \wedge \mathcal{H}A_h, M_h)$ (by assumption)
     $\Rightarrow \exists t \in T_h.(M_h, t \models B_h$ and $M_h, t \models \mathcal{H}A_h)$ (by def.2.8 :satisfied, semantics of $\wedge$)
     $\Rightarrow \exists t \in T_r.(M_r, t \models B_h$ and $M_r, t \models \mathcal{H}A_h)$ (by 2,3 $M_r = M_h$)
     $\Rightarrow \exists t \in T_r.(M_r, t \models B_r$ and $M_r, t \models \mathcal{H}A_h)$ (by IH $B_h = B_r$)
     $\Rightarrow \exists t \in T_r.(M_r, t \models B_r$ and $\forall v <_r t.(M_r, v \models A_h)$ (by def. of $\mathcal{H}$)
     $\Rightarrow \exists t \in T_r.(M_r, t \models B_r$ and $\forall v <_r t.(M_r, v \models A_r)$ (by IH $A_h = A_r$)
     $\Rightarrow \exists t \in T_r.(M_r, t \models B_r$ and $M_r, t \models \mathcal{H}A_r)$ (by def. of $\mathcal{H}$)
     $\Rightarrow \sigma(B_r \wedge \mathcal{H}A_r, M_r)$ (by def.2.8 :satisfied, semantics of $\wedge$)
  4. $\forall k.(i <_I k <_I j) \rightarrow \neg\sigma(\neg A_k, M_k)$ (by assumption)
  5. Pick arbitrary $s \in \mathbb{Q}$
     (a) $\exists h \in \{0, 1, ..., n\}.M_h = M_s$ (by def. of shuffle)
     (b) $\exists k \in \mathbb{Q}.(i <_I k <_I j \wedge M_k = M_h$ (by def. of shuffle)
     (c) $M_k = M_s$(by a, b)
     (d) $\neg\sigma(\neg A_k, M_k)$ (by 5)
     (e) $\neg\sigma(\neg A_s, M_s)$ (by c,d, as assuming $\sigma(\neg A_s, M_s)$ implies $\sigma(\neg A_k, M_k)$ by IH and def. of satisfied, contradiction)
  6. $\forall k \in \mathbb{Q}.\neg\sigma(\neg A_k, M_k)$ (by 6.e, as s was arbitrary)
  7. $\forall k.(j <_I k <_I r) \rightarrow \neg\sigma(\neg A_k, M_k)$ (by 7)
  8. $\exists r >_I j.(\sigma(B_r \wedge \mathcal{H}A_r, M_r) \wedge \forall k.((j <_I k <_I r) \rightarrow \neg\sigma(\neg A_k, M_k)))$ (by 3, 4, 8)
  9. $(A\mathcal{U}B)_j = A_j\mathcal{W}B_j$ (by 9)
  10. $(A\mathcal{U}B)_i = A_i\mathcal{W}B_i$ (by def. of loc., assumption)
  11. $(A\mathcal{U}B)_i = (A\mathcal{U}B)_i$ (by 9,10 and IH)

  Otherwise $\neg\exists h >_I i.(\sigma(B_h \wedge \mathcal{H}A_h, M_h) \wedge \forall k.(i <_I k <_I h) \rightarrow \neg\sigma(\neg A_k, M_k))$ (assumption)

  1. $(A\mathcal{U}B)_i = A_i\mathcal{U}B_i$ (by assumption, def. of loc.)
  2. Assume for contradiction that $\exists h' >_I j.(\sigma(B_{h'} \wedge \mathcal{H}A_{h'}, M_{h'}) \wedge \forall k.(j <_I k <_I h') \rightarrow \neg\sigma(\neg A_k, M_k))$. Then by applying same argument as above (swapping i and j) we can show $\exists h >_I i.(\sigma(B_h \wedge \mathcal{H}A_h, M_h) \wedge \forall k.(i <_I k <_I h) \rightarrow \neg\sigma(\neg A_k, M_k))$, which contradicts our assumption. Subsequently $\neg\exists h' >_I j.(\sigma(B_{h'} \wedge \mathcal{H}A_{h'}, M_{h'}) \wedge \forall k.(j <_I k <_I h') \rightarrow \neg\sigma(\neg A_k, M_k))$.

3. $(A\mathcal{U}B)_j = A_j\mathcal{U}B_j$ (by 2, def. of loc.)
4. $(A\mathcal{U}B)_i = (A\mathcal{U}B)_j$ (by 1,3 and IH)

- The case for $(A\mathcal{U}B)_i = (A\mathcal{U}B)_j$ is similar
- The case for $(A\mathcal{S}B)_i = (A\mathcal{S}B)_j$ is symmetric to that for $(A\mathcal{U}B)_i = (A\mathcal{U}B)_j$
- The case for $(A\mathcal{Z}B)_i = (A\mathcal{Z}B)_j$ is symmetric to that for $(A\mathcal{W}B)_i = (A\mathcal{W}B)_j$

□

As a result of the Lemma we can simplify the definition of localization for formulas $B \in S(A)$, in terms of the single localization B_λ that we need to compute:

- If B is atomic then $B_\lambda = B$
- $(\neg B)_\lambda = \neg B_\lambda$
- $(A \wedge B)_\lambda = A_\lambda \wedge B_\lambda$
- $(A\mathcal{U}B)_\lambda = \begin{cases} A_\lambda\mathcal{W}B_\lambda & \text{if } B_\lambda \in \bigcup_{i=0}^{n} \Sigma_A(M_i) \wedge \neg A_\lambda \notin \bigcup_{i=0}^{n} \Sigma_A(M_i) \\ A_\lambda\mathcal{U}B_\lambda & \text{otherwise} \end{cases}$
- $(A\mathcal{W}B)_\lambda = \begin{cases} A_\lambda\mathcal{W}B_\lambda & \text{if } \neg A_\lambda \notin \bigcup_{i=0}^{n} \Sigma_A(M_i) \\ A_\lambda\mathcal{U}B_\lambda & \text{otherwise} \end{cases}$
  $(A\mathcal{S}B)_\lambda = \begin{cases} A_\lambda\mathcal{Z}B_\lambda & \text{if } B_\lambda \in \bigcup_{i=0}^{n} \Sigma_A(M_i) \wedge \neg A_\lambda \notin \bigcup_{i=0}^{n} \Sigma_A(M_i) \\ A_\lambda\mathcal{S}B_\lambda & \text{otherwise} \end{cases}$
- $(A\mathcal{Z}B)_\lambda = \begin{cases} A_\lambda\mathcal{Z}B_\lambda & \text{if } \neg A_\lambda \notin \bigcup_{i=0}^{n} \Sigma_A(M_i) \\ A_\lambda\mathcal{S}B_\lambda & \text{otherwise} \end{cases}$

Note the modified localization conditions as a result of

$$\exists j >_I i.(\sigma(B_j \wedge \mathcal{H}A_j, M_j) \wedge \forall k.(i <_I k <_I j) \rightarrow \neg\sigma(\neg A_k, M_k))$$

$$\iff \exists j >_I i.(\sigma(B_\lambda \wedge \mathcal{H}A_\lambda, M_j) \wedge \forall k.(i <_I k <_I j) \rightarrow \neg\sigma(\neg A_\lambda, M_k))$$

(by lemma 3.28)

$$\iff \exists i \in \{0...n\}.\sigma(B_\lambda, M_i) \wedge \forall i \in \{0...n\}.\neg\sigma(\neg A_\lambda, M_i)$$

(by definition of shuffle)

$$\iff B_\lambda \in \bigcup_{i=0}^{n} \Sigma_A(M_i) \wedge \neg A_\lambda \notin \bigcup_{i=0}^{n} \Sigma_A(M_i)$$

(by def of $\Sigma_A(M_i)$, as $B_\lambda \in S(A)$, $\neg A_\lambda \in S(A)$)

Subsequently, as for L(F,P) we have

$$\begin{aligned}
\Sigma_A(< M_1, ..., M_n >) &= \{B \in S(A) : \exists i \in \mathbb{Q}.B_i \in \Sigma_A(M_i)\} \\
&= \{B \in S(A) : \exists i \in \mathbb{Q}.B_\lambda \in \Sigma_A(M_i)\} \text{ (by Lemma 3.28)} \\
&= \{B \in S(A) : B_\lambda \in \bigcup_{i\in\mathbb{Q}} \Sigma_A(M_i)\} \\
&= \{B \in S(A) : B_\lambda \in \bigcup_{i=0}^{n} \Sigma_A(M_i)\} \text{ (as } \bigcup_{i\in\mathbb{Q}} \Sigma_A(M_i) = \bigcup_{i=0}^{n} \Sigma_A(M_i) \\
&\quad \text{by definition of shuffle)}
\end{aligned}$$

and clearly it follows that:

Lemma 3.29. *($\Sigma_A(< M_1, ..., M_n >)$): There is an algorithm which for any models M_1,...,M_n given as input a formula $A \in L(\mathcal{U}, \mathcal{S})$ and $\Sigma_A(M_1)$,...,$\Sigma_A(M_n)$ computes $\Sigma_A(< M_1, ..., M_n >)$.*

Computing $\Sigma_A(\omega M_0)$

We start by specialising the definition of localisation to the model ωM_0. Recall that for ωM_0 the index ordering is $I = \mathbb{N}$. Thus for $C \in S(A)$, according to definition 3.19 C_n is defined for $n \geq 0$:

- If C is atomic, $\forall i \in I.C_i = C$
- $(\neg A)_n = \neg A_n$
- $(A \wedge B)_n = A_n \wedge B_n$
- $(A\mathcal{U}B)_n = \begin{cases} A_n\mathcal{W}B_n & \text{if } \exists m >_I n.(B_m \wedge \mathcal{H}A_m \in \Sigma_A(M_0) \wedge \forall k.(n <_I k <_I m) \rightarrow \neg A_k \notin \Sigma_A(M_0)) \\ A_n\mathcal{U}B_n & \text{otherwise} \end{cases}$
- $(A\mathcal{W}B)_n = \begin{cases} A_n\mathcal{W}B_n & \text{if } \exists m >_I n.(B_m \wedge \mathcal{H}A_m \in \Sigma_A(M_0) \wedge \forall k.(n <_I k <_I m) \rightarrow \neg A_k \notin \Sigma_A(M_0)) \\ & \text{or } \forall k >_I n.\neg A_k \notin \Sigma_A(M_0) \\ A_n\mathcal{U}B_n & \text{otherwise} \end{cases}$
- $(A\mathcal{S}B)_n = \begin{cases} A_n\mathcal{Z}B_n & \text{if } \exists m <_I n.(B_m \wedge \mathcal{G}A_m \in \Sigma_A(M_0) \wedge \forall k.(m <_I k <_I n) \rightarrow \neg A_k \notin \Sigma_A(M_0)) \\ A_n\mathcal{S}B_n & \text{otherwise} \end{cases}$
- $(A\mathcal{Z}B)_n = \begin{cases} A_n\mathcal{Z}B_n & \text{if } \exists m <_I n.(B_m \wedge \mathcal{G}A_m \in \Sigma_A(M_0) \wedge \forall k.(m <_I k <_I n) \rightarrow \neg A_k \notin \Sigma_A(M_0)) \\ & \text{or } \forall k <_I n.\neg A_k \notin \Sigma_A(M_0) \\ A_n\mathcal{S}B_n & \text{otherwise} \end{cases}$

Notice that as in the case of computing $\Sigma_A(M_0 + M_1)$, $\sigma(B_m \wedge \mathcal{H}A_m, M_m)$ is replaced with $B_m \wedge \mathcal{H}A_m \in \Sigma_A(M_0)$ (as $\forall n \in \mathbb{N}.M_n = M_0$, and $\sigma(B_m \wedge \mathcal{H}A_m, M_m) \Longleftrightarrow B_m \wedge \mathcal{H}A_m \in \Sigma_A(M_0)$ as $B_m \wedge \mathcal{H}A_m \in S(A)$). By proposition 3.24 we have $\Sigma_A(\omega M_0) = \{B \in S(A) : \exists n \in \mathbb{N}.B_n \in \Sigma_A(M_0)\}$. According to this definition in order to compute $\Sigma_A(\omega M_0)$ we potentially need to compute the localization B_n of B for all natural numbers. As in the $L(\mathcal{F}, \mathcal{P})$ algorithm this is in fact not necessary, as lemma 3.12 can be generalised to $L(\mathcal{U}, \mathcal{S})$.

Definition 3.30. ($d(F)$ for $F \in L(\mathcal{U}, S)$): For a formula $B \in L(\mathcal{U}, \mathcal{S})$, we define $d(B)$ to be the maximum depth of nesting of $\mathcal{S}/\mathcal{Z}$ in B:

- If B is atomic then $d(B) = 0$
- $d(\neg B) = d(B)$
- $d(A \wedge B) = d(A\mathcal{U}B) = d(A\mathcal{W}B) = max(d(A), d(B))$
- $d(A\mathcal{S}B) = d(A\mathcal{Z}B) = max(d(A), d(B)) + 1$

Lemma 3.31. *(ω localization lemma for $L(\mathcal{U}, \mathcal{S})$): given a model M, a formula A, and $C \in S(A)$,*

$$\forall n \geq d(C).C_n = C_{d(C)}$$

i.e. the localizations of C converge at the sub-model $M_{d(C)}$.

Proof. : We prove the lemma by induction on the structure of C. The cases where C is boolean are identical to those in the proof of the ω localization lemma for $L(\mathcal{F}, \mathcal{P})$ (lemma 3.12), so we consider here only the cases for $C = A\mathcal{U}B$, $C = A\mathcal{W}B$, $C = A\mathcal{S}B$, $C = A\mathcal{Z}B$. Assume inductively that the lemma holds for A, B (IH).

- $C = A\mathcal{U}B$: Pick arbitrary $n \geq d(A\mathcal{U}B) = max(d(A), d(B))$ (assumption).To show: $(A\mathcal{U}B)_n = (A\mathcal{U}B)_{d(A\mathcal{U}B)}$

  1. We first show that the localization context conditions are equivalent:
     $\exists m > n.(B_m \wedge \mathcal{H}A_m \in \Sigma_A(M_0) \wedge \forall k.(n < k < m) \rightarrow \neg A_k \notin \Sigma_A(M_0))$
     $\Leftrightarrow \exists m > n.(B_{d(B)} \wedge \mathcal{H}A_{d(A)} \in \Sigma_A(M_0) \wedge \forall k.(n < k < m) \rightarrow \neg A_{d(A)} \notin \Sigma_A(M_0))$
     (by IH, as $m, k > n \geq max(d(A), d(B))$, so $m > d(A)$, $m > d(B)$, and $k > d(A)$)
     $\Leftrightarrow \exists m > n.(B_{d(A\mathcal{U}B)} \wedge \mathcal{H}A_{d(A\mathcal{U}B)} \in \Sigma_A(M_0) \wedge \forall k.(n < k < m) \rightarrow \neg A_{d(A\mathcal{U}B)} \notin \Sigma_A(M_0))$
     (by IH, as $d(A\mathcal{U}B) \geq d(A)$, $d(A\mathcal{U}B) \geq d(B)$)
     $\Leftrightarrow \exists m > d(A\mathcal{U}B).(B_{d(A\mathcal{U}B)} \wedge \mathcal{H}A_{d(A\mathcal{U}B)} \in \Sigma_A(M_0) \wedge \forall k.(d(A\mathcal{U}B) < k < m) \rightarrow \neg A_k \notin \Sigma_A(M_0))$
     (as $n \geq d(A\mathcal{U}B) \geq d(A)$, so $A_k = A_{d(A)}$ and $A_{d(A\mathcal{U}B)} = A_{d(A)}$ by IH)
     $\Leftrightarrow \exists m > d(A\mathcal{U}B).(B_m \wedge \mathcal{H}A_m \in \Sigma_A(M_0) \wedge \forall k.(d(A\mathcal{U}B) < k < m) \rightarrow \neg A_k \notin \Sigma_A(M_0))$
     (by IH, as $m > d(A\mathcal{U}B) \geq d(A)$, $m > d(A\mathcal{U}B) \geq d(B)$)
  2. $A_n\mathcal{U}B_n = A_{d(A)}\mathcal{U}B_{d(B)} = A_{d(A\mathcal{U}B)}\mathcal{U}B_{d(A\mathcal{U}B)}$ (as $n \geq d(A\mathcal{U}B) \geq d(A)$, $n \geq d(A\mathcal{U}B) \geq d(B)$ so by IH $A_n = A_{d(A)} = A_{d(A\mathcal{U}B)}$, $B_n = B_{d(B)} = B_{d(A\mathcal{U}B)}$)
  3. $A_n\mathcal{W}B_n = A_{d(A)}\mathcal{W}B_{d(B)} = A_{d(A\mathcal{U}B)}\mathcal{W}B_{d(A\mathcal{U}B)}$ (by same argument as 2)
  4. $(A\mathcal{U}B)_n = (A\mathcal{U}B)_{d(A\mathcal{U}B)}$ (by 1-3)

- $C = A\mathcal{S}B$: Pick arbitrary $n \geq d(A\mathcal{S}B) = max(d(A), d(B)) + 1$ (assumption).To show: $(A\mathcal{S}B)_n = (A\mathcal{S}B)_{d(A\mathcal{S}B)}$

  1. $n \geq d(A\mathcal{S}B) > d(B)$ (by assumption)
  2. $n \geq d(A\mathcal{S}B) > d(A)$ (by assumption)
  3. $A_n\mathcal{S}B_n = A_{d(A)}\mathcal{S}B_{d(B)} = A_{d(A\mathcal{S}B)}\mathcal{S}B_{d(A\mathcal{S}B)}$ (as $n \geq d(A\mathcal{S}B) > d(A)$, $n \geq d(A\mathcal{S}B) > d(B)$ so by IH $A_n = A_{d(A)} = A_{d(A\mathcal{S}B)}$, $B_n = B_{d(B)} = B_{d(A\mathcal{S}B)}$)
  4. $A_n\mathcal{Z}B_n = A_{d(A\mathcal{Z}B)}\mathcal{Z}B_{d(A\mathcal{Z}B)}$ (by same argument as 3)
  5. We now show that the localization conditions are equivalent. For any $i \in \mathbb{N}$, let $X(i) = \exists j < i.(B_j \wedge \mathcal{G}A_j \in \Sigma_A(M_0) \wedge \forall k.(j < k < i) \rightarrow \neg A_k \notin \Sigma_A(M_0))$. Let $m = max(d(A), d(B))$, and $m' = d(A\mathcal{S}B) = m + 1$.
     To show: $\forall i \geq m'.X(i) \Leftrightarrow X(m')$. Pick arbitrary $i \geq m'$.

- $X(i) \Rightarrow X(m')$: Assume $X(i)$. Then $\exists j < i$ with the properties described in $X(i)$. There are two cases for j:
  * $j \leq m$: Then we trivially have $X(m')$ by just using the same j.
  * $j > m$: so $m' \leq j < i$. $B_j \wedge \mathcal{G}A_j \in \Sigma_A(M_0)$. As $j > m \geq d(A)$, $j > m \geq d(B)$, this means that by IH $B_j = B_m = B_{d(B)}$, and $A_j = A_m = A_{d(A)}$. Hence $B_m \wedge \mathcal{G}A_m \in \Sigma_A(M_0)$. Additionally, as $m' = m + 1$ we have $\forall k.(m < k < m') \rightarrow \neg A_k \notin \Sigma_A(M_0)$. Thus $X(m')$ holds,as we have $\exists m < m'.(B_m \wedge \mathcal{G}A_m \in \Sigma_A(M_0) \wedge \forall k.(m < k < m') \rightarrow \neg A_k \notin \Sigma_A(M_0))$.
- $X(m') \Rightarrow X(i)$: Assume $X(m')$. Recall $i \geq m'$. Clearly if $i = m'$ then we have $X(i)$, so pick $i > m'$. There are two cases for j:
  * $j < m$: Then we have
    (a) $\forall k.(j < k < m') \rightarrow \neg A_k \notin \Sigma_A(M_0)$ (as we assumed $X(m')$)
    (b) $\neg A_m \notin \Sigma_A(M_0)$ (by (a), as $j < m < m'$)
    (c) $\forall k' \geq m.A_{k'} = A_m$ (by IH, as $k' \geq m \geq d(A)$)
    (d) $\forall k' \geq m.\neg A_{k'} \notin \Sigma_A(M_0)$ (by (b), (c))
    (e) $\forall k.(j < k < i) \rightarrow \neg A_k \notin \Sigma_A(M_0)$ (by (a), (d), $i \geq m'$)
    (f) $B_j \wedge \mathcal{G}A_j \in \Sigma_A(M_0)$ (as we assumed $X(m')$)
    (g) $X(i)$ (by (e),(f), using j)
  * $j = m$: $B_m \wedge \mathcal{G}A_m \in \Sigma_A(M_0)$ (as we assumed $X(m')$). Take $j' = i - 1$. Then $j' \geq m'$ (as $i > m'$ by assumption), so $j' > m$. By IH this implies $B_{j'} \wedge \mathcal{G}A_{j'} = B_m \wedge \mathcal{G}A_m$, so we have $B_{j'} \wedge \mathcal{G}A_{j'} \in \Sigma_A(M_0)$. Also trivially as $j' = i - 1$ we have $\forall k.(j' < k < i) \rightarrow \neg A_k \notin \Sigma_A(M_0)$. Subsequently $\exists j' < i.(B_{j'} \wedge \mathcal{G}A_{j'} \in \Sigma_A(M_0) \wedge \forall k.(j' < k < i) \rightarrow \neg A_k \notin \Sigma_A(M_0))$, so $X(i)$.

6. $(A\mathcal{S}B)_n = (A\mathcal{S}B)_{d(A\mathcal{S}B)}$ (by 3,4,5)

- $C = A\mathcal{W}B$: similar to $C = A\mathcal{U}B$.
- $C = A\mathcal{Z}B$: similar to $C = A\mathcal{S}B$.

□

As a result of Lemma 3.31 we have $\Sigma_A(\omega M_0) = \{C \in S(A) : \exists n \in \mathbb{N}.C_n \in \Sigma_A(M_0)\} = \{C \in S(A) : \exists n \leq d(C).C_n \in \Sigma_A(M_0)\}$. Furthermore, as a result of the Lemma we have that for $i \leq d(C)$ the localization context conditions are equivalent to the finitely computable condition given below. This can be shown by an argument very similar to that of proof of claim 3.13.

As a result of this we get a finitely (inductively) computable definition of localization for $C \in S(A)$, $i \leq d(C)$ (where $m = d(A\mathcal{U}B) = max(d(A), d(B))$):

- If C is atomic, $\forall i \in I.C_i = C$
- $(\neg A)_i = \neg A_i$
- $(A \wedge B)_i = A_i \wedge B_i$
- $(A\mathcal{U}B)_i = \begin{cases} A_i\mathcal{W}B_i & \text{if } i < m-1 \wedge \exists j.(i < j \leq m \wedge (B_j \wedge \mathcal{H}A_j \in \Sigma_A(M_0)) \\ & \wedge \forall k.(i < k < j) \rightarrow \neg A_k \notin \Sigma_A(M_0)) \\ & \text{or } i \geq m-1 \wedge (B_m \wedge \mathcal{H}A_m \in \Sigma_A(M_0)) \\ A_i\mathcal{U}B_i & \text{otherwise} \end{cases}$

- $(A\mathcal{W}B)_i$=$\begin{cases} A_i\mathcal{W}B_i & \text{if } i < m-1 \wedge \exists j.(i < j \leq m \wedge (B_j \wedge \mathcal{H}A_j \in \Sigma_A(M_0)) \\ & \wedge \forall k.(i < k < j) \rightarrow \neg A_k \notin \Sigma_A(M_0)) \\ & \text{or } i < m-1 \wedge \forall k.(i < k \leq m) \rightarrow \neg A_k \notin \Sigma_A(M_0) \\ & \text{or } i \geq m-1 \wedge (B_m \wedge \mathcal{H}A_m \in \Sigma_A(M_0) \vee \neg A_m \notin \Sigma_A(M_0)) \\ A_i\mathcal{U}B_i & \text{otherwise} \end{cases}$

- $(A\mathcal{S}B)_i$=$\begin{cases} A_i\mathcal{Z}B_i & \text{if } \exists j.(0 \leq j < i \wedge (B_j \wedge \mathcal{G}A_j \in \Sigma_A(M_0)) \wedge \forall k.(j < k < i) \rightarrow \neg A_k \notin \Sigma_A(M_0)) \\ A_i\mathcal{S}B_i & \text{otherwise} \end{cases}$

- $(A\mathcal{Z}B)_i$=$\begin{cases} A_i\mathcal{Z}B_i & \text{if } \exists j.(0 \leq j < i \wedge B_j \wedge \mathcal{G}A_j \in \Sigma_A(M_0) \wedge \forall k.(j < k < i) \rightarrow \neg A_k \notin \Sigma_A(M_0)) \\ & \text{or } \forall k < i.\neg A_k \notin \Sigma_A(M_0) \\ A_i\mathcal{S}B_i & \text{otherwise} \end{cases}$

As we have shown $\Sigma_A(\omega M_0) = \{C \in S(A) : \exists n \leq d(C).C_n \in \Sigma_A(M_0)\}$ from the above it follows that

Lemma 3.32. *($\Sigma_A(\omega M_0)$): There is an algorithm which for any model M_0 given as input a formula $A \in L(\mathcal{U}, \mathcal{S})$ and $\Sigma_A(M_0)$ computes $\Sigma_A(\omega M_0)$.*

Computing $\Sigma_A(\omega^* M_0)$

as the ω^* operator is the mirror image of the ω operator, this whole case will be the mirror image of the case above. So unsurprisingly, as in the $\Sigma_A(\omega M_0)$ case we get

Lemma 3.33. *($\Sigma_A(\omega^* M_0)$): There is an algorithm which for any model M_0 given as input a formula $A \in L(\mathcal{U}, \mathcal{S})$ and $\Sigma_A(M_0)$ computes $\Sigma_A(\omega^* M_0)$.*

Summary

Proposition 3.34. *There is an algorithm which given any formula $A \in L(\mathcal{U}, \mathcal{S})$, and any model expression M, computes $\Sigma_A(M)$.*

Proof. : By induction on the structure of M. Pick arbitrary $A \in L(\mathcal{U}, \mathcal{S})$ and model M:

- (Base Case) $M = a$: can compute $\Sigma_A(M)$ by lemma 3.25

(IH) Assume lemma holds for M_0,...,M_n, i.e. we can compute $\Sigma_A(M_i)$ for $i \in \{0..n\}$

- $M = M_0 + M_1$: can compute $\Sigma_A(M)$ by IH and lemma 3.27
- $M =< M_1, ..., M_n >$: can compute $\Sigma_A(M)$ by IH and lemma 3.29
- $M = \omega M_0$: can compute $\Sigma_A(M)$ by IH and lemma 3.32
- $M = \omega^* M_0$: can compute $\Sigma_A(M)$ by IH and lemma 3.33

□

Our overall algorithm for determining satisfiability for $A \in L(\mathcal{U}, \mathcal{S})$ then works in the same way as the one for $A \in L(\mathcal{F}, \mathcal{P})$, but using the algorithms for computing $\Sigma_A(M)$ for $A \in L(\mathcal{U}, \mathcal{S})$.

3.3 Summary

We have given an algorithm for determining the satisfiability of formulas in the logics $L(\mathcal{F}, \mathcal{P})$, and $L(\mathcal{U}, \mathcal{S})$ by using the idea of localization. Note that this approach is one way of using localizations to determine satisfiability, but it is not the only way. In chapter 5 we will consider two variations of this algorithm which use localization to determine satisfiability in a different way.

Chapter 4

Implementation of the Algorithm

In this chapter we briefly discuss our implementation of the algorithm for $L(\mathcal{U}, \mathcal{S})$.

4.1 Motivation

A question which arises naturally at this point is why did we implement the algorithm? We have defined it in full, and illustrated (by means of the various intermediary lemmas) its correctness. Thus we know that it can be implemented in theory. There are however a number of reasons why it is beneficial to actually implement it in practice. The first of these relates to analysing the algorithms performance. It's true that we can analyse the algorithm theoretically, and determine how it is likely to perform for inputs of various sizes and structures, without implementing it. However having an implementation allows us to get a much more realistic idea of the algorithm's performance. Even if it takes exponential time in theory, it may be the case that in practice its performance is reasonably good for the average input[1]. The second reason we value an implementation is that it allows us to easily illustrate the algorithm to others. The algorithm itself is fairly complex, and the length and technical detail of its description alone will doubtlessly put some readers off. On the other hand the implementation combined with its intuitive interface allows users to understand the algorithm much faster, by visualising the computation in an interactive way. Subsequently the implementation is a useful tool for demonstrating the algorithm to others.

4.2 Key Implementation Decisions

Computing localisations

The first and most important decision made with regards to implementing the algorithm was that of iteratively computing localisations, instead of doing so recursively, in order to eliminate unnecessary recomputation. Assume we are trying to compute $\Sigma_A(M)$ at a given node M. The idea is to localise the formulas of $S(A)$ *in order of increasing size,*and store the computed localisations. What this means is that whenever considering a formula C in $S(A)$, we will have already computed and stored the localisations of its subformulas (as they are in $S(A)$, and will by definition be shorter that C). Thus all we need to do to localise C at a given M_i is look up these stored localisations, and check whether the context formula for C is contained in $\Sigma_A(M_j)$ (for some child M_j of M). Having computed C_i we just need to check whether $C_i \in \Sigma_A(M_i)$,

[1] consider for example the simplex method for linear programming, which is exponential in the worst case but polynomial for the average case

where $\Sigma_A(M_i)$ is recursively computed and stored (so it is only computed once). Subsequently localisation is just determined by looking up stored results. We illustrate this idea by means of an example. Consider the node $M = M_0 + M_1$, for which we have already computed $\Sigma_A(M_0)$ and $\Sigma_A(M_1)$ and want to compute $\Sigma_A(M)$. We start iterating through the formulas C of $S(A)$. Assume that at some point $C = A\mathcal{U}B$. Recall that $\Sigma_A(M_0 + M_1) = \{C \in S(A) : \ C_0 \in \Sigma_A(M_0) \vee C_1 \in \Sigma_A(M_1)\}$. Subsequently we need to compute C_0 and C_1. By the definition of localization for $+$, we have that $(A\mathcal{U}B)_0 = \begin{cases} A_0\mathcal{W}B_0 & \text{if } B_1 \wedge \mathcal{H}A_1 \in \Sigma_A(M_1) \\ A_0\mathcal{U}B_0 & \text{otherwise} \end{cases}$. As B, A are subformulas of $C = A\mathcal{U}B$ (and thus shorter in length) we will already have computed and stored A_1, B_1, and thus all that is required to compute $B_1 \wedge \mathcal{H}A_1 \in \Sigma_A(M_1)$ is a simple lookup. Similarly A_0, B_0 have already been computed and stored, so overall we can compute $(A\mathcal{U}B)_0$ by just looking up four stored localisations, and testing set membership. The case for $(A\mathcal{U}B)_1$ by definition requires only two lookups, and the cases for other formulas are similar. In summary by caching intermediate results we can eliminate the recomputation which would be necessary to recursively compute localisations according to the definitions (albeit at the cost of additional space usage).

Computing localisations for ωM

The second design choice which significantly affects the computation is the way in which localisations are computed for models of the form ωM. Recall from section 3.2 that $\Sigma_A(\omega M_0) = \{C \in S(A) : \ \exists n \leq d(C).C_n \in \Sigma_A(M_0)\}$. Thus we need to compute $d(C)$ localisations (using the aforementioned iterative approach). Furthermore , examining the definition of localisation for ωM it appears that to compute any individual localisation C_i we may need to compute $O(d(C))$ localisations (in the cases of $\mathcal{U}$, $\mathcal{W}$, $\mathcal{S}$,$\mathcal{Z}$). Upon examining the definition carefully, it should be clear that in fact the different localisations C_i we need to compute have some overlapping computation, and so we can actually compute C_i for all $i \leq d(C)$ by just doing $O(d(C))$ lookups (the key idea is to compute C_i for decreasing i, starting with $C_{d(C)}$). As a result of this computing $\Sigma_A(\omega M_0)$ takes at most $O(d(C))$ as many steps as computing $\Sigma_A(M)$ when M is not of the form ωT.

Technologies used

We chose to implement the algorithm in Java, as we wanted to create an intuitive interface as quickly as possible. The user interface was implemented using the JUNG(Java Universal Network/Graph) graphics library, in conjunction with java's built in SWING graphics library.

4.3 Performance Analysis

In order to analyse the performance of our implementation, we start by considering the algorithm, and determining the input parameters which are likely to significantly affect the algorithm's performance. Let M, F be the input model and formula respectively. We identify the following parameters:

1. $|S(A)|$, $|F|$: Clearly the key parameter affecting the performance of the algorithm is $|S(A)|$, as the size of all intermediate sets computed is $O(|S(A)|)$. Examining the definition of $S(A)$ we notice that :

   (a) $|S(A)|$ is exponential in the number of temporal operators in F: we see that the number of possible localisation of a formula F is exponential in the number of tem-

poral operators F contains. By the definition of $S(A)$, this in turn implies that $S(A)$ will be of size exponential in the number of temporal operators F contains.

(b) $|S(A)|$ is polynomial in the size of F if F contains a bounded number of temporal operators.

2. The size of M for a given F.

3. $d(F)$, number of ω's in M: As a result of the previous section we know that computing $\Sigma_A(\omega M_i)$ takes $O(d(F))$ more time than computing $\Sigma_A(M_j)$ where M_j is some other submodel of M and is not of the form ωT.

In order to identify the impact of each of these parameters on the algorithm's performance, we consider the following test cases:

1. For $S(A)$:

   (a) Fix a large general model M_G. Fix the size of F. Vary the number of temporal operators in F. Expect an exponential decrease in performance.

   (b) Fix a model M. Fix a number of temporal operators. Vary the size of F. Expect a polynomial decrease in performance.

2. For the size of M

   (a) Fix a Formula F. Vary the size of M. Expect polynomial decrease of performance.

3. For $d(F)$, number of ω's in M:

   (a) Fix model M with some omegas, fix the size of F, increase $d(F)$.

   (b) Fix F with some $d(F)$, fix size of M, increase number of ω's in M.

Input data

In order to generate meaningful results, we must consider reasonably large inputs, which in turn means that it becomes impractical to manually create input, and we resort instead to generating it :

1. We generate random formulas, with given length, and given number of temporal operators, so that our result are not specific to any kind of formula structure. Additionally a given atom never occurs more than once in a given formula.

2. Generating random models is harder, due to the way that they have been implemented. Thus in an attempt to maintain generality we consider models of the form $M_G(x)$ for $x \in \mathbb{N}$ where:
   $M_G(0) =< \{\} + \{\}, \omega^*\{\}, \omega\{\} >$
   $M_G(x) =< M_G(x-1) + M_G(x-1), \omega^* M_G(x-1), \omega M_G(x-1) >$
   These are easy to generate, and as they contain all operators nested within each other they are fairly general, and so should not bias our test results significantly.

S(A) Test cases

We first note that with regards to the size of $S(A)$ it is only the number of temporal operators that has a significant impact, not their type, so the only temporal operator appearing in F will be $\mathcal{S}$. There is some variance in the time the algorithm takes for a given input pair, due to the fact that we are generating a random formula F for each call. In order to reduce this, for any given input F, M combination we generated 10 formulas F, and take the average runtime of the algorithm for all 10 runs.

Fixed model, fixed formula length, varying occurrences of S

We consider $M = M_G(0)$ and random F with $|F| = 50$, $|F| = 100$, varying the number of occurrences of $\mathcal{S}$. The test results are displayed in figure4.1 . Note that the y axis is displayed on a logarithmic scale.

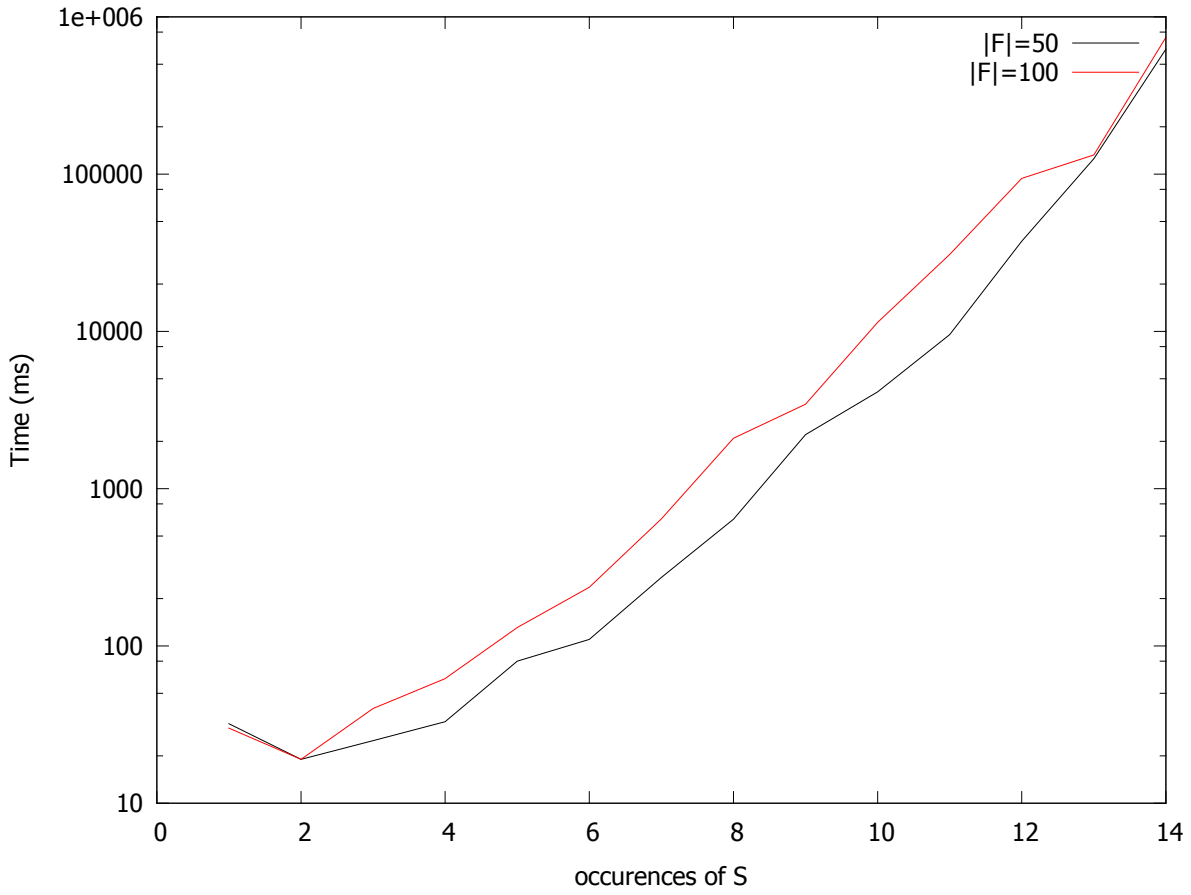

Figure 4.1: varying number of temporal operators

As expected, increasing the number of temporal operators quickly creates an exponential blow-up in the size of $S(A)$, meaning that even for small models ($M = M_G(0)$ has $depth = 2$ and only 4 operators) the algorithm's performance rapidly degrades.

Fixed model, fixed number of temporal operators, varying formula length

We consider first $M = M_G(1)$ and then $M = M_G(2)$, and random F with 3 occurrences of $\mathcal{S}$, increasing $|F|$ from 5 to 305. The test results are displayed in figure4.2 .

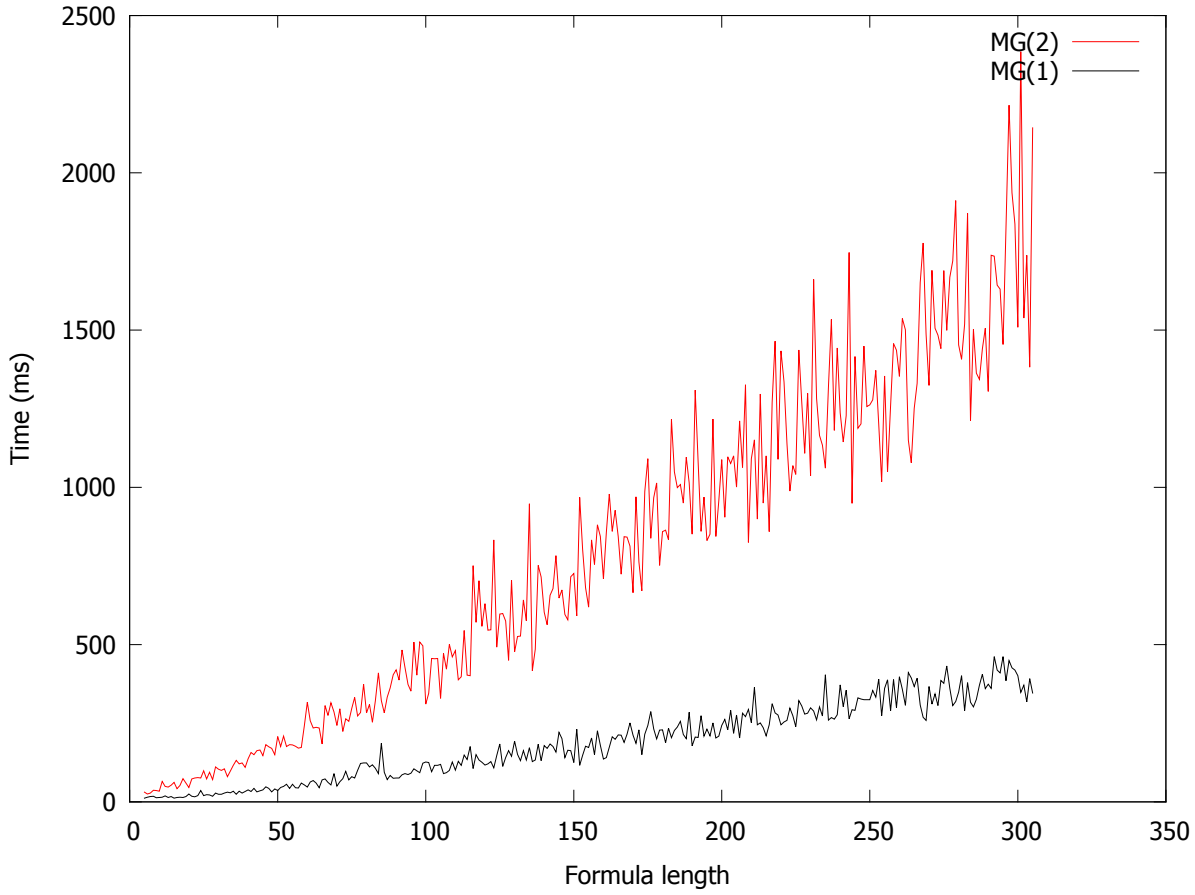

Figure 4.2: varying formula length

We notice that the performance does not degrade significantly as we increase formula size, even for fairly large models ($M_G(2)$ has 84 operator applications).

Size of M Test cases

We have already seen that even for small models, increasing the number of temporal operators rapidly degrades performance. Thus in order to measure the impact that the size of the model alone has on the performance, we consider a formula F with $|F| = 100$ and only 1, then 2, then 3 occurrences of $\mathcal{S}$. For each case we run the algorithm for $M_G(0)$,$M_G(0) + M_G(0)$,$(M_G(0) + M_G(0)) + M_G(0)$,etc. varying the number of $M_G(0)$'s in the sum from 1 to 100. The results are displayed in figure 4.3.

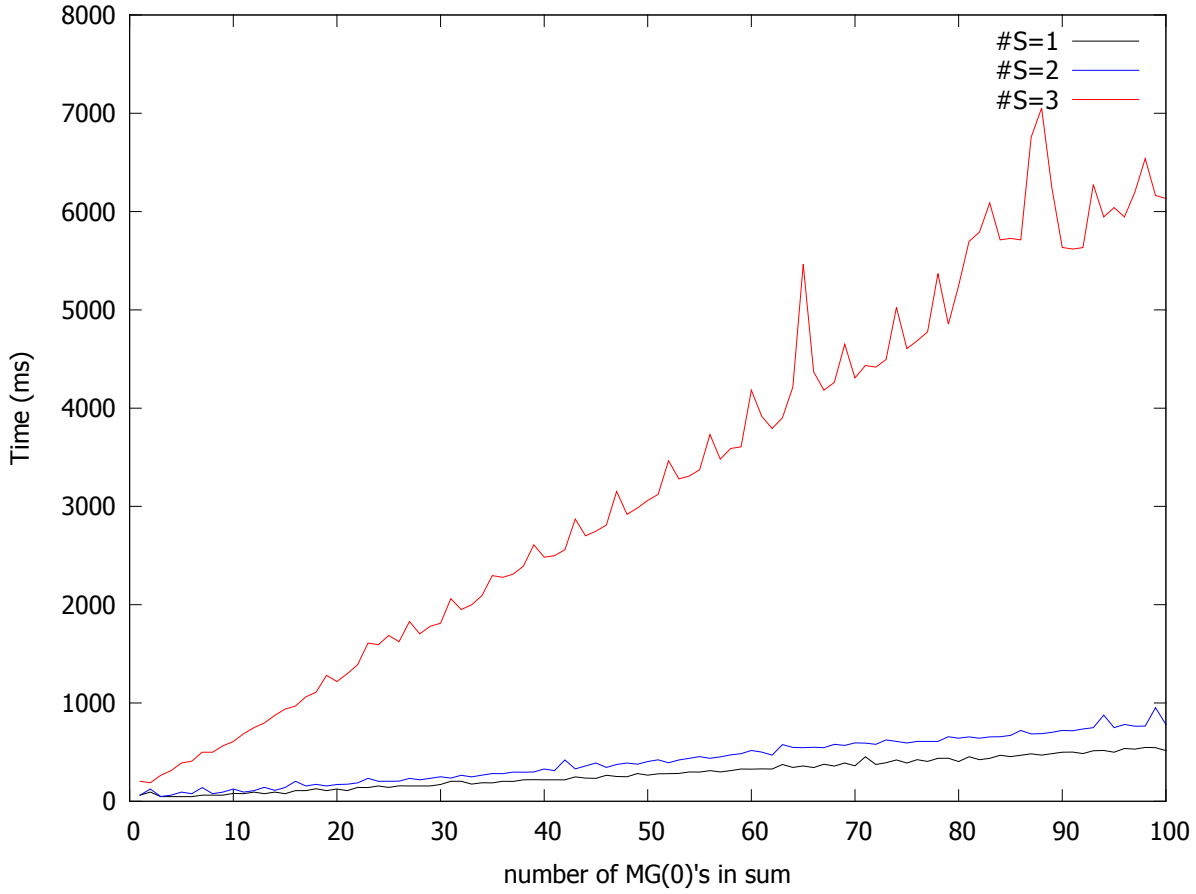

Figure 4.3: varying the model size

We notice that varying the model size seems to only affect performance polynomially, but increasing the number of occurrences of $\mathcal{S}$ even slightly makes the performance curve significantly steeper.

$d(F)$, ω Test cases

We already know from the very first test case that increasing the number of occurrences of $\mathcal{S}$ in the input formula F causes an exponential decrease of performance even for a model with a single ω. This implies that we will have at least an exponential decrease of performance when increasing $d(F)$. Thus what remains to be determined is how the number of ω's in M affect

performance for a given F with fixed $d(F)$. To establish this we consider formulas consisting solely of nested $\mathcal{S}$'s with $d(F) = 5$, $d(F) = 6$ and $d(F) = 7$ respectively. For each of these values we create a model consisting purely of nested ω's, varying the size from 1 to 100. Arguably we are varying both the size of the model and the number of ω's, however as we have already established that increasing the model size generally only increase runtime polynomially for a given formula, it seems reasonable to consider such nested omega models without too much loss of generality (as compared to the ideal case where the size of M is fixed throughout).The test results are illustrated in figure 4.4.

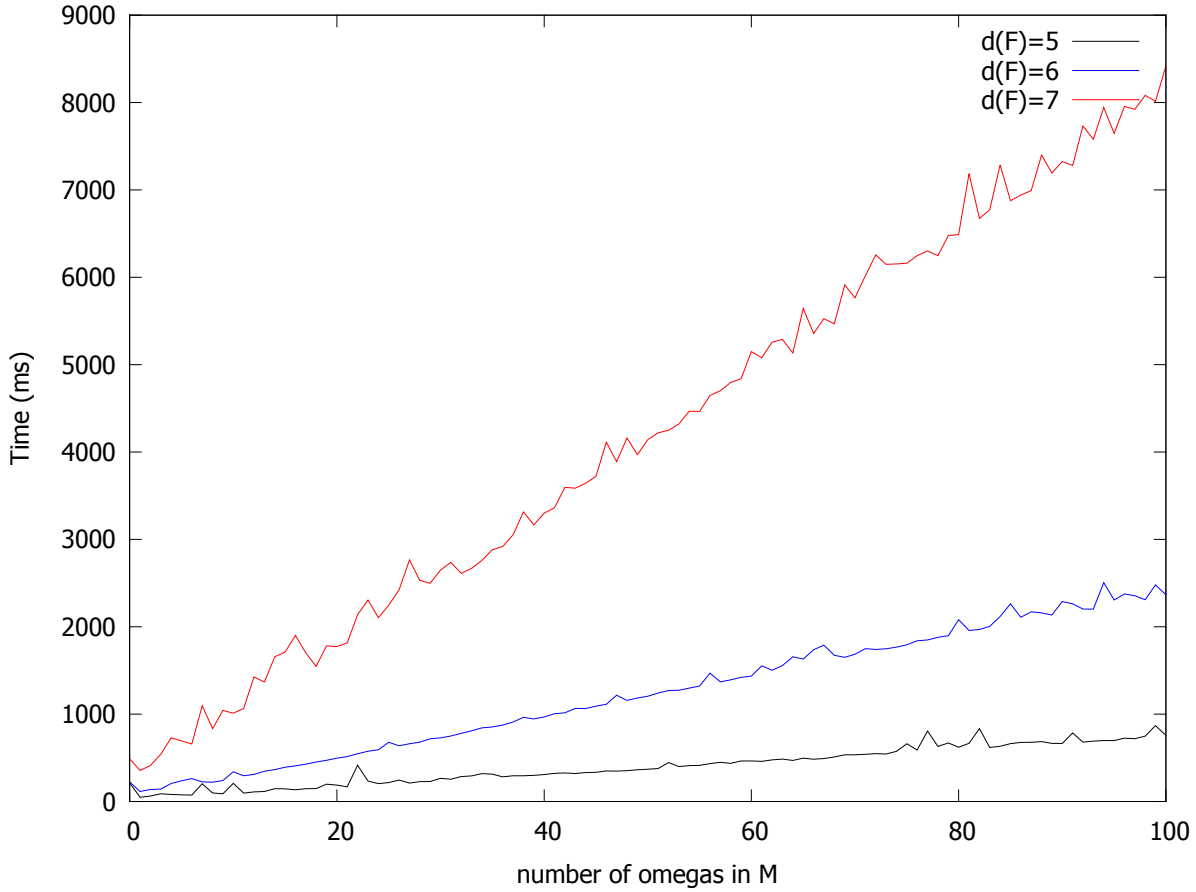

Figure 4.4: varying the number of ω's

Once again, we see that for a given $d(F)$ varying the number of ω's seems to only affect performance polynomially, but increasing $d(F)$ even slightly makes the performance curve significantly steeper.

Summary of test results

Given that we don't have any specific application in mind, it is hard to use the above results to evaluate the algorithm's performance in absolute terms. What the cases do allow us to do however is understand it's potential applicability or lack thereof for certain applications. The cases all seem to generally indicate the algorithm's performance is pretty good if we have a low bounded number of temporal operators appearing in formulas, and degrades exponentially as the number of temporal operators increases. However having a bounded number of temporal operators in formulas of arbitrary length, considering models of arbitrary size, still seems like

a fairly general use case, and so arguably does not limit the algorithm's potential applicability significantly.

Chapter 5

Complexity

In this chapter we will analyse the time and space complexity of our algorithm for determining satisfiability of a formula $A \in L(\mathcal{U}, \mathcal{S})$[1] in a model expression M, a problem which for the rest of this chapter will be referred to as MODELSAT. When calculating the resource usage of an algorithm (i.e. analysing it's space\time complexity), it is often the case that we modify the computation in way which will minimise the usage of whatever resource we are trying to measure the usage of. If for example we are trying to measure space usage, we would consider a version of the algorithm which favours recomputation of intermediate results over storing them, at the expense of time.Similarly if we are trying to measure the time complexity, we want to avoid recomputation but are not concerned with how much space we use, and would thus probably consider a version of the algorithm which caches results to avoid recomputation, at the cost of using additional space.

Thus we will consider two different version of our algorithm, one optimising the time usage and one the space usage, and use these modified version to get a precise idea of the minimal amount of time and space required. Both of these versions of the algorithm are different to the one presented in chapter 3. However all three are arguably variations of the same algorithm as they rely on the same fundamental computational idea, namely that of formula localization.

5.1 Reducing MODELSAT to BALLOONMODELSAT

Recall the algorithm from section 3.2 for determining if a given formula $A \in L(\mathcal{U}, \mathcal{S})$ is satisfied in a model expression T. The idea was to compute the set $\Sigma_A(T)$ of all subformulas\possible localizations of A that are satisfied in T by induction on the structure of the model expression.

Here we consider a different approach. Instead of computing the set $\Sigma_A(T)$ of all subformulas/possible localizations satisfied at a given node T in the model expression tree by computing $\Sigma_A(T_i)$ for each of it's children T_i, we will just compute the localization A_i of A at each of the children T_i of T, and then recursively compute whether A_i is satisfied in T_i for any child T_i. Localization will be computed according to the four localization definitions (one for each of the operators) given in section 3.2. The only difference will be that in each definition of localization all terms of the form $F_i \in \Sigma_A(T_i)$ will be replaced by a recursive call to the overall procedure. Thus using the results of section 3.2 the computation of determining satisfiability of F in a model T can be modelled by a recursive function$\sigma(F, T)$ defined as follows

Definition 5.1. $\sigma(F, T)$:

- $\sigma(F, a) \equiv a, 0 \models F$

[1] We limit our focus to $L(\mathcal{U}, \mathcal{S})$ as it is the more expressive than $L(\mathcal{F}, \mathcal{P})$

- $\sigma(F, T_0 + T_1) \equiv \sigma(F_0, T_0) \vee \sigma(F_1, T_1)$
- $\sigma(F, < T_1, ..., T_n >) \equiv \exists i \in \{0, ..., n\}.\sigma(F_\lambda, T_i)$
- $\sigma(F, \omega T_0) \equiv \exists n \leq d(F).\sigma(F_n, T_0)$[2]

Remark 5.2. Note that the choice of $\sigma(F, T)$ (which was previously used to abbreviate F is satisfied in T) as the name of the function is appropriate, as by definition (and our work in chapter 3) it returns true if and only if F is satisfied in T. Thus $\sigma(F, T)$ will be used interchangeably (depending on the context) to denote the statement "F is satisfied in T" or a call to the function deciding this.

Examining definition 5.1 we notice that for $+$, and $< ... >$ the definition just recurses on the structure of the model expression T, in other words it generates one recursive call to σ for each child T_i of T. However for ωT_0 this is not the case, as here we generate $d(F)$ calls[3] to σ. For the purposes of complexity analysis, such a definition is inconvenient, i.e. we would prefer if we could define σ purely by recursion on the structure of T. Thus our idea is given a model expression T to transform it into an equivalent (for the purposes of localization) expression T' without any occurences of ω. The overall complexity for deciding if A is satisfied in T will then be that of transforming T into T', and determining if A is satisfied in T'.

To achieve this we start by defining *o*model expressions.

Definition 5.3. (*o*model expression): $T = a| < T, ..., T > |T + T| +^o T| +_o T$

The new operators $+^o$, $+_o$ will be used to eliminate ω, ω^* respectively. Note that as the handling of $+_o$ is completely symmetrical to that of $+^o$, in what follows we will consider only the latter. We define localization for the new operator $+^o$:

Definition 5.4. The localization F_λ of a formula F at the child T of $+^o T$ is defined as :

- If F is atomic, $F_\lambda = F$
- $(\neg A)_\lambda = \neg A_\lambda$
- $(A \wedge B)_\lambda = A_\lambda \wedge B_\lambda$
- $(A\mathcal{U}B)_\lambda = \begin{cases} A_\lambda \mathcal{W} B_\lambda & \text{if } \sigma(B_\lambda \wedge \mathcal{H}A_\lambda, M) \\ A_\lambda \mathcal{U} B_\lambda & \text{otherwise} \end{cases}$
- $(A\mathcal{W}B)_\lambda = \begin{cases} A_\lambda \mathcal{W} B_\lambda & \text{if } \sigma(B_\lambda \wedge \mathcal{H}A_\lambda, M) \vee \neg\sigma(\neg A_\lambda, M) \\ A_\lambda \mathcal{U} B_\lambda & \text{otherwise} \end{cases}$
- $(A\mathcal{S}B)_\lambda = A_\lambda \mathcal{S} B_\lambda$
- $(A\mathcal{Z}B)_\lambda = A_\lambda \mathcal{Z} B_\lambda$

Localization for $+_o T$ is symmetrical. We will then say that A is satisfied in the *o*model expression $+^o T$, if and only if A_λ is satisfied in T. So we redefine $\sigma(F, T)$ for *o*model expressions T as follows:

Definition 5.5. $\sigma(F, T)$:

[2] and symmetrically for $\sigma(F, \omega^* T_0)$

[3] recall that $d(F)$(definition 3.30) was defined to be the maximum depth of nesting of $\mathcal{S}/\mathcal{Z}$ in B

- $\sigma(F, a) \equiv a, 0 \models F$
- $\sigma(F, T_0 + T_1) \equiv \sigma(F_0, T_0) \vee \sigma(F_1, T_1)$
- $\sigma(F, < T_1, ..., T_n >) \equiv \exists i \in \{0, ..., n\}.\sigma(F_\lambda, T_i)$
- $\sigma(F, +^o T_0) \equiv \sigma(F_\lambda, T_0)$[4]

It is essential at this point to emphasize that the expression $+^o T$ does not have any real meaning. The sole purpose of *o*model expressions is to allow us to model the way that satisfiability is computed (by means of definition 5.5) in a way which is easier to analyse. They are a purely syntactical construct. How we can transform a given ωmodel expression into an equivalent (for the purposes of satisfiability) *o*model expression (with ω's removed) becomes apparent from the following proposition.

Proposition 5.6. *for any formula $F \in L(\mathcal{U}, \mathcal{S})$,for any model expression M, we have*

$$F \text{ is satisfied in } \omega M \iff F \text{ is satisfied in } (\sum_{i=0}^{d(F)-1} M) + (+^o M)$$

Remark 5.7. We adopt here the convention that for $x, y \in \mathbb{N}$, if $y < x$ then the expression $(\sum_{i=x}^{y} M) + (+^o M)$ is equivalent to the expression $(+^o M)$.

Proof. : It can be shown that there is a sort of associativity for the model expression operator $+$ with regards to localization. So for example given models $M = (M_0 + M_1) + M_2$, $N = M_0 + (M_1 + M_2)$ the localization of a formula F at the sub model M_1 of M which we denote $F^M_{M_1}$ is identical to the localization of F at the submodel M_1 of N, i.e. $F^M_{M_1} = F^N_{M_1}$.

Bearing this associativity in mind, pick arbitrary $k \in \mathbb{N}$, and an arbitrary model expression M, and consider the models ωM and $N = (\sum_{i=0}^{k-1} M) + (+^o M)$. Let $F_i^{\omega M}$ denote the localisation of F at the i-th copy of M in the context of ωM , and F_i^N denote the localisation of F at the i-th copy of M in the the context of N. So for $i < k$, F_i^N is just F localised at the ith element of $(\sum_{i=0}^{k-1} M)$, but F_k^N denotes the localization of F at $+^o M$ in the context of N, further localised at M in the conext of $+^o M$. So in summary $F_i^{\omega M} = F^{\omega M}_{M_i}$, and $F_i^N = \begin{cases} (F^N_{+^o M})^{+^o M}_M & \text{if } i = k \\ F^N_{M_i} & 0 \leq i < k \end{cases}$.

It can then be shown by induction on the structure of a formula $F \in L(\mathcal{U}, \mathcal{S})$ that for any F such that $d(F) \leq k$, $\forall i \leq k.F_i^{\omega M} = F_i^N$.This property is illustrated in figure 5.1.

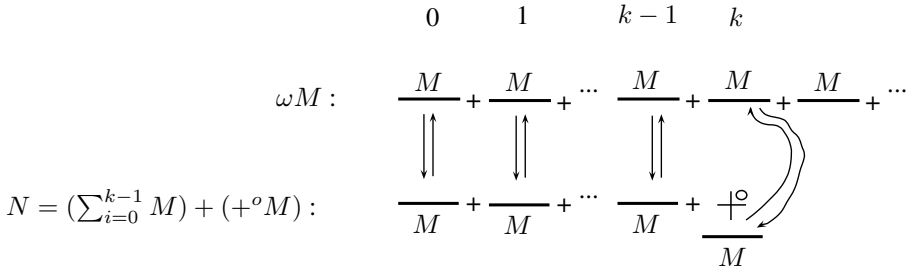

Figure 5.1: $\forall i \leq k.F_i^{\omega M} = F_i^N$

As a result of this we have that for $N = (\sum_{i=0}^{d(F)-1} M) + (+^o M)$, $\forall i \leq d(F).F_i^{\omega M} = F_i^N$. So

[4]and symmetrically for $\sigma(F, +_o T_0)$

1. $\forall i \leq d(F).F_i^{\omega M} = F_i^N$
2. F is satisfied in $\omega M \Longleftrightarrow \exists i \leq d(F).\sigma(F_i^{\omega M}, M)$ (By definition 5.1)
3. F is satisfied in $(\sum_{i=0}^{d(F)-1} M) + (+^o M) \Longleftrightarrow \exists i < d(F).\sigma(F_i^N, M) \vee \sigma(F_{+^o M}^N, +^o M)$
   (by definition 5.1, noted associativity of + with regards to localization)
4. $\exists i < d(F).\sigma(F_i^N, M) \vee \sigma(F_{+^o M}^N, +^o M) \Longleftrightarrow \exists i < d(F).\sigma(F_i^N, M) \vee \sigma((F_{+^o M}^N)_M^{+^o M}, M) \Longleftrightarrow$
   $\exists i \leq d(F).\sigma(F_i^N, M)$
   (by definition 5.5 for $+^o M$, and the def. of F_i^N)
5. F is satisfied in $\omega M \Longleftrightarrow F$ is satisfied in $(\sum_{i=0}^{d(F)-1} M) + (+^o M)$ (by 1,2,3,4)

□

It should be clear that a symmetrical proposition relating ω^* and $+_o$ can be proved. Based on proposition 5.6 we can define a procedure for constructing *o*model expressions from ωmodel expressions.

Definition 5.8. $(o(F,T))$: we define the function $o(F,T)$: $L(\mathcal{U},\mathcal{S}) \times \omega model \rightarrow omodel$ which given an ωmodel expression T and a formula $F \in L(\mathcal{U},\mathcal{S})$ computes an "equivalent" *o*model expression $o(F,T)$:

- T atomic, then $o(F,T) = T$
- $o(F, T_0 + T_1) = o(F,T_0) + o(F,T_1)$
- $o(F, < T_1, ..., T_j >) = < o(F,T_1), ..., o(F,T_j) >$
- $o(F, \omega T_{i+1}) = (\sum_{i=0}^{d(F)-1} o(F,T_{i+1})) + (+^o o(F,T_{i+1}))$
- $o(F, \omega^* T_{i+1}) = (+_o o(F,T_{i+1})) + \sum_{i=0}^{d(F)-1} o(F,T_{i+1})$

The sense in which T and $o(F,T)$ are equivalent is illustrated by the following proposition:

Proposition 5.9. *(omodel expresion equivalence) for any formula $F \in L(\mathcal{U},\mathcal{S})$, for any model expression T, we have*

$$F \text{ is satisfied in } T \Longleftrightarrow F \text{ is satisfied in } o(F,T)$$

Proof. : By induction on the structure of T, using proposition 5.6(and a symmetrical proposition relating ω^* and $+_o$). □

As a result of this we can compute $\sigma(F,T)$ by computing $\sigma(F, o(F,T))$ instead. Finally we consider the complexity of transforming an ωmodel expression T^ω into the equivalent *o*model expression $T = o(T^\omega)$

Proposition 5.10. *For any ωmodel expression T, and formula F there is a Turing Machine TM_o which computes $o(F,T)$ in time $O(d(F)^k * m)$, and polynomial space.*

Proof. :Let T be of size m (i.e. the model expression tree has m nodes), and have k occurrences of ω. For the sake of simplicity, we consider ωmodel expressions T with no occurrences of ω^*. Starting at the root of T, at each level of recursion of $o(F,T)$ (which corresponds to a level in the input model expression tree T) it transforms all subtrees $T_j = \omega T_{j+1}$ at that level into $T'_j = (\sum_{i=0}^{d(F)-1} T_{j+1}) + (+^o T_{j+1})$, so $size(T'_j) = O(d(F) * size(T_j))$, and subsequently

$size(T') = O(d(F) * size(T))$ (the worst case occurs when $T = \omega T_j$ for some T_j). The number of these tree modifying levels of recursion is bounded by k (it will be equal to k when all ω are nested within each other). Thus for the final tree $o(F,T)$ we have $size(o(F,T)) = O(d(F)^k * size(T)) = O(d(F)^m * m)$, so the size of the new model is exponential in the size of the original model. Consider a Turing machine TM_o implementing $o(F,T)$. As all that TM_o is doing is copying over nodes of T from it's input tape to it's output tape, and the overall number of nodes it needs to copy over is $O(d(F)^k * m)$, we can argue that this is also a reasonable upper bound for it's time complexity. So $time(o(F,T)) = O(d(F)^k * m)$.

Additionally TM_o operates in polynomial space: recall that it is just copying over parts of the tree T. This can be done by traversing T, and at each level creating a map of nodes at that level to number of times they must be written to output tape (with ω replaced by $d(F)$ applications of $+$, and $+^o$ at the end). This map can then be use to build the corresponding levels of the resulting $o$model expression $o(F,T)$ in space that is polynomial in the size of the map. Clearly the map for the last level (leaves) will be the largest , as it will have the most nodes, and the largest number of copies for each node (compared to previous levels). Thus the space usage of TM_o is polynomial in the size of this final map. The map has polynomially many elements in the size of the model, and each element maps to a number which is exponential in the size of T. Thus such a number will be representable by a bit string of size polynomial to the size of the original model. Hence the map takes space polynomial in the input size, and subsequently TM_o is in PSPACE.

Note that at the start of the proof we assumed for simplicity that there were no occurrences of ω^*in M. It should be clear that a completely symmetrical argument to the one above can be applied to extend the proof to cover the case where ω^* does occur in M .

□

Summary

Given a formula F and a model expression T^ω, we have reduced the problem of computing $\sigma(F,T^\omega)$ to that of computing the expression $T = o(F,T^\omega)$, and then computing $\sigma(F,T)$. We have considered the complexity of computing $T = o(F,T^\omega)$. Subsequently we can now proceed to analyse the complexity of computing $\sigma(F,T)$, in order to analyse the overall complexity of computing $\sigma(F,T^\omega)$.

Remark 5.11. In the sections that follow, the problem of determining whether F is satisfied in an $o$model expression T will be referred to as BALLOONMODELSAT, to contrast it with our original problem of determining whether F is satisfied in an ωmodel expression T^ω, which we refer to as MODELSAT.

5.2 Deciding BALLOONMODELSAT in P

It can be shown that a direct recursive implementation of definition 5.5 in fact takes time exponential in the size of the input (the proof is beyond the scope of this section). The cause of this is that as we are just recursively computing $\sigma(F,T)$ from the root of the $o$model expression to it's leaves, we will have to recompute various intermediate results exponentially many times. In this section we will present a modified version of the naive algorithm implied by definition 5.5, which by caching intermediate results can compute $\sigma(F,T)$ in polynomial time.

5.2.1 Overview of the algorithm

We start by giving an overview of the algorithm, where C is the input formula, and T is the *o*model in which we want to determine whether C is satisfied.

1. loop over subformulas of C in order of increasing size (from atomic subformulas up to C itself) and for each such subformula F:

   - If F is a Boolean (atom, conjunction, negation) iterate through every node T_i of model tree T trivially computing and storing the localization F_i (the localizations of F's subformulas at that node will have been inductively computed).
   - If $F = A\mathcal{S}B$:
     (a) Starting from the leaves of T and working to the root, compute $\sigma(B_i \wedge \mathcal{G}A_i, T_i)$,$\sigma(\neg A_i, T_i)$ at each node T_i in terms of $\sigma(B_j \wedge \mathcal{G}A_j, T_j)$,$\sigma(\neg A_j, T_j)$ where T_j i s any child of T_i, storing the results as 2 bits.
     (b) Then starting from the root of T and working towards the leaves, compute the localization $(A\mathcal{S}B)_i$ at each node T_i of T and store it. By (a) we have computed and stored the localization context for each T_i, and we have also inductively computed and stored the localizations A_i and B_i at T_i. Thus we compute $(A\mathcal{S}B)_i$ by a simple look-up.
   - Similarly for $F = A\mathcal{Z}B$.
   - If $F = A\mathcal{U}B$ we do the same as for $F = A\mathcal{S}B$ but this time computing and storing $\sigma(B_i \wedge \mathcal{H}A_i, T_i)$ instead of $\sigma(B_i \wedge \mathcal{G}A_i, T_i)$ for each node.
   - Similarly for $F = A\mathcal{W}B$.

2. On the final iteration of the loop above, we consider C itself. At the end of this iteration, we will have computed the localization of C at each atomic world, and thus to determine if C is satisfied in T just need to check whether $\sigma(C_i, T_i)$ for any atomic world T_i.

Claim 5.12. This algorithm decides BALLOONMODELSAT in p-time.

Proof. : We give in detail the steps of the algorithm outlined above, and simultaneously analyse the time taken by each step.

1. One iteration for each subformula of F so number of iterations is linear in the size of the formula. For each iteration:

   - If F is boolean, then loop over nodes T_i of T, computing and storing F_i at each one:
     – F is atomic: trivially $F = F_i$
     – $F = A \wedge B$: we have inductively computed and stored A_i and B_i, so just store $F_i = A_i \wedge B_i$
     – $F = \neg A$: similarly we have inductively computed and stored A_i, so just store $F_i = \neg A_i$

     The number of nodes T_i is linear in the input size, so this case takes polynomial time.
   - If $F = A\mathcal{S}B$:
     (a) loop over the nodes T_i of T starting from the leaves (single point models) and working up the tree to the root of T :
       i. If T_i atomic: can compute $\sigma(B_i \wedge \mathcal{G}A_i, T_i)$,$\sigma(\neg A_i, T_i)$ in p-time (A_i, B_i have been computed and stored by previous iterations).

ii. $T_i = T_j + T_k$: $\sigma(B_i \wedge \mathcal{G}A_i, T_i) = \sigma((B_i \wedge \mathcal{G}A_i)_j, T_j) \vee \sigma((B_i \wedge \mathcal{G}A_i)_k, T_k)$. $(B_i \wedge \mathcal{G}A_i)_k = B_{ik} \wedge \mathcal{G}A_{ik}$ by def. of localization, and we have already computed and stored $\sigma(B_{ik} \wedge \mathcal{G}A_{ik}, T_k)$ as the computation is moving up the tree. On the other hand again by the definition of localization $(B_i \wedge \mathcal{G}A_i)_j = B_{ij} \wedge (\mathcal{G}A_i)_j$, where $(\mathcal{G}A_i)_j = (A_i \mathcal{W} \bot)_j = \begin{cases} \bot & if\ \sigma(\neg A_{ik}, T_k) \\ \mathcal{G}A_{ij} & otherwise \end{cases}$.
As $\sigma(\neg A_{ik}, T_k)$, $\sigma(B_{ij} \wedge \mathcal{G}A_{ij}, T_j)$ have already been computed and stored (and clearly $\sigma(B_{ij} \wedge \bot, T_j) = \bot$), we can compute $\sigma(B_i \wedge \mathcal{G}A_i, T_i)$ in *constant time*, by just looking up our stored results.
Similarly $\sigma(\neg A_i, T_i) = \sigma(\neg A_{ik}, T_k) \vee \sigma(\neg A_{ij}, T_j)$ which can again be determined in constant time as we have already computed and stored $\sigma(\neg A_{ik}, T_k)$ and $\sigma(\neg A_{ij}, T_j)$.

iii. $T_i = +^o T_j$: very similar to ii.$\sigma(B_i \wedge \mathcal{G}A_i, T_i) = \sigma((B_i \wedge \mathcal{G}A_i)_j, T_j)$. $(B_i \wedge \mathcal{G}A_i)_j = B_{ij} \wedge (\mathcal{G}A_i)_j$, where $(\mathcal{G}A_i)_j = \begin{cases} \bot & if\ \sigma(\neg A_{ij}, T_j) \\ \mathcal{G}A_{ij} & otherwise \end{cases}$. As above all necessary terms have already been computed as we are moving up the tree, so here too we compute $\sigma(B_i \wedge \mathcal{G}A_i, T_i)$ in constant time. Similarly $\sigma(\neg A_i, T_i) = \sigma(\neg A_{ij}, T_j)$ which has already been computed.

iv. $T_i = +_o T_j$: similarly $\sigma(B_i \wedge \mathcal{G}A_i, T_i) = \sigma((B_i \wedge \mathcal{G}A_i)_j, T_j) = \sigma(B_{ij} \wedge \mathcal{G}A_{ij}, T_j)$, $\sigma(\neg A_i, T_i) = \sigma(\neg A_{ij}, T_j)$.

v. $T_i =< T_1 ... T_k >$: as by the definition of localization for shuffle the localization context is the same for all shuffled models, we have that $\sigma(B_i \wedge \mathcal{G}A_i, T_i) = \bigvee_{j=1}^{k} \sigma((B_i \wedge \mathcal{G}A_i)_\lambda, T_j)$. $(B_i \wedge \mathcal{G}A_i)_\lambda = B_{i\lambda} \wedge (\mathcal{G}A_i)_\lambda$, where $(\mathcal{G}A_i)_\lambda = \begin{cases} \bot & if\ \bigvee_{j=1}^{k} \sigma(\neg A_{i\lambda}, T_j) \\ \mathcal{G}A_{i\lambda} & otherwise \end{cases}$. Each of the $\sigma(\neg A_{i\lambda}, T_j)$ terms has been inductively computed so we can compute the disjunction in p-time. Similarly all the $\sigma(B_{i\lambda} \wedge \mathcal{G}A_{i\lambda}, T_j)$ terms have been inductively computed, so that $\sigma(B_i \wedge \mathcal{G}A_i, T_i)$ can be computed in p-time.

Thus in conclusion for any node T_i the algorithm computes $\sigma(B_i \wedge \mathcal{G}A_i, T_i)$,$\sigma(\neg A_i, T_i)$ in p-time, and as there are polynomially many nodes, computing it for all of them will be p-time.

(b) Starting from the root of of T, we recursively compute and store F_i at each of it's successors T_i, until we get to the atomic models. Thus at any given iteration we are considering the localization F_i of F at some node T_i, and trying to determine F_{ij} for each child T_j of T_i. Clearly $F_i = A_i \mathcal{S} B_i$ or $F_i = A_i \mathcal{Z} B_i$. Assume $F_i = A_i \mathcal{Z} B_i$ (the $F_i = A_i \mathcal{S} B_i$ case is computationally slightly simpler as we have a simpler good context condition). Then:

i. If T_i atomic: do nothing, as it has no children.

ii. $T_i = T_j + T_k$: By definition of localization $(A_i \mathcal{Z} B_i)_j = A_{ij} \mathcal{Z} B_{ij}$, where A_{ij}, B_{ij}have already been inductively computed and stored.
Similarly$(A_i \mathcal{Z} B_i)_k = \begin{cases} A_{ik} \mathcal{Z} B_{ik} & if\ \sigma(B_{ij} \wedge \mathcal{G}A_{ij}, T_j) \vee \neg\sigma(\neg A_{ij}, T_j) \\ A_{ik} \mathcal{S} B_{ik} & otherwise \end{cases}$ where A_{ik} , B_{ik} have been inductively computed and stored, as has the good context condition $\sigma(B_{ij} \wedge \mathcal{G}A_{ij}, T_j) \vee \neg\sigma(\neg A_{ij}, T_j)$ (in part (a)). Thus the algorithm computes F_{ij}, F_{ik} in constant time by looking up the stored results.

iii. $T_i = +^o T_j$: By definition of localization $(A_i \mathcal{Z} B_i)_j = A_{ij} \mathcal{Z} B_{ij}$.

iv. $T_i = +_o T_j{:}(A_i \mathcal{Z} B_i)_j = \begin{cases} A_{ij}\mathcal{Z}B_{ij} & if\ \sigma(B_{ij} \wedge \mathcal{G}A_{ij}, T_j) \vee \neg\sigma(\neg A_{ij}, T_j) \\ A_{ij}\mathcal{S}B_{ij} & otherwise \end{cases}$ where A_{ij} , B_{ij} have been inductively computed, and the good context condition has been computed in (a). Thus the algorithm computes F_{ij} in constant time by looking up the stored results.

v. $T_i =< T_1 ... T_k >:\ (A_i \mathcal{Z} B_i)_\lambda = \begin{cases} A_{i\lambda}\mathcal{Z}B_{i\lambda} & if\ \neg(\bigvee_{j=1}^{k} \sigma(\neg A_{i\lambda}, T_j)) \\ A_{i\lambda}\mathcal{S}B_{i\lambda} & otherwise \end{cases}$ where $A_{i\lambda}$, $B_{i\lambda}$have been inductively computed, as have all the $\sigma(\neg A_{i\lambda}, T_j)$ terms(in (a)), so F_{ij} is computed in p-time.

So all localisations computed in p-time, polynomially many nodes to localise at, so (b) takes p-time overall.

- The case for $F = A\mathcal{U}B$ is symmetrical to that for $F = A\mathcal{S}B$ and thus no more computationally complex..
- The case for$F = A\mathcal{Z}B$ is almost identical to that of $F = A\mathcal{S}B$ albeit with a slightly more general context, but clearly no more computationally complex.
- The case for $F = A\mathcal{W}B$ is symmetrical to that for $F = A\mathcal{Z}B$ and thus no more computationally complex.

In conclusion (a) and (b) both take polynomial time, and the number of iterations is linear in the length of their formula, so the whole of part 2 is p-time.

2. Determining if any of the localizations of C at the atomic worlds are satisfied is clearly p-time (polynomially many such worlds, p-time to check satisfiability at each one).

Thus the algorithm consists of 2 polynomial time parts, and as such takes p-time overall. It should also be clear from the above that it decides BALOONMODELSAT (as all computation is according to definition 5.5, and the respective definitions of localisation).

□

From the above we know that there exists a machine TM_{BMSAT} deciding BALLONMODELSAT in polynomial time. As a result of section 5.1 we know that by composing the machine TM_o within TM_{BMSAT} we get a machine $TM_o \cdot TM_{BMSAT}$which decides MODELSAT. This gives us an interesting result. Consider the problem MODELSAT-K which is the same as MODELSAT, but where the number of ω's in the input model is bounded by an integer K.

Proposition 5.13. *(MODELSAT-K): MODELSAT-K is in P (for any $K \in \mathbb{N}$)*

Proof. :Recall that by proposition 5.10 $time(TM_o) = O(d(F)^k * m)$. as $k \leq K$, we have $time(TM_o) = O(d(F)^K * m)$ where K is constant, implying that TM_o operates in p-time, and as TM_{BMSAT} is p-time $TM_o \cdot TM_{BMSAT}$ will also be p-time (as P is closed under composition), and by their definition decides MODELSAT-K. □

5.3 Deciding MODELSAT in PSPACE

In this section we will show that MODELSAT can be decided in PSPACE. We start be considering the machine $TM_o \cdot TM_{BMSAT}$ described in the previous section. Unfortunately it does not operate in polynomial space.Recall that given an ωmodel T^ω of size m, TM_o outputs an $o$model T of size m' where $m' = exp(m)$, $d(T) = poly(d(T^\omega))$. As the algorithm described in section 5.2 is storing localizations at every node of T, and the size of T is exponential in the size of the model T^ω, the algorithm's space usage will be exponential in the size of the input model T^ω. Subsequently we will modify the algorithm to use space linear in the depth of T, as this is only polynomially larger than that of T^ω. The result will be an algorithm which given T and a formula F of size f, decides BALLOONMODELSAT in $poly(m, f) - SPACE$.

We start by giving a high level overview of the algorithm. We then give give a formal description of the Turing Machine implementing this algorithm, and analyse it's space usage.

5.3.1 Overview of the algorithm deciding BALLOONMODELSAT in $poly(m, f)-SPACE$

The algorithm for deciding $\sigma(A, T)$ is illustrated in figure 5.1. We limit our focus to models T with no operators occurring apart from $+$. This is because the other operators are handled in a similar way, and will be covered in detail in the following section where we give a formal description of the machine implementing the algorithm.

5.3.2 The machine M deciding BALLOONMODELSAT in $poly(m, f)-SPACE$

We now proceed to give a formal description of the Turing Machine implementing the algorithm.

5.3.2.1 Input Encoding

Model Encoding: Model T is encoded as tree of nodes, with a node corresponding to each operation application, and single point model. Each node has a unique identifier, and is encoded as a triple $(nid, type, children)$ where:

- nid is the identifier of that node.
- $type$ is a, $+$, $+^o$, $+_o$ or$<>$.
- $children$ is a list of the nodes children (so for $+$ an ordered list of two elements, for $+o, o+$ a list with a single element, for $<>$ an unordered list of n elements, for a a list of propositional atoms).

A model tree is then encoded as a list of nodes. E.g. the expression $+^o(\{p, q\} + \{r\})$ could be encoded as $\{(0, +^o, [1]), (1, +, [2, 3]), (2, a, [p, q]), (3, a, [r])\}$.

Formula Encoding: Formula F is written in polish notation.

5.3.2.2 Machine Description

In this section we describe the aforementioned machine.The machine we describe can be conceptually split into two parts. The first is the $M\sigma$ part, which decides satifiability, by making calls to the localization part $M\lambda$. $M\lambda$ in turn computes localizations of formulas by making calls to the $M\sigma$ part in order to determine localization contexts. Each part has it's own work tapes, but there is one global input tape, and one global output tape. The input tape contains

Algorithm 5.1 the algorithm deciding $\sigma(F,T)$

```
function σ(F,T):
    if T is atomic:
        evaluate F at single world of T, returning result
    else if T = T_0 + T_1:
        Create two copies F0, F1 of F to edit in loop below
        loop over subformulas of F, from atomic subformulas up to F itself:
            if current subformula is B𝒰C:
                extract subformulas B1, C1 of F1 corresponding to B, C
                if σ(C1 ∧ ℋB1,T_1)://recursive call
                    replace the character 𝒰 in F0 corresponding
                    to the 𝒰 in B𝒰C by 𝒲
            else if current subformula is B𝒲C:
                extract subformulas B1, C1 of F1 corresponding to B, C
                if (not σ(C1 ∧ ℋB1,T_1)) and σ(¬B1,T_1)://recursive calls
                    replace the character 𝒲 in F0 corresponding
                    to the 𝒲 in B𝒲C by 𝒰
            else if current subformula is B𝒮C:
                    mirror image of 𝒰 case,
                    swapping 𝒰 with 𝒮 , 𝒲 with 𝒵 ,
                    0 with 1 and ℋ with 𝒢
            else if current subformula is B𝒵C:
                    mirror image of 𝒲 case,
                    swapping 𝒰 with 𝒮 , 𝒲 with 𝒵 ,
                    0 with 1 and ℋ with 𝒢
            end if
        end loop
        if σ(F0,T_0)://recursive call
            return true
        if σ(F1,T_1)://recursive call
            return true
        return false
end σ(F,T)
```

model T and formula F. On the tapes, we represent models by the *nid* of their root, which takes space $O(log(m'))$. If we need any other information about a particular node, such as it's *children* we can use an $O(log(m'))$ counter to look it up in the input tape using the node's *nid*. Thus for the remainder of this section, whenever we refer to storing a model on a tape, all that we are actually storing is the *nid* of the root of that model. Finally, it is worth noting that as this is a recursive computation, we treat all tapes as stacks, and assume the existence of some stack element delimiter character. For a given stack tape S, we will assume the existence of a stack pointer tape, and will use Ssp to denote the top of the stack tape S.

Below we describe the two parts of the machine.

The M_σ part

The M_σ part of the machine models a recursive function $\sigma(F, T)$ which determines if F is satisfied in model T. It does this by localising F at each child T_i of T in turn(by calling a localization procedure $\lambda(F, T, T_i)$), and recursively computing $\sigma(F_i, T_i)$.This behaviour is formalised in figure 5.2.

Algorithm 5.2 $\sigma(F, T)$

```
σ(F,T){
    if (T= atomic) {
        //σ1(F,T) evalutes F
        // at single point model T
        return σ1(F,T);
    }
    else{
        for (U child(T)) {
            FL = λ(F,T,U);
            If( σ(FL,U)) {
                return true;
            }
        }
        return false;
    }
}
```

We can now describe the part of the machine implementing this behaviour. It has 3 tapes:

- T_σ: stack of models for which we want to determine satisfiability.
- U_σ: stack of submodels. Submodels of different models separated by '*' character. The submodels at the top of the U_σ stack are the children of the model at the top of the T_σ stack.
- F_σ: stack of localised formulas.

The overall machine M starts by pushing the input model T to the tape T_σ (updating $T_\sigma sp$), and pushing the input formula F onto the tape F_σ (this is only done once), after which it invokes M_σ.

M_σ starts by checking if $T_\sigma sp$ is atomic, in which case it just calls the machine $M_{\sigma 1}$ for determining satisfiability in an atomic model, returning it's result.

If on the other hand $T_\sigma sp$ is not atomic, M_σ pushes all the children of $T_\sigma sp$ onto the stack U_σ. For each child T_i on U_σ in turn, it first computes the localization F_i by calling $M_\lambda(F, T, T_i)$, then pushes the result F_i onto F_σ, and pops T_i off U_σ, pushing it onto T_σ, and then recursing. Subsequently if any recursive call returns true (meaning that the localization F_i of F at some atomic model T_i is satisfied in that model) M_σ returns true, and if none of them do then M_σ returns false. This is formalised by the pseudocode in figure 5.3.

Preliminary remarks concerning figure 5.3:

- In the code that follows, it is implicitly assumed that a "return" statement only writes to the output tape if its is nested within the original call to $M_\sigma(F, T)$. If on the other hand it is called as a result of a recursive call to M_σ by M_λ, then the machine M just notifies the caller of the result.
- It is implicitly assumed that the machine pushes and pops symbols indicating recursive calls as necessary. We omit such symbols from the pseudocode in order to not further obscure it's function.
- We assume the existence of a machine $M_{\sigma 1}$ which determines if a formula is satisfied in a single point model using the algorithm given as part of the proof of lemma 3.25. This can be done by using space linear in the size of the formula for evaluation, and a counter which takes logspace in the size of the model. Thus it makes no significant contribution in terms of space usage.
- We often refer in the code to " the last call to M_σ by M_λ". Clearly there will be only one top-level call to M_σ which is not made by M_λ, and that is the original call by the machine M itself.

Algorithm 5.3 The M_σ part of the machine

```
1  //these lines are executed once to get input
2  //onto work tapes
3  push input.M to T_σ
4  push input.F to F_σ
5
6  M_σ:
7  //Ssp indicates the top element of stack S.
8  if (Atomic(T_σsp)) {
9      pop F_σ, T_σ to M_σ1 input tape
10     res= run M_σ1
11     If (res == True) {
12         pop all data put onto F_σ,T_σ,U_σ by the last recursive call to M_σ by M_λ
13         return True
14     }
15     else if (isEmpty(T_σ) ) {
16         // isEmpty(T_σ) determines if there are no more models to check on
17         // T_σ for the last call to M_σ by M_λ,
18         // indicating that we have recursively checked all models for that call.
19         pop all data put onto F_σ,T_σ,U_σ by the last recursive call to M_σ by M_λ
20         return False
21     }
22     else {
23         while (U_σsp='*') {
24             // This is the case where we have checked all submodels
25             // for T_σsp
26             pop T_σ //updates T_σsp
27             pop F_σ
28             pop U_σ // U_σsp should now be pointing to last submodel of T_σsp
29         }
30         jumpto Localize
31     }
32 }
33 else{//T_σsp.type = +|| < ... > || +_o ||+^o
34     push '*' onto U_σ //indicates new set of submodels
35     for ( model child: T_σsp.getChildren()) {
36         push child onto U_σ
37     }
38 Localise:
39     if (isEmpty(T_σ) ) {
40         pop all data put onto F_σ,T_σ,U_σ by the last recursive call to M_σ by M_λ
41         return False
42     }
43     //call M_λ
44     push T_σsp to T_λ, U_σsp to U_λ, |F_σsp| to FC, F_σsp to F0 and F1
45     Call M_λ
46     pop T_λ, U_λ, FC, F0, F1//clean up
47     //recursive call to M_σ
48     push M_λ.result onto F
49     // M_λ.result is F0sp or F1sp depending on what we're localising
50     pop U_σ, push result onto T_σ
51     jumpto M_σ
52 }
```

The M_λ Part

The aforementioned localisation procedure $\lambda(F, T, T_i)$ is implemented by the M_λ part of the machine, which computes the localization of the formula F at the child model T_i of model T. It has 5 tapes:

- T_λ: stack of models.
- U_λ: stack of models, each of which is a child of a model on T_λ, at which we are trying to localise F.
- FC_λ: stack of pointers. Each pointer indicates the current position of localization in $F0$, $F1$.
- $F0$: it's top element is the localisation at world T_0 for the model $T_\lambda sp$ if $T_\lambda sp = T_0 + T_1$, or the single localization for $T_\lambda sp$ if it's of the form $+^o T_0, +_o T_0, < T_0, ..., T_1 >$.
- $F1$: it's top element is the localisation at world T_1 for the model $T_\lambda sp$ if $T_\lambda sp = T_0 + T_1$.

The top element of each tape/stack represent the current (recursive) computation. So at any given time, we know that M_λ is localising a formula F at the submodel $U_\lambda sp$ of the model $T_\lambda sp$, with $F0sp$, $F1sp$ corresponding to the partially computed localizations of F, and FC_{sp} indicating the progress of the localization of F.

Initially the formula F is pushed onto $F0$, $F1$, and $FCsp$ points to the last symbol of $F0sp$, $F1sp$. On each iteration $M_\lambda$examines the symbol of $F0sp$, $F1sp$ pointed to by $FCsp$. If it is a boolean symbol, it leaves it unmodified. If on the other hand it is a temporal symbol ($\mathcal{U}$,$\mathcal{W}$,$\mathcal{S}$ or $\mathcal{Z}$) it calls the M_σ part (by jumping to M_σ) in order to determine the localization context (where the context is determined based on our definitions of localization for each formula/model). After each symbol is processed the counter $FCsp$ is decremented. When the counter reaches 0, $F0$, $F1$ will contain the localizations of the original formula F.

The whole procedure is formalised below by means of four cases, depending on the type of the model T in the context of which we are localising.

$T = T_0 + T_1$**:**
Initially $FCsp = |F|$. In a loop, process $F0sp[FCsp]$ (the symbol of $F0sp$ at index $FCsp$), $F1sp[FCsp]$ according to the following rules:

- if $F0sp[FCsp] = \mathcal{U}$: Find the sub formulas of $F0sp$ which are the arguments of this $\mathcal{U}$, call the left one $A1$, and the right one $B1$[5], and push $B1 \wedge \mathcal{H}A1$ onto F, T_1 onto T . Run M . if it returns true then set $F0sp[FCsp] = \mathcal{W}$, else leave it unmodified.
- if $F0sp[FCsp] = \mathcal{W}$: As above, but else case becomes: push $\neg A1$ onto F, T_1 onto T. Run M . If it returns true set $F0sp[FCsp] = \mathcal{U}$. Note that the second call $M(\neg A1, T_1)$ reuses the space of the first, so space complexity will be the same as for $\mathcal{U}$.
- if $F0sp[FCsp] = \mathcal{S}$: temporally symmetrical to $\mathcal{U}$ case ($F0sp[FCsp]$ is not modified but $F1sp[FCsp]$ may be), thus no more complex in terms of space.

[5]finding these formulas can be done with a single counter taking logspace in the size of $F0$, $F1$

- if $F0sp[FCsp] = \mathcal{Z}$: temporally symmetrical to $\mathcal{W}$ case ($F0sp[FCsp]$ is not modified but $F1sp[FCsp]$ may be), thus no more complex in terms of space.

After processing a symbol, we decrement $FCsp$. When $FCsp = 0$ we exit the loop, and we will have computed both localizations of F , $F_0 = F0sp$, $F_1 = F1sp$.

The remaining cases $T = +^o T_0$, $T = +_o T_0$, $T =< ... >$ are extremely similar, with the notable difference that there is only a single localization to compute, and we use the $F0$ tape to compute it. As in the $T = T_0 + T_1$ case we process each symbol of $F0sp$, $F1sp$ in a loop, and all that changes is the handling of $\mathcal{U}, \mathcal{W}, \mathcal{S}, \mathcal{Z}$.

$T = +^o T_0$**:**

- if $F0sp[FCsp] = \mathcal{U}$: Identical to the case for $T_0 + T_1$, with the exception that we call $M\,(B0 \wedge \mathcal{H}A0, T_0)$ instead of $M\,(B1 \wedge \mathcal{H}A1, T_1)$.
- if $F0sp[FCsp] = \mathcal{W}$: Identical to the case for $T_0 + T_1$, with the exception that we call $M\,(\neg A0, T_0)$ instead of $M\,(\neg A1, T_1)$.

This case clearly has the same space complexity as $+$.

$T = +_o T_0$**:** This case is temporally symmetrical to the $+^o$ case (swap $\mathcal{S}$ case with $\mathcal{U}$ case, $\mathcal{Z}$ case with $\mathcal{W}$ case, and replace $\mathcal{H}$ with $\mathcal{G}$), so space complexity is no higher than for $+^o$.

$T =< ... >$**:**

- if $F0sp[FCsp] = \mathcal{U}$: As in the previous cases extract arguments $A0$, $B0$ of $\mathcal{U}$. For each child T_i of T one after the other, push $B0$ onto F, T_i onto T, and run M. Do this successively, reusing space for each call. If any call returns true then successively for each child T_i of T, push $\neg A0$ onto F, T_i onto T and run M (reusing space between calls). If none of them returns true then set $F0sp[FCsp] = \mathcal{W}$.
- if $F0sp[FCsp] = \mathcal{W}$: Similarly to the above, call $M_\sigma(\neg A0, T_i)$ for each child T_i, and if any call returns true set $F0sp[FCsp] = \mathcal{U}$.
- if $F0sp[FCsp] = \mathcal{S}$: temporally symmetrical to $\mathcal{U}$ case, thus no more complex in terms of space.
- if $F0sp[FCsp] = \mathcal{Z}$: temporally symmetrical to $\mathcal{W}$ case, thus no more complex in terms of space.

Due to reuse of space this case is also no more complex in terms of space than that of $+$.

5.3.2.3 Space Usage

Let $s(X)$ denote the space used by an element X on the tapes of the machine, where X is a call to M_σ, M_λ, a formula F, or an *nid* T. In order to analyse the space complexity of the machine described, we rely on the fact that space can be reused between recursive calls (to M_σ or M_λ) that are not nested within each other. So all calls made at a given depth of recursion can reuse space.

$$s(M_\sigma(F,T)) =$$
$$s(T.children) + max(s(T) + s(T_i) + 2 * s(F) + s(M_\lambda(F,T,T_i)),\ s(F_i) + s(T_i) + s(M_\sigma(F_i,T_i)))$$

where

- $s(T.children)$: space used to store $T.children$, figure 5.3 line 36.
- $s(T) + s(T_i) + 2 * s(F) + s(M_\lambda(F,T,T_i))$: pushing T to T_λ, T_i toU_λ, and F to $F0$ and $F1$ respectively, and running M_λ ($|F|$ takes $log(|F|)$ so does not contribute significantly to space)
- $s(F_i) + s(T_i) + s(M_\sigma(F_i,T_i))$: determining if resulting localization F_i is satisfied in T_i.

We take the maximum space of localising/checking if localisation is satisfied, as one can reuse the space of the other, and furthermore this space can be reused amongst children of a given node.

$$s(M_\lambda(F,T,T_i) = s(F') + s(T_i) + s(M_\sigma(F',T_i))$$

where

- $s(F') + s(T_i) + s(M_\sigma(F',T_i))$: recursively calling M_λ,where F' is one of the extra formulas we need to determine the context for a temporal formula, e.g. $B_i \wedge \mathcal{H}A_i$or $\neg A_i$

Although we are iteratively computing localizations for larger subformulas of F for each of which we must make a call to $M_\sigma(F',T_i)$, this space can be reused between the subformulas of F.

So by expanding the above we get
$s(M_\sigma(F,T))$
$= s(T.children) + max(s(T) + s(T_i) + 2 * s(F) + s(F') + s(T_i) + s(M_\sigma(F',T_i)),\ s(F_i) + s(T_i) + s(M_\sigma(F_i,T_i)))$
$= s(T.children) + max(3 * s(T) + 3 * s(F) + s(M_\sigma(F',T_i)), s(F_i) + s(T_i) + s(M_\sigma(F_i,T_i)))$
(as $|F_i| = O(|F|)$, $|F'| = O(|F|)$ by sup. arg. 1, and T, T_i are both nid's of the same size)
$\leq s(T.children) + 3 * s(T) + 3 * s(F) + s(M_\sigma(|F+2|,T_i))$
(as $s(M_\sigma(F_i,T_i) \leq s(M_\sigma(|F+2|,T_i)$ trivially, and $s(M_\sigma(F',T_i) \leq s(M_\sigma(|F+2|,T_i)$ by sup. arg. 2)[6]
$\leq s(T.children)+3*s(T)+3*s(F)+s(T_i.children)+3*s(T_i)+3*s(|F+2|)+s(M_\sigma(|F+2+2|,T_{ij}))$
$\leq d(T) * |T.maxNoChildren| * s(T) + 3 * d(T) * s(T) + 3 * d(T) * (s(F) + 2 * d(T))$
where $d(T)$ is the depth of T and is polynomial in m, $|T.maxNoChildren| = poly(m)$ by sup. arg. 3, $s(T) = poly(m)$ by sup. arg.4. So whole machine takes space polynomial in m,f.

Supplementary arguments (sup. arg.):

1. $|F'| = O(|F|)$*:* Let A and B be formulas, and let $F = A\mathcal{U}B$. then $F' = B \wedge \mathcal{H}\mathcal{A}$ or $F' = \neg A$. clearly $|\neg A| = O(|F|)$. On the other hand $|B \wedge \mathcal{H}A| = |B \wedge (A\mathcal{Z}\bot)| = |A\mathcal{U}B| + 2 = O(|F|)$. Same argument applies for $\mathcal{W}, \mathcal{S}, \mathcal{Z}$.
   $|F_i| = O(|F|)$*:*this follows from the fact that F_i is just F but with some temporal operators replaced with their weaker/stronger counterparts.

[6] *$M_\sigma(|F+2|,T_i)$ is a call to M_σ on a formula of length 2 longer than F*

2. $s(M_\sigma(F', T) \leq s(M_\sigma(|F + 2|, T)$: Let A and B be formulas, and let $F = A\mathcal{U}B$. Then $F' = B \wedge \mathcal{H}\mathcal{A}$ or $F' = \neg A$. The case for $F' = \neg A$ is trivial. For $F' = B \wedge \mathcal{H}A$, we have $|F'| = |B \wedge \mathcal{H}A| = |B \wedge (A\mathcal{Z}\bot)| = |A\mathcal{U}B|+2$. Recall the high level overview of the algorithm given at the start of this chapter. In order to compute $\sigma(F', T) = \sigma(B \wedge (A\mathcal{Z}\bot), T)$, we need to recursively compute $\sigma(\bot \wedge \mathcal{G}A, T_i) = \sigma(\bot \wedge (A\mathcal{W}\bot), T_i)$ for some child T_i of T (in order to localise $(A\mathcal{Z}\bot)$). However $\bot \wedge (A\mathcal{W}\bot)$ is no longer than F', so the recursive calls to σ generated by $\sigma(F', T)$ are all on formulas of size no more than $|F + 2|$. Subsequently $s(M_\sigma(F', T) \leq s(M_\sigma(|F + 2|, T)$.

3. $|T.maxNoChildren| = poly(m)$:

   (a) The maximum number of children of any node in ωmodel T^ω is $O(m)$.

   (b) The maximum number of children of any node in T is polynomial in the maximum number of children of any node in T^ω(by construction of T).

   (c) Thus maximum number of children of any node in T (denoted $|T.maxNoChildren|$) is $poly(m)$ (by a, b).

4. for any nid of a node in T , $s(T) = |nid| = log(m') = log(exp(m)) = poly(m)$.

5.3.3 Conclusion

Proposition 5.14. *MODELSAT is in PSPACE*

Proof. : By composing the machine TM_o described in proposition 5.10 with the machine M described in the previous section. $TM_o \cdot M$ decides MODELSAT. However the model T produced by TM_o on input model T^ω is exponentially larger than T^ω, thus storing it would use too much space; instead we don't store it, but use the phantom input tape trick from the proof of logspace being closed under composition from [5], p.164. The basic idea is that we remove M's input tape, and instead maintain a counter i representing the position of the head in the "phantom" input tape (which is incremented/decremented accordingly by M to reflect a change of the head's position). We start by running M. As soon as M requires the current symbol from it's phantom input tape, we put M on hold, and run TM_o, only outputting the ith bit of it's output (we only need a logspace counter to determine the ith bit). As we are no longer storing the output of TM_o, overall space usage is just the sum of the space usage of TM_o and M (the counter i takes space logarithmic in the size of T, so polynomial in the size of T^ω). TM_o takes polynomial space, and M takes space polynomial in the size of the input to TM_o. Subsequently the whole machine $TM_o \cdot M$ takes space polynomial in the size of it's input. □

Chapter 6

Conclusion

Evaluation of contributions and discussion of future work

The first contribution of this project is a new algorithm for determining satisfiability of formulas of $L(\mathcal{U}, \mathcal{S})$ in $L(\mathcal{F}, \mathcal{P})$, as well as an implementation of said algorithm. The implementation is definitely valuable as a tool for introducing the algorithm to others, as well for performance analysis of the algorithm itself. Whether its value extends beyond this into the realm of actual model-checking applications is currently unknown. Model checking generally models systems as finite state processes, and so is concerned with determining the satisfiability of formulas in models specified as finite state processes as opposed to model expressions (essentially expression trees). It would be interesting to investigate the relationship between the two specification approaches, with a view to determining the potential applicability of our algorithm to model checking. As a result of the performance analysis section, we already know the kind of reasoning for which the algorithm is suited, namely reasoning over formulas where the number of temporal operators is relatively low.

Of particular interest are our complexity results. As a result of proposition 5.13, we know that the problem of satisfiability is solvable in polynomial time when the number of ω's in model expressions is bounded. This implies that the problem of satisfiability is in P if we restrict input models to contain only the operators $+, < \ldots >$. Thus the first question raised is whether we might be able to get a similar classification for other subsets of operators. If for example we considered model expressions containing only ω's , could it be that we can again achieve a lower complexity than for unrestricted model expressions? In a similar vein it would be interesting to consider the complexity of the algorithm for other temporal operators. For example it might be that if we just consider formulas of $L(\mathcal{F}, \mathcal{P})$ the algorithm allows us to decide satisfiability in a lower complexity class than for $L(\mathcal{U}, \mathcal{S})$. Finally, it would be interesting to consider extending the algorithm to handle formulas of the μ-calculus.

However the main extension to the project is to answer the obvious question raised by our most significant result. As a result of proposition 5.14 we showed that the problem MODELSAT of determining whether a given formula in $L(\mathcal{U}, \mathcal{S})$ is satisfied in a model expression is in PSPACE. This is a new complexity characterisation of a significant problem. This result immediately raises the question of whether or not MODELSAT is PSPACE-COMPLETE. Apart from giving us an even more precise characterisation of MODELSAT's complexity, proving PSPACE-COMPLETEness would also imply optimality of our algorithm (with respect to the complexity class hierarchy). To prove it we would have to show that MODELSAT is in fact PSPACE-HARD, as this would imply that MODELSAT is one of the "hardest" problems in PSPACE, and thus unlikely to be solvable by an algorithm in any complexity class believed to be smaller than PSPACE. Proving PSPACE-HARDness consists in identifying a PSPACE-

COMPLETE problem which can be reduced to MODELSAT in polynomial time, i.e. finding a language $L \in PSPACE - C$ and a function $f(x)$ computable in polynomial time such that $x \in L \iff f(x) \in MODELSAT$. Furthermore, as a result of proposition 5.13 we have some clues to the nature of f; namely we know that we should not be able to bound the number of omegas contained in the model part M of $f(x)$, as given that $f(x)$ is computable in polynomial time this would imply that L is in fact in P (which is false unless P=PSPACE) . Bearing this in mind, the main extension to our work would be to identify such a function f, proving that MODELSAT is PSPACE-COMPLETE.

Bibliography

[1] Tim French, John McCabe-Dansted, Mark Reynolds, Synthesis and Model Checking for Continuous Time, 2012

[2] Rosenstein J., Linear Orderings, Academic Press, 1982

[3] Burgess, J. P. and Y. Gurevich, The decision problem for linear temporal logic, Notre Dame J. Formal Logic 26 (1985), pp. 115-128.

[4] Hodkinson I., Modal and Temporal Logic Lecture Notes,Imperial College London, 2012

[5] Papadimitriou, C. H., Computational Complexity, Addison-Wesley, 1994

[6] Pnueli, A., The temporal logic of programs, in: Proceedings of the Eigh- teenth Symposium on Foundations of Computer Science, 1977, pp. 46- 57

[7] Reynolds, M., The complexity of the temporal logic over the reals, Annals of Pure and Applied Logic 161 (2010), pp. 1063-1096

[8] A. Sistla, E. Clarke, Complexity of propositional linear temporal logics, J. ACM 32 (1985) 733–749.

[9] R. Dedekind, Continuity and Irrational Numbers, Essays on the Theory of Numbers, part I, Open Court Publishing Company, 1901

[10] Phillips I., Complexity Lecture Notes,Imperial College London, 2012

[11] Gabbay, D., Hodkinson, I., Reynolds, M., Temporal Logic: Mathematical Foundations and Computational Aspects, Volume 1. Oxford Science Publications, 1994

[12] Lichtenstein, O., Pnueli, A., Checking That Finite State Concurrent Programs Satisfy Their Linear Specification, POPL '85 Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages pp. 97-107 (1985)

[13] Kamp, H., Tense logic and the theory of linear order, Ph.D. thesis, University of California, Los Angeles (1968)

[14] http://en.wikipedia.org/wiki/Model_checking

[15] http://en.wikipedia.org/wiki/Temporal_logic

Index