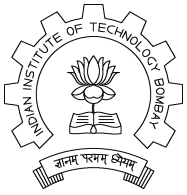
Dense Time Logics

Thesis
Submitted in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

by

Mohsin Ahmed



Department of Computer Science and Engineering,
Indian Institute of Technology, Bombay, INDIA.
1993

APPROVAL SHEET

Thesis entitled: *Dense Time Logics* by *Mohsin Ahmed*

is approved for the degree of **DOCTOR OF PHILOSOPHY** .

Examiners

_____

_____

Advisor

_____

Chairman

Date: _____ _____

To my parents,

Sofiya and Shafi.

Abstract

This report investigates the automation of temporal reasoning, by extending classical logics with temporal modalities. These modalities can arbitrarily refine the time division leading to a dense model of time based on infinite binary tree. This allows the reasoner to impose a hierarchy on time and deal with it in a modular fashion at several levels.

We propose Propositional Dense Time Logic (PDTL), and show how it can be decided in exponential time using a tableau based method. PDTL is extended to Propositional Ordinal Tree Logic (POTL) by restricting the temporal propositions to have certain stability and recurrence properties. POTL too is shown to be decidable in exponential time by modifying the tableau method of PDTL.

We give a prolog implementation of a Dense Time Logic Programming (DTLP) language, which treats models of POTL as temporal horn clauses. The DTLP interpreter can answer temporal queries from temporal facts and temporal rules.

The relation of these logics with that of Propositional Dynamic Logic and Monadic Second Order Logic with Two Successor functions is discussed and various language issues like cuts, rigid and flexible variables are reviewed in the temporal context.

Keywords: Temporal Logic, Dense Time, Decidability, Temporal Logic Programming.

Contents

1 Introduction **1**

1.1 Propositional Dense Time Logic 2

1.2 Propositional Ordinal Tree Logic 3

1.3 Dense Time Logic Programming 4

1.4 Related Work 6

1.4.1 Theory 6

1.4.2 The AI approach 6

1.4.3 Discrete Time Programming 7

1.5 Our approach 7

1.6 Organization of the Report 8

2 Preliminaries **9**

2.1 Terminology 9

2.2 Proposition Logic 10

2.3 First Order Logic 11

2.4 Monadic Second Order Logic 11

2.5 Propositional Modal Logic 12

2.6 Propositional Temporal Logic 14

2.7 First Order Temporal Logic 15

2.8 Propositional Dynamic Logic 15

2.9 Logic Programming 16

2.10 Temporal Logic Programming 17

3 Propositional Dense Time Logic **19**

3.1 The Language 19

3.2 Semantics 20

3.3 Examples 21

3.4 Satisfiability and Tableaux 22

3.4.1 Tableaux Building 22

3.4.2 Closing Nodes 23

3.4.3 State Node Tableau 24

3.5 Axiomatization of PDTL 27

3.5.1 Derived Rules of Inference and Theorems of PDTL 29

3.6 Completeness 32

4 Propositional Ordinal Tree Logic **39**

4.1 Ordinal Trees 39

4.1.1 Language of Ordinal Trees 41

4.2 Tableau and Satisfiability 41

4.2.1 Stable Nodes 42

4.2.2 Closing Nodes 42

4.2.3 State Node Tableau 42

4.2.4 Marking Nodes 43

5 Decidability by Interpretations **48**

5.1 Monadic Second Order Logic of Two Successor Functions 48

5.1.1 Definable sets in S2S 48

5.1.2 Sets defined by Regular Expressions 49

5.1.3 Decidability of S2S 50

5.1.4 Decidability by Interpretation in S2S 50

5.2 Decidability by Interpretation in PDL 51

6 Dense Time Logic Programming **52**

6.1 Omega Trees 52

6.1.1 Labeling nodes and Variables 53

6.2 Ordinal Trees 53

6.3 Representing Ordinal Trees in Prolog 55

6.4 Standardizing Ordinal Tree Representation 55

6.5 Temporal Clauses 57

6.5.1 Facts 57

6.5.2 Queries 57

6.5.3 Rules 58

6.6 Temporal Resolution 59

6.6.1 The Aligning Algorithm 59

6.6.2 Aligning Examples 60

6.7 Extensions to DTLP 62

6.7.1 Existential Queries 62

6.7.2 Unanswerable Queries 63

6.7.3 Flexible Variables 64

6.7.4 Temporal Terms 65

6.7.5 Clock Trees 65

6.7.6 Operator Grammars 66

7 Conclusion **67**

A Testing the DTLP Interpreter **72**

A.1 Test Programs 72

A.1.1 Program 1 72

A.1.2 Program 2 73

A.1.3 Program 3 73
A.1.4 Program 4 74
A.1.5 Program 5 74
A.2 Trace of Test Programs 75
A.2.1 Trace 1 75
A.2.2 Trace 2 76
A.2.3 Trace 3 78
A.2.4 Trace 4 80
A.2.5 Trace 5 81

B DTLP Interpreter Listing **84**

List of Tables

3.1 Expansion table for logical nodes 23

3.2 Closing Algorithm 24

3.3 Mapping $m : \mathcal{N}^+ \mapsto \Delta$. 26

4.1 Marking Algorithm 44

List of Figures

2.1 A kripke model 13

2.2 A PTL model 14

3.1 Omega tree 20

3.2 Tableau for $\{\boxplus p, \Diamond\!\!\!\!-\neg p\}$ 27

3.3 Tableau for $\{\Box p, \Diamond\!\!\!\!-\neg p\}$ 28

4.1 An ordinal tree 40

4.2 Ordinal tree tableau for $\boxplus(p \vee \neg p)$ 47

6.1 Omega tree 53

6.2 An ordinal tree 53

6.3 Nested loops 54

Chapter 1

Introduction

There is frequent need to qualify information with time, especially if we are talking about the real world and the events taking place in it. Consider, for example, the statement:

> "The ball hit the wall before rolling off into the drain."

This describes a sequence of events, the first of which consists of the ball hitting the wall, the last event describes the ball falling into the drain, and the events in-between describe the ball rolling. A convenient way of describing this information is to say what happened in each of the three consecutive intervals: one for striking, one for rolling and one for falling. The rolling interval can be further refined to describe what occurs within it. While a rolling interval may not be particularly eventful, we could instead consider the example:

> "After hitting the wall, the ball bounced on the ground before falling into the drain."

Here, the intervening interval consists of an unknown number of bouncing events. As another example, consider the following temporal scenario:

> "It is raining heavily today,"

and the causal rule:

> "Whenever there is heavy rain, the janitor is absent the next day."

How do we conclude that:

> "The janitor will be absent tomorrow" ?

While there have been a number of studies in representing temporal information within a logical framework [GPSS80, BAPM83, LPZ85, Wol83, Lam90, AV93c, Ahm93], many of these deal with a time model which is discrete (i.e. consists of a sequence of states). This does not allow us to refine the time between two states. In particular, we cannot interpolate an unbounded (or infinite) sequence of states between two states. Such a requirement also arises when we wish to compose specifications of systems [Lam90]. We do not focus on this issue, but instead look at the issue of representing dense time information.

1.1 Propositional Dense Time Logic

We present a temporal logic based on a model of time which allows arbitrary nesting of sequences. To be more precise, our model consists of a sequence of states at the top level. Between any two states of a sequence, another sequence of states can be interpolated. This model can be represented as an infinite binary tree, and is called the *omega tree* model.

Two new operators are introduced into the propositional linear time temporal logic (PTL) [GPSS80]. The new operators $\oslash$ (read *within*) and $\boxplus$ (read *everywhere*), allow us to refine an interval and describe it in more detail. The dual of $\boxplus$ is $\diamondplus$ (read *somewhere*). This gives us the ability to represent dense time information. A behavior consisting of a sequence of events terminating in a limit point can also be expressed, for example a bouncing ball coming to rest can be represented by

$$\oslash(\oslash Up \wedge \Box(\oslash Up \leftrightarrow \bigcirc \oslash Down)) \wedge (\bigcirc \boxplus Down) \wedge \boxplus(Down \leftrightarrow \neg Up),$$

where Up and $Down$ are propositions. This formula can be read as:

> Within the first interval the ball is initially up, and it alternates between up and down. From the next interval onwards it remains down, and at every time point the ball is either up or down.

Such a sequence can be nested in a bigger sequence of events, as in the statement

> 'Every time he hit the ball, the ball bounced and came to rest.'

This statement can represented by

$$\Box(\oslash Hit \rightarrow \oslash \phi),$$

where ϕ is the formula given before, and Hit is a new propositional symbol.

We show that the resulting logic, called Propositional Dense Time Logic (PDTL), is decidable in exponential time by a tableau based decision procedure similar to that used for deciding the

satisfiability of PTL formulas [LPZ85, Wol83]. A simple axiomatization for PDTL is given and its completeness is shown [AV93b].

Though PDTL is reducible to Propositional Dynamic Logic (PDL) and hence its decidability follows from that of PDL[Har84, KT90], the *within* and *next* operators of PDTL have a direct temporal interpretation in terms of nested sequences. Moreover the tableau based method is intuitively more appealing, and is a direct extension of the methods used for PTL [Wol83, LPZ85].

PDTL allows us to write a satisfiable sentence:

$$\boxplus(\diamond\!\!\!\!-\!\!\!\!\diamond p \wedge \diamond\!\!\!\!-\!\!\!\!\diamond \neg p)$$

which describes a dense mix of states that satisfy p and $\neg p$. Such formulas contribute to unwieldy models and rarely occur in practice [Gal87b, Bar87, Ham71, Pri67].

The dense mix has been described as *fuzz* by Prior [Pri67] and *indefinitely fine intermingling* by Hamblin [Ham71]. McDermott [McD82] uses an axiom to rule this out. Barringer applies a *finite variability* restriction to to rule out a proposition changing its value infinitely within a finite interval. In [Gab91] Gabbay argues that temporal logics should account for stability and recurrence properties of time.

To filter out such models, we define *stability* of propositions. We call a proposition p *stable* in an omega tree if there is no dense mixture of p and $\neg p$ in the omega tree.

1.2 Propositional Ordinal Tree Logic

Ordinal Trees are omega trees where all the propositions are stable, and every right branch of the omega tree becomes periodic. We call this logic *propositional ordinal tree logic* (POTL).

Ordinal trees are omega trees, which can be represented by finite binary trees with back-arcs allowed in a restricted way so that infinitely nested sequences do not appear. The logic of formulas valid over ordinal trees (POTL) is shown to be decidable by extending PDTL's tableau based procedure. Here nodes which do not contain any ordinal tree models are also closed. Thus POTL is also decidable in exponential time, as compared to deciding it in non-elementary time by interpreting it in *monadic second order logic with two successor functions* (S2S) [Rab69]. The decidability results for POTL hold even if only some of the propositions are stable.

An ordinal tree can represent a nested infinite sequences of events such as:

$$a \cdot b^{\omega} \cdot c^{\omega^2} \cdot b,$$

where a, b and c are some events, a^k denotes a occurring k times, and a^{ω} denotes a occurring an *infinite* number of times.

By using an ordinal tree as a temporal data structure we can represent and manipulate temporal information in a simple way within any logic programming language [AV93a].

1.3 Dense Time Logic Programming

While computing with static information has been successful, with the advent of logic programming languages there is no parallel for handling temporal information. Programming languages have provided a wide range of abstract data structures like arrays, lists, trees, graphs, stacks and queues, which are useful to model a large variety of real world data. However none of these are convenient for representing temporal information and typically temporal information is still represented with time stamps and other ad hoc methods. Moreover every user must have his own means to manipulate temporal information without any help from the system which the system treats at par with any other static data type. For example, if the user represents a clock using integers, then the system allows all arithmetic operations on the clock. It would be more appropriate if the system treated temporal objects as abstract data types and permitted only meaningful operations on them.

In this direction, we investigate the usefulness of an ordinal tree as a data structure to represent dense time information. An ordinal tree is an infinite binary tree represented by a directed graph with some regularity properties. It is made up of three kinds of nodes:

leaf nodes that represent a single interval,

split nodes that breakup an interval into 'present' and 'future' intervals; and

loop nodes that denote repetitive intervals.

Ordinal trees break up the rational line recursively into a hierarchy of infinitely nested sequences of time points. Each node of the ordinal tree represents a closed-open interval, which can be labeled by a prolog clause, to assert facts or make queries about that interval. Ordinal trees can assert temporal information like:

"The ball hit the wall before rolling into the drain",

and ask queries like:

"Did the ball roll into the drain after hitting the wall?".

Recursive computations can be modeled naturally using ordinal trees reflecting the natural procedural control mechanisms.

Given two ordinal trees, they can be re-structured so that their intervals are *aligned*. Tree aligning is the basis for transferring temporal information between two labeled ordinal trees. We give an algorithm to *align* any two ordinal trees, whose complexity is $O(mn)$ where m and n are the number of nodes of the two ordinal trees. However in most cases, the complexity is linear i.e. $O(max(m, n))$.

An ordinal tree can be represented as a set of clauses, each clause contains a set of temporal predicates. A temporal predicate represents information about some node of the ordinal tree. If a temporal clause has at most one non-negated temporal predicate, it is called a *temporal horn clause*.

Temporal resolution works on a query clause trying to align it with temporal facts and rules, expanding the leaves of the query to fill in details and performing other book-keeping until the whole query clause can be proved. For example, if we are given the following rule:

"Whenever it rained, the janitor was absent the next day"

and the fact:

"It rains every Tuesday".

Then the following query will succeed:

"Was the janitor absent on Wednesday?"

The rule is an example of a *causal rule*, as the causes temporally precede the effects.

Queries can be temporally existential, asking if something ever happened. For example, a query can ask

"Was the janitor absent on some day ?"

Queries can include *cuts*, using which we can ask:

"Did it flood after the first rains ?"

DTLP variables once instantiated retain their value over all time, and are usually known as *rigid* variables and are useful in computation in temporal queries.

However there are many queries that cannot be asked yet. Therefore we suggest several extensions to enhance the power of DTLP.

1. Variables that can take different values at different times are called *flexible variables*, they add to the expressive power of DTLP. Arithmetical relations on temporal terms involving flexible variables can express many interesting time varying sequences.

2. Ordinal tree allows the representation of information that uses either a finite sequence of events or an infinite periodic sequence of events. In some applications it would be more realistic to have a finite division of time at certain levels and also be able to represent finite sequences in a periodic fashion. We achieve this by simply re-interpreting ordinal trees as *clock trees*, which count time modulo some fixed number at each level. Clock trees permit only finite sized sequences at certain depths. For example, consider time to be an infinite sequence of days, where each day is divided into 24 hours, and each hour is divided into 60 minutes, then the following two facts:

   "Every seventh day is a holiday",
   "It rains for one hour from 5PM, every second day"

   and the rule:

   "Whenever it rains for a full hour, the buses are late for the next one hour"

   can all be conveniently represented as clock trees. Aligning clock tree involves more complex re-structuring than required for ordinal trees. In the above example, if a temporal rule involving two consecutive hours is to be applied on the boundary of two days, then the rule requires some more re-structuring.

1.4 Related Work

1.4.1 Theory

Several researchers have tried to model temporal phenomena [GPSS80, LPZ85, Lam90, SC85, Wol83, AV93b]. Linear, branching, and tree based temporal logics have been surveyed in [vB83, vB89, Bur84, Eme90]. Logics based on infinite trees have been surveyed in [Tho90, Har84]. Many temporal and modal logics can be decided by interpreting them in S2S [Gab75]. S2S is decidable in non-elementary time [Rab69]. Barringer et. al. have described a real time logic in [BKP86]. Alur and Henzinger have shown another real time logic to be undecidable[AH90] .

1.4.2 The AI approach

Logic of action and change has been widely studied in Artificial Intelligence. Shoham has studied temporal phenomena from the model theoretic point of view, and describes his logic of change in terms of minimal models [Sho89]. Temporal relations have also been used to model time by several researchers [All84, KS86, Gal87b, Ram89, Sar87, San93, Sow84].

1.4.3 Discrete Time Programming

Discrete time temporal logic programming has also been investigated by several researchers.

Gabbay has proposed [Gab87, Gab91] temporal deduction systems based on an extension of horn clauses, and in [Gab87, Gab91] he shows how temporal operators may be introduced in a conventional logic programming language.

Abadi and Manna show how prolog can be extended with *Propositional Temporal Logic* to obtain a temporal logic programming language called *Templog* [AM87]. All programs of *Templog* can also be written in DTLP. The completeness and expressive power of *Templog* is studied by Baudinet [Bau89].

Moszkowski and Hale have implemented an interval temporal logic programming language called *Tempura* [Mos86, Hal87]. A *Tempura* formula is also an interval temporal logic program and its execution gives rise to a model for the formula. Another temporal logic programming language *Tokio* is described in [AFMO85, FKTMO85]. It extends prolog with temporal constructs like *chop* (sequencing) to describe control. *Tokio* is based on unification and backtracking.

1.5 Our approach

Our method differs from others, in that we provide for the first time a dense time logic (POTL) that is:

- Theoretically interesting, since it is decidable in exponential time by a simple extension of the tableau method for *Propositional Temporal Logic*[AV93b].
- Practically interesting, as it gives rise to a temporal data structure that is useful in a logic programming language[AV93a].

DTLP differs from the other approaches to temporal logic programming as:

- It is based on logic programming like Gabbay's work and can be embedded in a language like Prolog.
- It models dense time and extends Abadi and Manna's *Templog*.
- It takes the data structure approach to model time.
- It can model activities, which may contain an infinite sequence of sub-activities.
- It can reflect the underlying hierarchy of time that we usually use to describe temporal information. For example: time can be organized into years, days and hours.

1.6 Organization of the Report

Chapter 1 provided the overall introduction. The preliminaries are given in Chapter 2. Chapter 3 and Chapter 4 discuss PDTL and POTL respectively. Chapter 5 shows how PDTL and POTL can be decided by interpreting them in S2S and PDL. Chapter 6 explains DTLP, the temporal logic programming based on POTL. Finally Chapter 7 concludes the report.

The references are followed by appendices giving test programs, their runs on the DTLP interpreter and the interpreter listing.

Chapter 2

Preliminaries

We present our notation and terminology in this chapter. The syntax and semantics of below mentioned logics is also reviewed:

- Propositional Logic (PL).
- First Order Logic (FOL) [Bar77, End72].
- Monadic Second Order Logic (MSO) [Rab69, Rab77, Tho90].
- Modal Logic [Che80, BS84, Boo79].
- Propositional Temporal Logic (PTL) [Wol83].
- First Order Temporal Logic (FOTL) [Sza87b].
- Propositional Dynamic Logic (PDL) [Par81, Har84, KT90].
- Logic Programming [CM84, Gal87a, Llo84].
- Temporal Logic Programming [AM87, Gal87b, Gab87, Gab91, Mos86].

2.1 Terminology

The *Language* ($\mathcal{L}$) of a logic consists of logical and non-logical constants together with a set of well formed formulas ($wff(\mathcal{L})$). The logical constants have fixed meaning in a logic, while the non-logical constants can have any interpretation in a logic. M is called a *model* for a set Γ of formulas ($\Gamma \subseteq wff(\mathcal{L})$) , if every $\phi \in \Gamma$ is interpreted as true in M; written as $\models_M \Gamma$. A set Γ *logically implies* ϕ (ϕ is a *logical consequence* of Γ), if ϕ is true in every model of Γ; written as $\Gamma \models \phi$. The set of

logical consequences of Γ is denoted by $Cn(\Gamma) = \{\phi : \Gamma \models \phi\}$. A formula is *valid* if it is true in all models ($\models \phi$), and *unsatisfiable* if it has no models. It follows that ϕ is valid iff $\neg\phi$ is unsatisfiable. A *rule of inference* is a rule to deduce true formulas from formulas already known to be true. A *proof* of ϕ from Γ is a finite deduction of ϕ from Γ using the inference rule and is denoted by $\Gamma \vdash \phi$. The set of formulas provable from Γ is denoted by $prv(\Gamma) = \{\phi : \Gamma \vdash \phi\}$.

A *theory* T is any set of sentences closed under logical consequences, that is $Cn(T) = T$. A *theory of a model* M is the set of sentences true in a model, denoted by $Th(M) = \{\phi : \models_M \phi\}$. A theory T is *inconsistent* if it contains all formulas in $wff(\mathcal{L})$, otherwise it is *consistent*. An inconsistent theory is unsatisfiable. It is *complete* if for every $\phi \in L$, ϕ or $\neg\phi$ is in T. Clearly for any model M, $Th(M)$ is complete. Theory T is (finitely) *axiomatizable* if there is a (finite) recursive set of axioms Ax, such that $Cn(Ax) = T$. A theory T with axioms Ax is *sound* if $prv(Ax) \subseteq T$. T is *partially decidable* (a recursively enumerable set), iff there is a recursive set of axioms Ax such that $Cn(Ax) = T$. It is *decidable* if T is a recursive set. Clearly, a partially decidable complete theory is decidable.

A logic is *compact*, if whenever $\Gamma \models \phi$ then there is a finite subset Γ' of Γ such that $\Gamma' \models \phi$. A theory T has *finite model property* (fmp), when every sentence not in T can be falsified in a finite model. If a complete theory has finite model property, then we can decide if a sentence is not in it by enumerating finite models till we find a model falsifying the sentence. Thus a complete theory having fmp is decidable, and so is a finitely axiomatizable theory having fmp [Gab73].

2.2 Proposition Logic

The language of PL has the following symbols $\neg$ (negation), $\rightarrow$ (implication), $\mathcal{F}$ (false) and a set of propositions $PROP = \{p_i : \ i \geq 0\}$. The set of formulas is the closure of $PROP$ and $\mathcal{F}$ under the logical operations. A model M is a set $s \subseteq PROP$, and truth values to formulas is assigned as follows:

$$\begin{array}{lll} \not\models_M \mathcal{F} & & \\ \models_M p,\ p \in PROP & \text{iff} & p \in s \\ \models_M \neg\phi & \text{iff} & \not\models_M \phi \\ \models_M \phi \rightarrow \psi & \text{iff} & \not\models_M \phi \text{ or } \models_M \psi \end{array}$$

The other operators and truth constants are defined as follows:

$$\begin{array}{lcl} \mathcal{T} & \stackrel{\text{def}}{=} & \neg\mathcal{F} \\ (\phi \vee \psi) & \stackrel{\text{def}}{=} & (\neg\phi \rightarrow \psi) \\ (\phi \wedge \psi) & \stackrel{\text{def}}{=} & \neg(\phi \rightarrow \neg\psi) \\ (\phi \leftrightarrow \psi) & \stackrel{\text{def}}{=} & (\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi) \end{array}$$

A PL formula is a *tautology* if it is true in every model. The inference rule used is *detachment*:

$$\text{MP.} \quad \frac{\vdash \phi \text{ and } \phi \to \psi}{\vdash \psi}$$

PL is compact, finitely axiomatizable and decidable in polynomial space and exponential time.

2.3 First Order Logic

The language of FOL has the following symbols: the logical connectives and truth constants of PL, quantifiers $\forall$ (for all) and its dual $\exists$ (there exists), equality $=$, a set $X = \{x_i : i \geq 0\}$ of individual variables.

The non-logical symbols are sets of constants, predicates and functions. *Terms* are generated by closing constants and variables under functions. *Atomic formulas* are obtained by equating terms, and $\mathcal{F}$ is an atomic formula. The set of formulas is the closure of atomic formulas under the logical operations and quantification by variables. A variable x occurring in ϕ is *bound*, if it is in the scope of $\exists x$ or $\forall x$ quantifier, otherwise it is called *free*. The set of free variables of ϕ is denoted by $FV(\phi)$. A *sentence* (or *closed-formula*) is a formula without free variables.

A structure M consist of a non-empty set U (called the *domain* of interpretation), where the constants, predicates and functions are interpreted as elements, relations and functions (of appropriate arity) on U. The meaning of quantifiers is as follows:

$$\begin{array}{lll} \models_M \forall x \phi(x) & \text{iff} & \forall b (b \in U \to \models_M \phi(b)) \\ \exists x \phi(x) & \stackrel{\text{def}}{=} & \neg \forall x \neg \phi(x) \end{array}$$

$\models_M (x = y)$ iff x and y represent the same element in U.

The inference rules used are MP and *generalization*:

$$\text{Gen.} \ \frac{\vdash \phi(x)}{\vdash \forall x \phi(x)}$$

FOL is compact, finitely axiomatizable and undecidable.

2.4 Monadic Second Order Logic

Monadic Second Order (MSO) logic extends first order logic with quantifiable monadic predicates (set variables). The language of MSO logic consist of PL connectives and truth constants, along with equality: '='; individual variables: $\{x_i : i \geq 0\}$; individual constants; functions; predicates; and quantifiable monadic predicates: $\{\bar{A}, \bar{B}, \ldots\}$. The quantifiers $\exists$ and $\forall$ range both over individual variables and monadic predicates.

Terms are generated from constants and individual variables by application of functions. Atomic formulas are generated from terms by applying predicates and equality. Formulas are generated from atomic formulas by applying logical connectives and quantifiers.

By treating monadic predicates as sets, the following can be defined:

$$\begin{array}{lcl} x \in A & \stackrel{\text{def}}{=} & \models \bar{A}(x) \\ x \in (A \cup B) & \stackrel{\text{def}}{=} & x \in A \vee x \in B \\ x \in (A \cap B) & \stackrel{\text{def}}{=} & x \in A \wedge x \in B \\ x \in A - B & \stackrel{\text{def}}{=} & x \in A \wedge x \notin B \\ A \subseteq B & \stackrel{\text{def}}{=} & \forall x(x \in A \rightarrow x \in B) \\ A = B & \stackrel{\text{def}}{=} & (A \subseteq B) \wedge (B \subseteq B) \end{array}$$

2.5 Propositional Modal Logic

This is an extension of PL by the modalities for *necessity* and *possibility*, denoted by $\Box$ and $\Diamond$ respectively. The language now allows us to qualify formulas with them. We will consider *normal* modal logics, that satisfy Kripke's axiom:

$$K. \quad \Box(\phi \rightarrow \psi) \rightarrow \Box\phi \rightarrow \Box\psi$$

Normal modal logics have simple frame based models called *kripke* models.

Definition 1 (Kripke Model) *A* kripke *model is a structure* $M = (PROP, W, R)$*, where*

- $PROP$ *is a set of propositions.*
- $W = \{w : w \subseteq PROP\}$ *is a set of worlds. A proposition* $p \in PROP$ *is said to hold at* w *iff* $p \in w$*.*
- $R \subseteq W \times W$ *is a binary accessibility relation on the worlds. This relation completely characterizes the model. The* possible worlds *of* w *is the set* $\{v \in W : R(w, v)\}$*.*

The semantics are:

$$\begin{array}{lcl} \models_M \phi & \text{iff} & \forall v \in W : \models_{(M,v)} \phi \\ \models_{(M,w)} \Box\phi & \text{iff} & \forall v : R(w, v) \rightarrow (\models_{(M,v)} \phi) \\ \Diamond\phi & \stackrel{\text{def}}{=} & \neg\Box\neg\phi \end{array}$$

The rules of inference are MP and *necessitation*:

$$\text{RN.} \quad \frac{\vdash \phi}{\vdash \Box\phi}$$

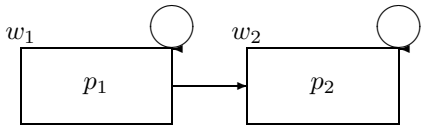


Figure 2.1: A kripke model

Example 1 (A kripke model) Figure 2.1 shows a kripke model $M = (PROP, W, R)$. The set of propositions is $PROP = \{p_1, p_2\}$, and the set of worlds is $W = \{w_1, w_2\}$. The propositions p_1 and p_2 hold in w_1 and w_2 respectively. The possible worlds of w_1 is the set $\{w_1, w_2\}$; and that of w_2 is $\{w_2\}$. This is defined by the relation $R = \{(w_1, w_1), (w_2, w_2), (w_1, w_2)\}$.

Since R is reflexive and transitive, we have:

$$\begin{aligned}
&\models_M && (\Box A \to A) \wedge (\Box A \to \Box\Box A) \wedge (A \to \Diamond A)\\
&\models_{(M,w_1)} && p_1 \wedge \Diamond p_1 \wedge \Diamond p_2 \wedge \Box(p_1 \vee p_2)\\
&\models_{(M,w_2)} && p_2 \wedge \Box p_2 \wedge \Diamond p_2
\end{aligned}$$

Several interesting class of kripke models can be described by suitable axioms, we first list out the axioms and the conditions they force on the relation R [Che80, BS84]:

D.	$\Box A \to \Diamond A$,
	Serial: $\forall w_1 \exists w_2 : R(w_1, w_2)$
T.	$\Box A \to A$,
	Reflexive: $\forall w : R(w, w)$
4.	$\Box A \to \Box\Box A$,
	Transitive: $\forall w_1, w_2, w_3 : (R(w_1, w_2) \wedge R(w_2, w_3)) \to R(w_1, w_3)$
$E, 5$.	$\Diamond A \to \Box\Diamond A$,
	Euclidean: $\forall w_1, w_2, w_3 : (R(w_1, w_2) \wedge R(w_1, w_3)) \to R(w_2, w_3)$
B.	$A \to \Box\Diamond A$,
	Symmetric: $\forall w_1, w_2 : R(w_1, w_2) \to R(w_2, w_1)$
M.	$\Box\Diamond A \to \Diamond\Box A$
G.	$\Diamond\Box A \to \Box\Diamond A$
H.	$(\Diamond A \wedge \Diamond B) \to (\Diamond(A \wedge B) \vee \Diamond(A \wedge \Diamond B) \vee \Diamond(\Diamond A \wedge B))$
Grz.	$\Box(\Box(A \to \Box A) \to A) \to A$
Dum.	$\Box(\Box(A \to \Box A) \to A) \to (\Diamond\Box A \to A)$
W.	$\Box(\Box A \to A) \to \Box A$

These axioms give rise to the following modal logics:

$KT4$	$S4$
$KT4B$	$S5$
$KT4M$	$S4.1$
$KT4G$	$S4.2$
$KT4H$	$S4.3$
KW	Löb

2.6 Propositional Temporal Logic

The language of *propositional temporal logic* is an extension of the language of propositional logic with the following symbols: $\Box$ (henceforth), $\Diamond$ (eventually), $\bigcirc$ (next), $\sqcup$ (until), and *atnext* (atnext). A model M for PTL is an infinite linear sequence of worlds $< s_0, s_1, \ldots >$, where each world is a propositional logic structure representing a time point. Truth is assigned to formulas as follows:

$$
\begin{array}{lll}
\models_M \phi & \text{iff} & \models_{s_0} \phi \\
\models_{s_i} \Box\phi & \text{iff} & \forall k \geq 0, \models_{s_{(i+k)}} \phi \\
\models_{s_i} \Diamond\phi & \text{iff} & \exists k \geq 0, \models_{s_{(i+k)}} \phi \\
\models_{s_i} \bigcirc\phi & \text{iff} & \models_{s_{(i+1)}} \phi \\
\models_{s_i} \phi \sqcup \psi & \text{iff} & \exists k \geq i : \forall m \in \{i, \ldots, k-1\} \models_{s_m} \phi \\
 & & \text{and } \models_{s_k} \psi \\
\models_{s_i} \phi \, atnext \, \psi & \text{iff} & \exists k \geq i : \forall m \in \{i, \ldots, k-1\} \models_{s_m} \neg\phi \\
 & & \text{and } \models_{s_k} (\phi \wedge \psi)
\end{array}
$$

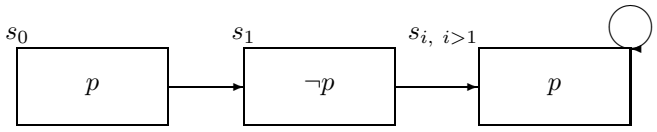


Figure 2.2: A PTL model

Example 2 (A PTL model) A model M for PTL is shown in figure 2.2. The proposition p holds in all worlds except in s_1. Then the following formulas hold:

$$
\begin{array}{ll}
\models_M & p \wedge \bigcirc\neg p \wedge \Diamond\Box p \\
\models_{(M,s_2)} & \Box p
\end{array}
$$

2.7 First Order Temporal Logic

FOTL is the first order version of PTL. The language includes constants, functions, variables, equality, predicates, propositional connectives, quantifiers, and PTL modalities. There are two types of variables: quantifiable global variables $X = \{x_i : i \geq 0\}$, and state variables $Z = \{z_i : i \geq 0\}$ which cannot be quantified.

A model for it is an infinite linear sequence of states: $< s_0, s_1, \ldots >$, where each state is a first order structure. The domain of interpretation (U) of the first order structure is the same in each state. The constants, functions and predicates have the same interpretation in each state. Every state can independently assign values to the state variables from the domain. The meaning of logical constants are those from PL, PTL and FOL, (except that of the quantifiers). The meaning of quantifiers depend on the temporal context:

$$\models_{s_0} \Box\exists x\phi(x) \quad \text{iff} \quad \forall i \geq 0, \exists b_i \in U \models_{s_i} \phi(b_i)$$
$$\models_{s_0} \exists x\Box\phi(x) \quad \text{iff} \quad \exists b \in U, \forall i \geq 0 \models_{s_i} \phi(b)$$

Example 3 (Finite Models) The following sentence has only models with finite domains:

$$(a = 0) \wedge \forall z \Diamond (a = x) \wedge \Diamond\Box(a = 0)$$

Here 'a' is a *state* variable that is initially zero, visits every element in the domain in a finite number of states, and finally equals zero permanently.

The looping problem of a turing machine can be expressed as a FOTL formula, this formula is valid iff the turing machine loops. Hence FOTL is undecidable [SH88]. FOTL with addition and multiplication is strongly incomplete, as the theory of numbers is finitely axiomatizable in it [Sza87a].

2.8 Propositional Dynamic Logic

PDL has been used to reason about programs. Here programs are treated as modalities, whose action on a state lead to another state(s). A model is a set of states, with the programs relating the states. A program may be nondeterministic, if its action can lead to multiple states.

The language of PDL consists of *formulas* and *programs*. Let $PROP$ be the set of atomic formulas, and r_0, r_1 be the two atomic programs. Then the set of programs and formulas are defined by mutual induction: If A, B are formulas and a, b are programs then: $A \vee B$, $\neg A$, $<a>A$, $[a]A$ are formulas, and $a; b$, $a + b$, a^*, $A?$ are programs.

These new operations can be intuitively described as follows:

Possibility: '$<a>A$'

It is possible to execute a and terminate in a state satisfying A.

Necessity: '$[a]A$'

All termination executions of a terminate in a state satisfying A. Note that a may not terminate.

Sequence: '$a; b$'

Execute a and then execute b.

Choice: '$a + b$'

Execute a or b nondeterministically.

Iterate: 'a^*'

Execute a zero or more times.

Test: '$A?$'

Proceed if A holds, else fail.

The following operators and identities can be defined in PDL:

$$\begin{array}{lcl} [a]A & \stackrel{\text{def}}{=} & \neg<a>\neg A \\ \{A\}\, a\, \{B\} & \stackrel{\text{def}}{=} & A \rightarrow [a]B \\ \text{if } A \text{ then } a \text{ else } b & \stackrel{\text{def}}{=} & (A?; a) + (\neg A?; b) \\ \text{while } A \text{ do } a & \stackrel{\text{def}}{=} & (A?; a)^*; (\neg A?) \end{array}$$

PDL is not compact. PDL is decidable in exponential time in the size of the formula.

2.9 Logic Programming

A class of formulas amenable to fast automated deduction are the horn clause programs [CM84, Gal87a, Llo84]. A *clause* C is a disjunction of a set of signed atomic formulas: $C = (A_1 \vee \ldots \vee A_j \vee \neg B_1 \vee \ldots \vee \neg B_k)$. It also written as: $\{A_1, \ldots, A_j\} \leftarrow \{B_1, \ldots, B_k\}$, where the set on the left is called the *head* and the set on the right is called the *body* of the clause.

'C' is a *horn clause* if $j \leq 1$, that is, it has at most one positive atomic formula. A *fact* is of the form '$A \leftarrow$'. A *query* is of the form '$\leftarrow B_1, \ldots, B_k$', where each of the B_is are called the *goals*. A *rule* is of the form '$A \leftarrow B_1, \ldots, B_k$'.

A *horn clause program* consist of a finite set of *facts* and *rules*. A query '$\leftarrow B_1, \ldots, B_k$' is said to be provable from a program, if each of $B_i, (1 \leq i \leq k)$ can be proved. A goal 'B' is proved, if either

- it *unifies* (matches) with a fact 'A' in the program, or
- B unifies with a head 'A' of a rule '$A \leftarrow B'_1, \ldots, B'_l$' in the program, and the query '$B'_1, \ldots, B'_l$' is in turn provable.

Intuitively two terms *unify* if they can represent the same term consistently, details are in [Gal87a, Llo84]. For example: $f(a)$ unifies with $f(X)$, but not with $g(a)$. While $f(X, g(X), a, b)$ unifies with $f(a, g(Y), Y, X)$, but not with $f(Z, Z, Z, b)$.

2.10 Temporal Logic Programming

Temporal logic programming concerns itself with the automation of reasoning about time in an elegant and efficient fashion. Several researchers have shown how *time* can be incorporated in conventional programming languages.

Abadi and Manna extend horn clauses to *temporal horn clauses* by qualifying the atomic formulas with PTL modalities to get the temporal logic programming language *Templog*[AM87]. For example they give the following *Templog* program:

$$
\begin{aligned}
& fib(0) \leftarrow . \\
& \bigcirc fib(1) \leftarrow . \\
& \bigcirc \bigcirc fib(X) \Leftarrow fib(Y), \bigcirc fib(Z), X \, is \, Y + Z.
\end{aligned}
$$

to generate the fibonacci sequence. The first two clauses are applicable on the initial state. The third clause is applicable on all states and is known as a *permanent clause*. The query '*fib(X)*' is answered by the sequence: "$fib(0), \bigcirc fib(1), \bigcirc \bigcirc fib(2), \ldots$".

The *Templog* answering mechanism is an extension of the normal SLD-resolution [AM87, Bau89]. *Templog* allows '$\Box$' in the head of a rule, and '$\Diamond$' in the body of the rule. Allowing facts and queries to be qualified by arbitrary modalities leads to complications in program execution.

Many examples show that time is better treated as a set of intervals than as a set of points[vB83, Bur84, Gal87b]. For example, '*John drove from Bombay to Delhi*,' holds over an interval. In linear time, there are exactly 13 mutually exclusive ways in which two intervals may occur. Several systems of temporal reasoning have been built around this fact.

Moszkowski and Hale have developed the temporal logic programming language *Tempura* based on *interval temporal logic* (ITL)[Mos86, Hal87, RDMMS92].

A *Tempura* program is also an ITL formula of a special type where the current state deterministically specifies the 'next' state. The program is executed by rewriting it, and moving from the

current state to the next state. This results in creation of a *finite* sequence of states that is a model of the program.

Moszkowski gives the following example [Mos86]:

$$(M = 4) \wedge (N = 1) \wedge halt(M = 0) \wedge (M\ gets\ M - 1)$$
$$\wedge (N\ gets\ 2N) \wedge \Box display(M, N).$$

This ITL formula is also a *Tempura* program, it is true on an interval of five states. State zero is completely specified by the initial values of M and N; *'gets'* relates the values of the variables from one state to the next one; and when the argument of *halt* is true the program halts.

They also provide a powerful operator called *chop* ';' that can sequence two programs one after the other, as in the following example:

$$(M = 4) \wedge while\,(M \neq 0)\,do\,(M\ gets\ M - 1)\,;\,(M = 4)$$
$$\wedge while\,(M \neq 0)\,do\,(M\ gets\ M - 1)$$

This formula is composed of two formulas, that are executed sequentially resulting in two consecutive intervals, each of length five.

The model theory of an ITL is related to *büchi* automata and a decision procedure is given in [RDMMS92].

Chapter 3

Propositional Dense Time Logic

This chapter presents our work on a new logic for representing and reasoning with dense time. We show the logic to be decidable in exponential time and give a complete axiomatization for it.

3.1 The Language

Propositional Dense Time Logic (PDTL) is an extension of the usual PTL. The language of PDTL contains the usual propositional logic symbols: truth constants $\mathcal{T}$ (true) and $\mathcal{F}$ (false), set of propositions $PROP = \{p_i : i \geq 0\}$, logical connectives: $\neg$ (not), $\rightarrow$ (implies), the usual temporal operators: $\Box$ (henceforth), $\bigcirc$ (next), with the following new operators: $\boxplus$ (everywhere) and $\oslash$ (within). The *until* operator of PTL [GPSS80] is not considered here.

Define the language of PDTL to be a set L_{PDTL} of formulas generated by:

$$L ::= \mathcal{F} | \neg L | L \rightarrow L | \Box L | \bigcirc L | \boxplus L | \oslash L | p, \quad p \in PROP$$

The syntactic definitions are:

$$\begin{array}{lcl} \Diamond\phi & \stackrel{\text{def}}{=} & \neg\Box\neg\phi \\ \diamondsuit\phi & \stackrel{\text{def}}{=} & \neg\boxplus\neg\phi \end{array} \qquad \begin{array}{lcl} \mathcal{T} & \stackrel{\text{def}}{=} & \neg\mathcal{F} \\ \phi \vee \psi & \stackrel{\text{def}}{=} & \neg\phi \rightarrow \psi \\ \phi \wedge \psi & \stackrel{\text{def}}{=} & \neg(\phi \rightarrow \neg\psi) \\ \phi \leftrightarrow \psi & \stackrel{\text{def}}{=} & (\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi) \end{array}$$

Notation:

1. $\mathcal{N} \stackrel{\text{def}}{=} \{0, 1, 2, \ldots\}$, the set of natural numbers.
2. $\mathcal{N}^+ \stackrel{\text{def}}{=} \{(k_0, \ldots, k_n) : k_i \in \mathcal{N}, 0 \leq i \leq n\}$

3. $\mathcal{N}^* \stackrel{\text{def}}{=} \mathcal{N}^+ \cup \{()\}$

4. An element of $\mathcal{N}^*$ will be denoted by $\overline{k} = (k_0, \ldots, k_n)$,

5. $\overline{0}_i \stackrel{\text{def}}{=} (\underbrace{0, \ldots, 0}_{i \text{ times}})$, $i \geq 0$, an i-tuple of zeroes.

6. The concatenation of $\overline{k}$ with i will be denoted by $\overline{k} \cdot i = (k_0, \ldots, k_n, i)$,

7. The concatenation of $\overline{k}$ with $\overline{m} = (m_0, \ldots, m_j)$, will be $\overline{k} \cdot \overline{m} = (k_0, \ldots, k_n, m_0, \ldots, m_j)$.

8. $PROP(\phi)$ will denote the set of propositions occurring in a formula ϕ.

3.2 Semantics

While the models of PTL are based on the natural numbers, the models of PDTL are based on nested sequences of natural numbers. The language and the semantics of PTL are extended so that we can talk about the truth of formulas in such a model.

A state is a subset of $PROP$. An ω sequence of states (the usual model for PTL [GPSS80, LPZ85]) is a sequence of states indexed by elements of $\mathcal{N}$. The model for PDTL is an infinitely nested sequence of states.

Definition 2 (Omega Tree) *An* omega tree *model for PDTL is an infinite binary tree* $T = (\mathcal{N}^+, 0, w, x, s)$*, rooted at* (0)*, with two successor functions:* $w : \mathcal{N}^+ \mapsto \mathcal{N}^+$ *(*within*) and* $x : \mathcal{N}^+ \mapsto \mathcal{N}^+$ *(*next*). The valuation function* $s : \mathcal{N}^+ \mapsto 2^{PROP}$*, maps its nodes* $(\mathcal{N}^+)$ *to subsets of* $PROP$*.*

The w-child of $(\overline{k})$ is $(\overline{k} \cdot 0)$ and the x-child of $(\overline{k} \cdot i)$ is $(\overline{k} \cdot i + 1)$, see figure 3.1. The omega tree rooted at $(k_0, \ldots, k_n)$ will be denoted by $T((k_0, \ldots, k_n))$.

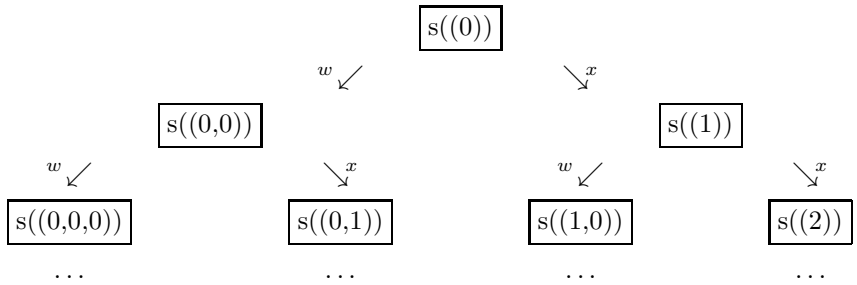


Figure 3.1: Omega tree

The sequence of states $\ll s((0)), s((1)), \ldots \gg$ occurs as in a PTL model, but here the sequence of states $\ll s((0,0)), s((0,1)), \ldots \gg$ is a nested sequence between the states $s((0))$ and $s((1))$. This

sequence is accessible from $s((0))$ by the $\oslash$ operator. The states can be temporally ordered as follows:

$$\begin{aligned} &s(\overline{k}) =_t s(\overline{k} \cdot 0) && \overline{k} \in \mathcal{N}^+ \\ &s(\overline{k} \cdot i) \leq_t s(\overline{k} \cdot i \cdot \overline{m}) <_t s(\overline{k} \cdot i + 1) && \overline{k} \in \mathcal{N}^*,\ \overline{m} \in \mathcal{N}^+,\ i \geq 0 \end{aligned}$$

Since we identify the time point $s(\overline{k})$ with $s(\overline{k} \cdot 0)$,

$$p \in s(\overline{k}) \leftrightarrow p \in s(\overline{k} \cdot 0), \quad p \in PROP$$

Truth of formulas in T is defined as follows:

$$\begin{aligned} T \models \phi \quad &\stackrel{\text{def}}{=} \quad T((0)) \models \phi \\ (T(\overline{k}) \models p),\ p \in PROP \quad &\text{iff} \quad p \in s(\overline{k}) \\ T(\overline{k}) &\not\models \mathcal{F} \\ T(\overline{k}) \models \neg\phi \quad &\text{iff} \quad \text{not } (T(\overline{k}) \models \phi) \\ T(\overline{k}) \models \phi \rightarrow \psi \quad &\text{iff} \quad \text{not } (T(\overline{k}) \models \phi) \text{ or } (T(\overline{k}) \models \psi) \\ T((k_1, \ldots, k_n)) \models \bigcirc\phi \quad &\text{iff} \quad T((k_1, \ldots, k_{(n-1)}, k_n + 1)) \models \phi \\ T((k_1, \ldots, k_n)) \models \Box\phi \quad &\text{iff} \quad \forall j \geq 0,\ T((k_1, \ldots, k_{(n-1)}, k_n + j)) \models \phi \\ T((k_1, \ldots, k_n)) \models \Diamond\phi \quad &\text{iff} \quad \exists j \geq 0,\ T((k_1, \ldots, k_{(n-1)}, k_n + j)) \models \phi \end{aligned}$$

$\oslash\phi$ asserts that ϕ holds in the w-child of the current state, and $\boxplus\phi$ asserts that ϕ holds everywhere below the current state in the omega tree model.

$$\begin{aligned} T(\overline{k}) \models \oslash\phi \quad &\text{iff} \quad T(\overline{k} \cdot 0) \models \phi \\ T((k_1, \ldots, k_n)) \models \boxplus\phi \quad &\text{iff} \quad \forall j \geq 0,\ \forall \overline{m} \in \mathcal{N}^*,\ T((k_1, \ldots, k_{(n-1)}, k_n + j) \cdot \overline{m}) \models \phi \\ T((k_1, \ldots, k_n)) \models \diamondsuit\phi \quad &\text{iff} \quad \exists j \geq 0,\ \exists \overline{m} \in \mathcal{N}^*,\ T((k_1, \ldots, k_{(n-1)}, k_n + j) \cdot \overline{m}) \models \phi \end{aligned}$$

3.3 Examples

In the examples below, we assume that time is divided into days at the top level so that the intervals $[s_0, s_1), [s_1, s_2), \ldots$ represent days.

Example 4 It is always hot: $\boxplus Hot$.

Example 5 There will be rain: $\diamondsuit Rain$.

Example 6 If we want to assert that the rain will last over a time interval we assert: $\diamondsuit \oslash \boxplus Rain$.

Example 7 It rains daily: $\Box \oslash \diamondsuit Rain$. This does not mean that it rains the whole day, but that on everyday there is rain. In fact it is consistent with the next example.

Example 8 It never rains for a full day: $\Box \oslash \diamondsuit \neg Rain$.

Example 9 Everyday the rains are followed by a flood:

$$\Box \oslash \Diamond\!\!\!\!\Diamond (\oslash \Diamond\!\!\!\!\Diamond \boxplus Rain \wedge \bigcirc \Diamond\!\!\!\!\Diamond \boxplus Flood)$$

Example 10 If he takes a walk everyday, then someday he will get wet in the rain:

$$\Box \oslash \Diamond\!\!\!\!\Diamond Walk \rightarrow \Diamond \oslash \Diamond\!\!\!\!\Diamond (Walk \wedge Rain)$$

Note that this is not a theorem. Consider in contrast the next example.

Example 11 Even though it rains daily, it is possible for him to take a walk everyday and not get wet in the rain:

$$\Box \oslash (\Diamond\!\!\!\!\Diamond Walk \wedge \Diamond\!\!\!\!\Diamond Rain \wedge \boxplus (Walk \rightarrow \neg Rain))$$

PDTL offers us an interesting way of modeling qualitative temporal information 'event p occurred *many* times' in terms of '$\oslash\Box p$', which deliberately identifies 'many occurrences' with 'infinitely many occurrences in finite time' [AV93a]. We could also use this logic to reason more clearly about program executions. To represent that a property ϕ holds after one statement is executed we write $\bigcirc\phi$. If the statement is an abstraction of further statements (e.g. a procedure call), then the properties during the call can be written as $\oslash \bigcirc^i \psi$.

3.4 Satisfiability and Tableaux

Given a formula ϕ, we give a tableau method [LPZ85, Wol83] of deciding whether it is satisfiable or not. A tableau is a rooted directed graph, containing two kinds of nodes:

1. A *state node* contains only *literals* (propositions and their negations) and formulas of types: $\bigcirc\phi$ or $\oslash\phi$. A state-node η has two subnodes: the *within subnode* $[\eta]_w$ and the *next subnode* $[\eta]_x$. An empty node is considered to be a state node.

2. All other nodes are *logical nodes*. A logical-node η has two subnodes $[\eta]_l$ and $[\eta]_r$.

3.4.1 Tableaux Building

We build a tableau for ϕ starting from the root node $\{\phi\}$. A leaf-node is expanded according to the type of formulas in it. On expansion, the leaf nodes generate sub-nodes to which it is linked by outgoing arcs. If a new subnode has the same formulas as an existing node μ, then the new sub-node is not created, instead η is linked to μ.

In a logical-leaf-node η, pick any expandable formula and call it the *principal formula* of η (denoted by $Pr(\eta)$). This formula will be expanded to generate subnodes according to table 3.1. Other formulas in η, called the *side formulas* are then copied unchanged to the subnodes. A state-

	$Pr(\eta) \in \eta$	$[\eta]_l,\ ([\eta]_r = \mathcal{F})$
1	$\Box A$	$A, \bigcirc\Box A$
2	$\Diamond A$	$A \vee \bigcirc\Diamond A$
3	$\boxplus A$	$A, \bigcirc\boxplus A, \oslash\boxplus A$
4	$\Diamond\!\!\!\!-\!\!\!\!\Diamond A$	$A \vee (\bigcirc\Diamond\!\!\!\!-\!\!\!\!\Diamond A \vee \oslash\Diamond\!\!\!\!-\!\!\!\!\Diamond A)$
5	$\neg \bigcirc A$	$\bigcirc\neg A$
6	$\neg \oslash A$	$\oslash\neg A$
7	$\neg\Box A$	$\Diamond\neg A$
8	$\neg\Diamond A$	$\Box\neg A$
9	$\neg\boxplus A$	$\Diamond\!\!\!\!-\!\!\!\!\Diamond\neg A$
10	$\neg\Diamond\!\!\!\!-\!\!\!\!\Diamond A$	$\boxplus\neg A$

	$Pr(\eta) \in \eta$	$[\eta]_l$	$[\eta]_r$
11	$A \to B$	$\neg A$	B
12	$A \wedge B$	A, B	$\mathcal{F}$
13	$A \vee B$	A	B
14	$A \leftrightarrow B$	$\neg A, \neg B$	A, B
15	$\neg(A \to B)$	$A, \neg B$	$\mathcal{F}$
16	$\neg(A \wedge B)$	$\neg A$	$\neg B$
17	$\neg(A \vee B)$	$\neg A, \neg B$	$\mathcal{F}$
18	$\neg(A \leftrightarrow B)$	$\neg A, B$	$A, \neg B$
19	$\neg\neg A$	A	$\mathcal{F}$

Table 3.1: Expansion table for logical nodes

leaf-node generates two subnodes: $[\eta]_w = \{\phi | \oslash \phi \in \eta\} \cup \{literal \in \eta\}$ and $[\eta]_x = \{\phi | \bigcirc \phi \in \eta\}$.

Let S be the set of formulas and E be the set of arcs that appear in the tableau of $\{\phi\}$. We have $|E| \leq (2 \cdot |S|)$ and $|S| < (4 \cdot |\phi|)$, so number of nodes $< 2^{|S|} < 2^{4|\phi|}$ and the tableau expansion eventually halts.

3.4.2 Closing Nodes

A node is *closed* when we can show it is unsatisfiable, that is we can prove the negation of the node. A node which is not closed is called *open*. An empty node is considered open. If a node contains both A and $\neg A$ or contains $\mathcal{F}$, then it is obviously unsatisfiable and therefore closed. However it is more difficult to give conditions for closing a node containing an unsatisfiable formula of the form $\Diamond A$ or $\Diamond\!\!\!\!-\!\!\!\!\Diamond A$.

If $\Diamond A$ is unsatisfiable at a node, it will be carried over only to every x-descendant of that node, that is, it will be indefinitely postponed. However if $\Diamond\!\!\!\!-\!\!\!\!\Diamond A$ is unsatisfiable at a node, it will be carried over separately to both its x and w descendants. Thus, we should close a node containing $\Diamond A$, if $\neg A$ holds at every node reachable from it by logical and x-arcs. Similarly we close a node containing $\Diamond\!\!\!\!-\!\!\!\!\Diamond A$, if $\neg A$ holds at every reachable node.

Define RO, ROx and $ROwx$ to be the set of *open* descendants of an open node η as follows:

$RO(\eta)$	$=$	$\{\eta\} \cup RO([\eta]_l) \cup RO([\eta]_r)$	if η is a logical node
$RO(\eta)$	$=$	$\{\eta\}$	otherwise
$ROx(\eta)$	$=$	$\{[\eta]_l, [\eta]_r\} \cup ROx([\eta]_l) \cup ROx([\eta]_r)$	if η is a logical node
$ROx(\eta)$	$=$	$\{[\eta]_x\} \cup ROx([\eta]_x)$	if η is a state node
$ROwx(\eta)$	$=$	$\{[\eta]_l, [\eta]_r\} \cup ROwx([\eta]_l) \cup ROwx([\eta]_r)$	if η is a logical node
$ROwx(\eta)$	$=$	$\{[\eta]_x, [\eta]_w\} \cup ROwx([\eta]_w) \cup ROwx([\eta]_x)$	if η is a state node
x-leaf-scc(η)	iff	$\forall\mu(\mu \in ROx(\eta) \rightarrow \eta \in ROx(\mu))$	
wx-leaf-scc(η)	iff	$\forall\mu(\mu \in ROwx(\eta) \rightarrow \eta \in ROwx(\mu))$	

$RO(\eta)$ is the set of logical descendants of η, that is, nodes reachable from η through logical arcs. $ROx(\eta)$ is the set of nodes reachable from η by a path of *logical* and *next* arcs; this set is a maximal x-strongly connected component when x-leaf-scc(η) is true. Similarly $ROwx(\eta)$ is the set of nodes reachable from η by any path, and this set is a wx-strongly connected component when wx-leaf-scc(η) is true.

Once the tableau is built, we apply the *closing algorithm* given in table 3.2.

Repeat

1. Construct $ROx(\eta)$ and $ROwx(\eta)$ for each open node η.
2. Close any node η that satisfies any one of the conditions:
   - 2.1. $\psi, \neg\psi \in \eta$ or $\mathcal{F} \in \eta$
   - 2.2. All the logical children of η are closed
   - 2.3. Either $[\eta]_x$ or $[\eta]_w$ is closed
   - 2.4. $\Diamond\psi \in \eta$, $\psi \notin \bigcup ROx(\eta) \cup \eta$ and η is an immediate child of a state node
   - 2.5. $\Diamond\!\!\!\!-\psi \in \eta$, $\psi \notin \bigcup ROwx(\eta) \cup \eta$ and η is an immediate child of a state node

Until no more nodes can be closed.

Table 3.2: Closing Algorithm

3.4.3 State Node Tableau

We include $\boxplus\mathcal{T}$ in the root of τ to ensure that the tableau τ of ϕ expands completely. $\boxplus\mathcal{T}$ is satisfiable in every omega tree and hence does not affect the satisfiability of ϕ.

For the purpose of building models we need only the open state nodes. The tableau τ is shortened to a *state node tableau* Δ by removing the logical nodes as follows:

- The nodes of Δ are the open state nodes of τ.
- For $\mu_1, \mu_2 \in \Delta$, if $\mu_2 \in RO([\mu_1]_x)$ in τ, then an x arc is drawn from μ_1 to μ_2.
- For $\mu_1, \mu_2 \in \Delta$, if $\mu_2 \in RO([\mu_1]_w)$ then a w arc is drawn from μ_1 to μ_2.
- Note that a node in Δ may have many x and w children. Hence $[\mu]_x$ and $[\mu]_w$ will denote sets when $\mu \in \Delta$.

Theorem 1 *$\{\phi\}$ is open iff it is satisfiable.*

Proof:
($\Leftarrow$) Let $\{\phi\}$ be closed, we show that it is unsatisfiable. Suppose $\{\phi\}$ is closed, then by theorem 4 $\vdash_{Ax_{PDTL}} \neg\phi$. By soundness of the Ax_{PDTL} (proposition 3) we get $\models \neg\phi$. Therefore ϕ is unsatisfiable. ∎

($\Rightarrow$) An omega tree model for the formulas in a node η can be extracted from the set of open state nodes in Δ. The mapping $m : \mathcal{N}^+ \to \Delta$ is defined in table 3.3. Here the set $R(\{\phi\})$ is the set of open state nodes reachable from the root of τ after the closing algorithm has been applied. $Rx(\eta)$ is the set of open state nodes reachable from η in Δ by traversing one or more x-arcs. $Rwx(\eta)$ is the set of open state nodes reachable from η in Δ by traversing one or more x or w arcs.

The states of the omega tree can now be defined:

$$s(\overline{k}) \stackrel{\text{def}}{=} \bigcup_{i \geq 0} (m(\overline{k} \cdot \overline{0}_i) \cap PROP), \quad \overline{k} \in \mathcal{N}^+$$

$(T(\overline{k}) \models \psi) \Leftrightarrow (\psi \in m(\overline{k}))$ is shown by induction on the structure of ψ, from this we infer $T \models \phi$. The important part is to verify that if $T(\overline{k} \cdot k') \models \Diamond\psi$ (respectively $T(\overline{k} \cdot k') \models \Diamond\!\!\!-\psi$) then $T(\overline{k} \cdot k' + j) \models \psi$ for some $j \in \mathcal{N}$ (respectively $T(\overline{k} \cdot k'' \cdot \overline{j}) \models \psi$ for some $k'' \geq k'$, $\overline{j} \in \mathcal{N}^*$). This can be shown by using the fact $T(\overline{k} \cdot k')$ was constructed using every node of the x-leaf-scc (wx-leaf-scc respectively) [LPZ85]. ∎

Corollary 2 *PDTL is decidable in exponential time.*

Proof: The required strongly connected components can be constructed in time $O(|E| + |\tau|)$, where $|E|$ is the number of edges in τ. Therefore the whole tableau procedure takes $O(2^{c|\phi|})$ for some constant c. ∎

Let $m((0)) \in R(\{\phi\})$
If $m(\overline{k} \cdot k') = \mu$ then
  If x-leaf-scc(μ) then
    $\exists \nu_0, \ldots, \nu_k : \{\nu_0, \ldots, \nu_k\} = Rx(\mu)$
    $\wedge(\mu = \nu_0)\ \wedge(\mu \in [\nu_k]_x)$
    $\wedge(\nu_{j+1} \in [\nu_j]_x), 0 \leq j < k$
    For $j := 0$ to k do
      Let $m(\overline{k} \cdot k' + j) := \nu_j$
      If $\exists \nu'_j \in ([\nu_j]_w - Rx(\mu))$ then
        Let $m(\overline{k} \cdot k' + j \cdot 0) := \nu'_j$
      Else
        Let $m(\overline{k} \cdot k' + j \cdot 0) \in [\nu_j]_w$
      EndIf
  ElseIf wx-leaf-scc(μ) then
    $\exists \nu_0, \ldots, \nu_k : \{\nu_0, \ldots, \nu_k\} = Rwx(\mu)$
    $\wedge(\mu = \nu_0)\ \wedge(\mu \in ([\nu_k]_x \cup [\nu_k]_w))$
    $\wedge(\nu_{j+1} \in ([\nu_j]_x \cup [\nu_j]_w)),\ 0 \leq j < k$
    Let $\overline{l} \cdot l' := \overline{k} \cdot k'$
    For $j := 0$ to k do
      If $\nu_{j+1} \in [\nu_j]_x$ then
        Let $m(\overline{l} \cdot l' + 1) := \nu_{j+1}$
        Let $m(\overline{l} \cdot l' \cdot 0) \in [\nu_j]_w$
        Let $\overline{l} \cdot l' := \overline{l} \cdot (l' + 1)$
      Else
        Let $m(\overline{l} \cdot l' \cdot 0) := \nu_{j+1}$
        Let $m(\overline{l} \cdot l' + 1) \in [\nu_j]_x$
        Let $\overline{l} \cdot l' := \overline{l} \cdot l' \cdot 0$
      EndIf
  Else
    Let $m(\overline{k} \cdot k' + 1) \in [\mu]_x$
    Let $m(\overline{k} \cdot k' \cdot 0) \in [\mu]_w$
  EndIf
EndIf

Table 3.3: Mapping $m : \mathcal{N}^+ \mapsto \Delta$.

Example 12 The tableau for the unsatisfiable formula $\boxplus p \wedge \Diamond\!\!\!\!-\neg p$ is given in figure 3.2. We close η_a because $\Diamond\!\!\!\!-\neg p \in \eta_a$ and $\neg p \notin \bigcup ROwx(\eta_a) \cup \eta_a$.

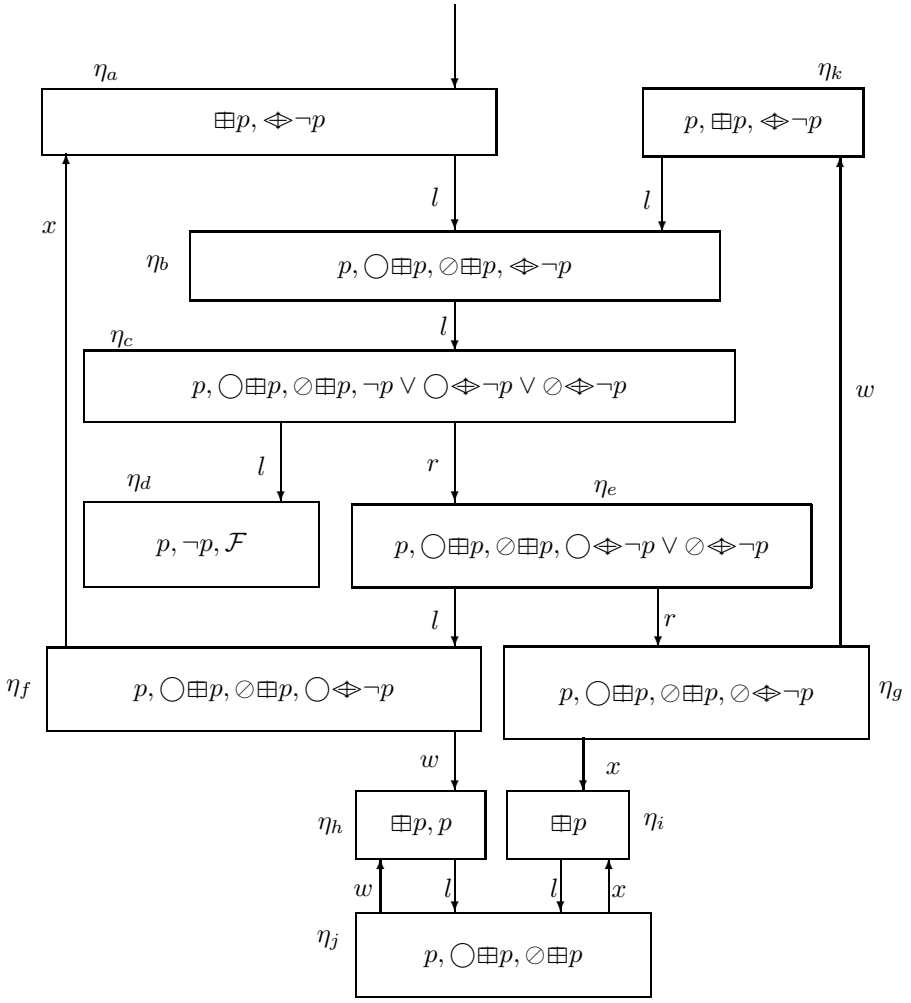


Figure 3.2: Tableau for $\{\boxplus p, \Diamond\!\!\!\!-\neg p\}$

Example 13 The tableau for the satisfiable formula $\Box p \wedge \Diamond\!\!\!\!-\neg p$ is given in figure 3.3.

3.5 Axiomatization of PDTL

We now give axioms and rules of inference for PDTL.

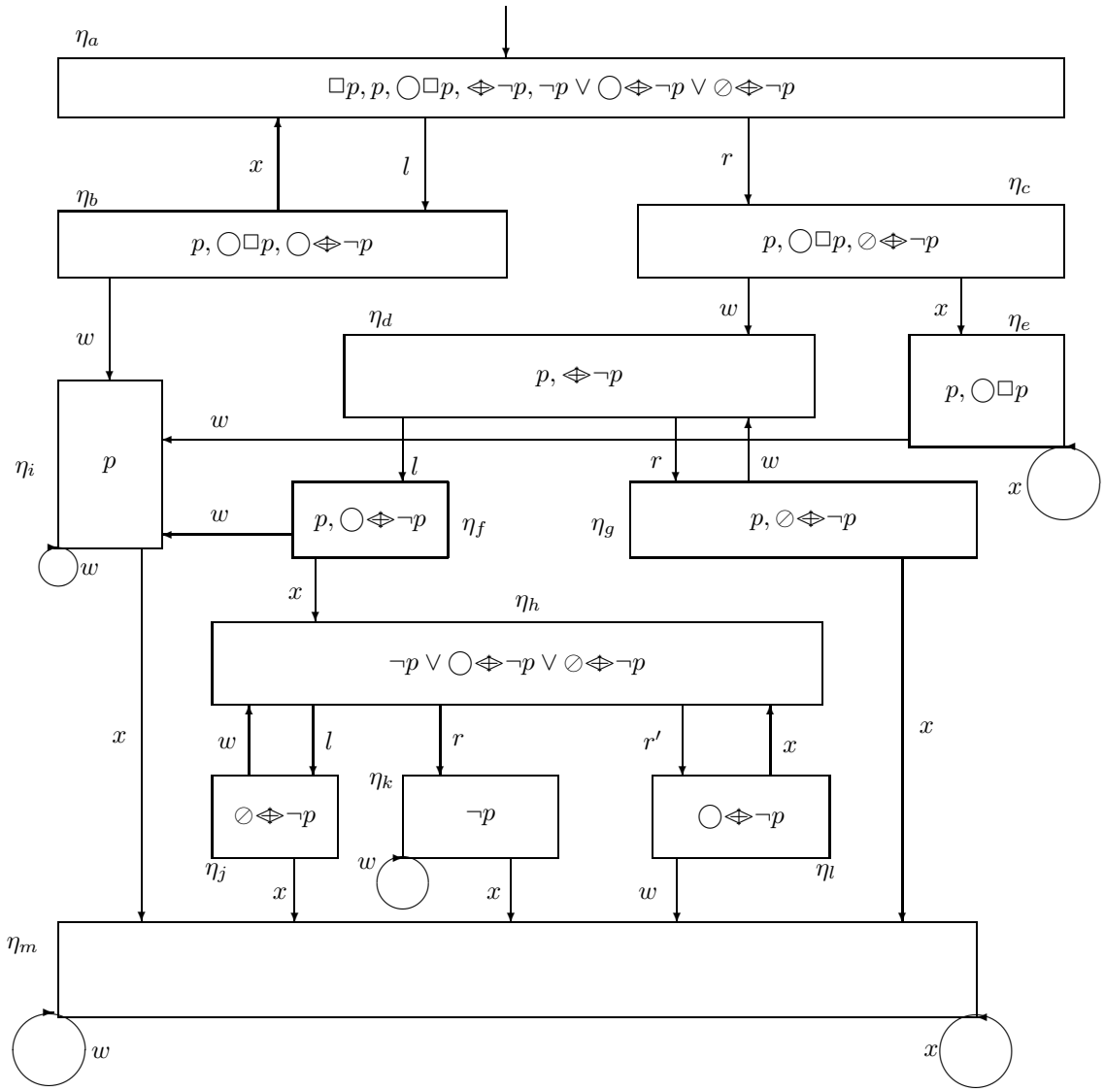


Figure 3.3: Tableau for $\{\Box p, \Diamond\!\!\!\!-\neg p\}$

The Axioms (Ax_{PDTL}):

$$
\begin{array}{ll}
Ax1 & \Box(A \to B) \to (\Box A \to \Box B) \\
Ax2 & \boxplus(A \to B) \to (\boxplus A \to \boxplus B) \\
Ax3 & \bigcirc(A \to B) \to (\bigcirc A \to \bigcirc B) \\
Ax4 & \oslash(A \to B) \to (\oslash A \to \oslash B) \\
Ax5 & \bigcirc\neg A \leftrightarrow \neg \bigcirc A \\
Ax6 & \oslash\neg A \leftrightarrow \neg \oslash A \\
Ax7 & \Box A \to (A \wedge \bigcirc A \wedge \bigcirc\Box A) \\
Ax8 & \boxplus A \to (A \wedge \oslash A \wedge \bigcirc A \wedge \oslash\boxplus A \wedge \bigcirc\boxplus A) \\
Ax9 & (A \wedge \Box(A \to \bigcirc A)) \to \Box A \\
Ax10 & (A \wedge \boxplus(A \to (\bigcirc A \wedge \oslash A))) \to \boxplus A \\
Ax11 & p \leftrightarrow \oslash p, \quad \forall p \in PROP
\end{array}
$$

Rules of Inference:

$$
PL.\frac{\vdash_{PL} A}{\vdash A}, \quad MP.\frac{\vdash A, A \to B}{\vdash B}, \quad RN.\frac{\vdash A}{\vdash \Box A}, \quad RE.\frac{\vdash A}{\vdash \boxplus A}
$$

The rule PL allows all Propositional Logic tautologies to be theorems of PDTL.

Proposition 3 (Soundness) *The axioms are sound, and the rules of inference preserve soundness.*

3.5.1 Derived Rules of Inference and Theorems of PDTL

$$
\begin{array}{ll}
R1. & \dfrac{\vdash A}{\vdash \bigcirc A \wedge \oslash A} \\
R2. & \dfrac{\vdash A \to B}{\vdash \bigcirc A \to \bigcirc B} \\
R3. & \dfrac{\vdash A \to B}{\vdash \oslash A \to \oslash B} \\
R4. & \dfrac{\vdash A \to B}{\vdash \Box A \to \Box B} \\
R5. & \dfrac{\vdash A \to B}{\vdash \boxplus A \to \boxplus B}
\end{array}
$$

$T1. \vdash \Box(A \wedge B) \leftrightarrow (\Box A \wedge \Box B)$

$T2. \vdash \bigcirc(A \wedge B) \leftrightarrow (\bigcirc A \wedge \bigcirc B)$

$T3. \vdash \boxplus(A \wedge B) \leftrightarrow (\boxplus A \wedge \boxplus B)$

$T4. \vdash \oslash(A \wedge B) \leftrightarrow (\oslash A \wedge \oslash B)$

$T5. \vdash (A \wedge \bigcirc\Box A) \rightarrow \Box A$

$T6. \vdash (A \wedge \bigcirc\boxplus A \wedge \oslash\boxplus A) \rightarrow \boxplus A$

$T7. \vdash \bigcirc(A \vee B) \leftrightarrow (\bigcirc A \vee \bigcirc B)$

$T8. \vdash \oslash(A \vee B) \leftrightarrow (\oslash A \vee \oslash B)$

$T9. \vdash \boxplus A \rightarrow \Box A$

Proof of $R1$.

$\vdash A$	1	$Hyp.$
$\vdash \boxplus A$	2	$RE, 1$
$\vdash \boxplus A \rightarrow \bigcirc A \wedge \oslash A$	3	$Ax8$
$\vdash \bigcirc A \wedge \oslash A$	4	$PL, 2, 3$ ∎

Proof of $R2$.

$\vdash A \rightarrow B$	1	$Hyp.$
$\vdash \bigcirc(A \rightarrow B)$	2	$R1, 1$
$\vdash \bigcirc(A \rightarrow B) \rightarrow (\bigcirc A \rightarrow \bigcirc B)$	3	$Ax3$
$\vdash \bigcirc A \rightarrow \bigcirc B$	4	$PL, 2, 3$ ∎

Proofs of $R3$,$R4$ and $R5$ are similar to that of $R2$.

Proof of $T4$.

$\vdash (A \wedge B) \rightarrow A$	1	PL
$\vdash \oslash(A \wedge B) \rightarrow \oslash A$	2	$R3, 1$
$\vdash (A \wedge B) \rightarrow B$	3	PL
$\vdash \oslash(A \wedge B) \rightarrow \oslash B$	4	$R3, 3$
$\vdash \oslash(A \wedge B) \rightarrow (\oslash A \wedge \oslash B)$	5	$PL, 2, 4$
$\vdash A \rightarrow (B \rightarrow (A \wedge B))$	6	PL
$\vdash \oslash A \rightarrow \oslash(B \rightarrow (A \wedge B))$	7	$R3, 6$
$\vdash \oslash(B \rightarrow (A \wedge B)) \rightarrow (\oslash B \rightarrow \oslash(A \wedge B))$	8	$Ax4$
$\vdash \oslash A \rightarrow (\oslash B \rightarrow \oslash(A \wedge B))$	9	$PL, 7, 8$
$\vdash (\oslash A \wedge \oslash B) \rightarrow \oslash(A \wedge B)$	10	$PL, 9$
$\vdash (\oslash A \wedge \oslash B) \leftrightarrow \oslash(A \wedge B)$	11	$PL, 5, 10$ ∎

the proofs of $T1$,$T2$, and $T3$ are similar.

Proof of $T5$.

Let: $B \stackrel{\text{def}}{=} (A \wedge \bigcirc\Box A)$

$\vdash \Box A \rightarrow B$	1	$Ax7$
$\vdash \bigcirc\Box A \rightarrow \bigcirc B$	2	$R2, 1$
$\vdash B \rightarrow \bigcirc B$	3	$PL, 2$
$\vdash \Box(B \rightarrow \bigcirc B)$	4	$RN, 3$
$\vdash (B \wedge \Box(B \rightarrow \bigcirc B)) \rightarrow \Box B$	5	$Ax9$
$\vdash B \rightarrow \Box B$	6	$PL, 4, 5$
$\vdash \Box B \rightarrow \Box A$	7	$T1, PL$
$\vdash B \rightarrow \Box A$	8	$PL, 6, 7$
$\vdash (A \wedge \bigcirc\Box A) \rightarrow \Box A$	9	8 ∎

Proof of $T6$.

Let: $B \stackrel{\text{def}}{=} (A \wedge \bigcirc\boxplus A \wedge \oslash\boxplus A)$

$\vdash \boxplus A \rightarrow B$	1	$Ax8$
$\vdash \bigcirc\boxplus A \rightarrow \bigcirc B$	2	$R2, 1$
$\vdash \oslash\boxplus A \rightarrow \oslash B$	3	$R3, 1$
$\vdash B \rightarrow (\bigcirc B \wedge \oslash B)$	4	$PL, 2, 3$
$\vdash \boxplus(B \rightarrow (\bigcirc B \wedge \oslash B))$	5	$RE, 4$
$\vdash (B \wedge \boxplus(B \rightarrow (\bigcirc B \wedge \oslash B))) \rightarrow \boxplus B$	6	$Ax10$
$\vdash B \rightarrow \boxplus B$	7	$PL, 5, 6$
$\vdash \boxplus B \rightarrow \boxplus A$	8	$T3, PL$
$\vdash B \rightarrow \boxplus A$	9	$PL, 7, 8$
$\vdash (A \wedge \bigcirc\boxplus A \wedge \oslash\boxplus A) \rightarrow \boxplus A$	10	9 ∎

Proof of $T7$.

$\vdash \neg \bigcirc (\neg A \wedge \neg B) \leftrightarrow \neg(\bigcirc\neg A \wedge \bigcirc\neg B)$	1	$T2$
$\vdash \bigcirc\neg(\neg A \wedge \neg B) \leftrightarrow (\neg \bigcirc \neg A \vee \neg \bigcirc \neg B)$	2	$Ax5, PL, 1$
$\vdash \bigcirc(A \vee B) \leftrightarrow (\bigcirc A \vee \bigcirc B)$	3	$Ax5, PL, 2$ ∎

Proof of $T8$ is similar.

Proof of $T9$.

$\vdash (\boxplus A \wedge A) \rightarrow (\bigcirc \boxplus A \wedge \bigcirc A)$	1	$Ax8$
$\vdash (\boxplus A \wedge A) \rightarrow \bigcirc(\boxplus A \wedge A)$	2	$T2, 1$
$\vdash \Box((\boxplus A \wedge A) \rightarrow \bigcirc(\boxplus A \wedge A))$	3	$RN, 2$
$\vdash (\boxplus A \wedge A) \rightarrow \Box(\boxplus A \wedge A)$	4	$Ax9, 3$
$\vdash \Box(\boxplus A \wedge A) \rightarrow (\Box \boxplus A \wedge \Box A)$	5	$T1$
$\vdash (\boxplus A \wedge A) \rightarrow \Box A$	6	$PL, 4, 5$
$\vdash \boxplus A \rightarrow A$	7	$Ax8$
$\vdash \boxplus A \rightarrow \Box A$	8	$PL, 6, 7$ ∎

3.6 Completeness

Theorem 4 *If $\{\phi_1, \ldots, \phi_m\}$ is closed, then $\neg(\phi_1 \wedge \cdots \wedge \phi_m)$ is provable from the axioms.*

Proof:

We prove this in the order in which the nodes are closed, so that at each step the theorem holds for nodes closed at an earlier stage. We use N to denote the conjunction of formulas in a node η. Consider the ways in which a node η is closed:

1. Node η contains both ϕ and $\neg\phi$ (or $\mathcal{F}$), then it is unsatisfiable, and $\neg N$ is provable by PL.

2. Logical node η is closed, because all its children are closed. Let L and R denote the conjunction of formulas in $[\eta]_l$ and $[\eta]_r$ respectively.

$\vdash \neg L$	1	$Hyp.$
$\vdash \neg R$	2	$Hyp.$
$\vdash N \rightarrow (L \vee R)$	3	$PDTL$
$\vdash \neg N$	4	$PL, 1, 2, 3$

   Line 3 follows from the expansion table for a logical node. Rules 1-10 in the expansion table follow from the axioms and definitions for PDTL, and rules 11-19 follow from PL.

3. A state node η is closed because $[\eta]_x$ was closed, let X be the conjunction of formulas in $[\eta]_x$:

$\vdash \neg X$	1	$Hyp.$
$\vdash \bigcirc \neg X$	2	$R2, 1$
$\vdash \neg \bigcirc X$	3	$Ax5, 2$
$\vdash N \rightarrow \bigcirc X$	4	$PDTL$
$\vdash \neg N$	5	$PL, 3, 4$

4. State node η is closed because $[\eta]_w$ was closed. Let L be the conjunction of literals in η, and W be the conjunction of other formulas in $[\eta]_w$.

$$\begin{array}{lll}
\vdash \neg(L \wedge W) & 1 & Hyp. \\
\vdash \oslash\neg(L \wedge W) & 2 & R2, 1 \\
\vdash \neg \oslash (L \wedge W) & 3 & Ax6 \\
\vdash N \rightarrow (L \wedge \oslash W) & 4 & PDTL \\
\vdash N \rightarrow (\oslash L \wedge \oslash W) & 5 & Ax11, 4 \\
\vdash N \rightarrow \oslash(L \wedge W) & 6 & T4, 5 \\
\vdash \neg N & 7 & PL, 6, 3
\end{array}$$

5. A node η was closed because $\Diamond A \in \eta$ was indefinitely postponed.

   - Let $ROx(\eta)$ be the set constructed by the closing algorithm, just before η was closed.
   - Let $ROx'(\eta)$ be the set of all the immediate x-children of state nodes in $ROx(\eta)$. Since $A \notin \cup ROx(\eta)$, $\Diamond A$ will appear in every node in $ROx'(\eta)$.
   - Define $P = \bigvee_{\eta^1 \in ROx'(\eta)} \wedge(\eta^1 - \{\Diamond A\})$.

   (a) We first show $P \rightarrow \neg A$. Let the expansion of a state node $\eta^1 \in ROx'(\eta)$ proceed as follows:

$$\begin{array}{lll}
\eta^1 & \{\Diamond A, N^1\} & \in ROx'(\eta) \\
 & \Downarrow * & \text{logical arc(s)} \\
\eta_i^2 & \{\Diamond A, N_i^2\} & 1 \leq i \leq m \\
 & & Pr(\eta_i^2) = \Diamond A \\
 & \Downarrow & \text{logical arc} \\
\eta_i^3 & \{A \vee \bigcirc\Diamond A, N_i^2\} & \vdash N^1 \leftrightarrow \bigvee_{i=1}^{m} N_i^2 \\
 & \Downarrow * & \text{logical arc(s)} \\
\eta_{ij}^4 & \{A \vee \bigcirc\Diamond A, N_{ij}^4\} & 1 \leq j \leq m_i', 1 \leq i \leq m \\
 & & Pr(\eta_{ij}^4) = (A \vee \bigcirc\Diamond A) \\
 & \Downarrow & \text{logical arc} \\
\eta_{ij}^5 & \{A, N_{ij}^4\} & \vdash N_i^2 \leftrightarrow \bigvee_{j=1}^{m_i'} N_{ij}^4 \\
 & \text{closed} & A \notin \cup ROx(\eta)
\end{array}$$

   Hence

$$\begin{array}{l}
\vdash N^1 \rightarrow \neg A \\
\vdash (\vee_{\eta^1 \in ROx'(\eta)} N^1) \rightarrow \neg A \\
\vdash P \rightarrow \neg A
\end{array}$$

(b) We now show $P \rightarrow \bigcirc P$. Let the expansion of a state node $\eta^1 \in ROx'(\eta)$ proceed as follows:

$$\begin{array}{lll} \eta^1 & \{\Diamond A, N^1\} & \in ROx'(\eta) \\ & \Downarrow * & \text{logical arc(s)} \\ \eta_i^2 & \{\Diamond A, N_i^2\} & \text{state node} \\ & & 1 \leq i \leq m \\ & & \vdash N^1 \leftrightarrow \bigvee_{i=1}^m N_i^2 \\ & \Downarrow & x\text{-arc} \\ \eta_i^3 & \{\Diamond A, N_i^3\} & \in ROx'(\eta) \end{array}$$

Now we reason as follows:

$$\begin{array}{lll} \vdash \wedge_{i=1}^m (N_i^2 \rightarrow \bigcirc N_i^3) & 1 & \mathit{Above} \\ \vdash (\vee_{i=1}^m N_i^2) \rightarrow (\vee_{i=1}^m \bigcirc N_i^3) & 2 & PL, 1 \\ \vdash N^1 \rightarrow (\vee_{i=1}^m \bigcirc N_i^3) & 3 & \mathit{Above}, 2 \\ \vdash N^1 \rightarrow \bigcirc(\vee_{i=1}^m N_i^3) & 4 & T7, 3 \\ \vdash \vee_{i=1}^m N_i^3 \rightarrow P & 5 & \text{Defn.}P \\ \vdash \bigcirc \vee_{i=1}^m N_i^3 \rightarrow \bigcirc P & 6 & R2, 5 \\ \vdash N^1 \rightarrow \bigcirc P & 7 & PL, 4, 6 \\ \vdash \wedge_{\eta^1 \in ROx'(\eta)} (N^1 \rightarrow \bigcirc P) & 8 & 7 \\ \vdash (\vee_{\eta^1 \in ROx'(\eta)} N^1) \rightarrow \bigcirc P & 9 & PL, 8 \\ \vdash P \rightarrow \bigcirc P & 10 & \text{Defn.}P, 9 \end{array}$$

(c) Now we show $\neg(N^1 \wedge \Diamond A)$:

$$\begin{array}{lll} \vdash P \rightarrow \neg A & 1 & \mathit{Above} \\ \vdash \Box P \rightarrow \Box \neg A & 2 & R4, 1 \\ \vdash P \rightarrow \bigcirc P & 3 & \mathit{Above} \\ \vdash \Box(P \rightarrow \bigcirc P) & 4 & RN, 3 \\ \vdash N^1 \rightarrow (P \wedge \Box(P \rightarrow \bigcirc P)) & 5 & PL, 4 \\ \vdash N^1 \rightarrow \Box P & 6 & Ax9, 5 \\ \vdash N^1 \rightarrow \Box \neg A & 7 & PL, 6, 2 \\ \vdash \neg(N^1 \wedge \Diamond A) & 8 & PL, 7 \end{array}$$

(d) Finally letting $\eta_1 = \eta, \eta \in ROx'(\eta)$ we have $\vdash \neg\eta$.

6. A node η was closed because $\Diamond\!\!\!\!-\, A \in \eta$ has been indefinitely postponed. This can happen only if $\neg A$ holds in every descendant of η.

- Let $ROwx(\eta)$ be the set constructed by the closing algorithm just before η was closed.

- Let $ROwx'(\eta)$ denote the set nodes containing $\Diamond\!\!\!\!-A$ and who are the immediate children of the state nodes in $ROwx(\eta)$.
- Define: $T = \bigvee_{\eta^1 \in ROwx'(\eta)} \wedge(\eta^1 - \{\Diamond\!\!\!\!-A\})$.

(a) We first show $T \rightarrow \neg A$. Let the expansion of a node $\eta^1 \in ROwx'(\eta)$ proceed as follows:

η^1	$\{\Diamond\!\!\!\!-A, N^1\}$	$\in ROwx'(\eta)$
	$\Downarrow *$	logical arc(s)
η_i^2	$\{\Diamond\!\!\!\!-A, N_i^2\}$	$1 \leq i \leq m$
		$Pr(\eta_i^2) = \Diamond\!\!\!\!-A$
	$\Downarrow$	logical arc
η_i^3	$\{A \vee (\bigcirc\Diamond\!\!\!\!-A \vee \oslash\Diamond\!\!\!\!-A), N_i^2\}$	$\vdash N^1 \leftrightarrow \bigvee_{i=1}^{m} N_i^2$
	$\Downarrow *$	logical arc(s)
η_{ij}^4	$\{A \vee (\bigcirc\Diamond\!\!\!\!-A \vee \oslash\Diamond\!\!\!\!-A), N_{ij}^4\}$	$1 \leq j \leq m_i', 1 \leq i \leq m$
		$Pr(\eta_{ij}^4) = A \vee (\bigcirc\Diamond\!\!\!\!-A \vee \oslash\Diamond\!\!\!\!-A)$
	$\Downarrow$	logical arc
η_{ij}^5	$\{A, N_{ij}^4\}$	$\vdash N_i^2 \leftrightarrow \bigvee_{j=1}^{m_i'} N_{ij}^4$
	closed	$A \notin \cup ROwx(\eta)$

Hence

$$\vdash N^1 \rightarrow \neg A$$
$$\vdash (\vee_{\eta^1 \in ROwx'(\eta)} N^1) \rightarrow \neg A$$
$$\vdash T \rightarrow \neg A$$

(b) We now show $T \rightarrow \bigcirc T$. Consider the expansion of $\eta^1 \in ROwx'(\eta)$:

η^1	$\{\Diamond\!\!\!\!-\, A, N^1\}$	$\in ROwx'(\eta)$
	$\Downarrow *$	logical arc(s)
η^2_i	$\{A \vee (\bigcirc \Diamond\!\!\!\!-\, A \vee \oslash \Diamond\!\!\!\!-\, A), N^2_i\}$	$1 \leq i \leq m$
		$\vdash N^1 \leftrightarrow \bigvee_{i=1}^{m} N^2_i$
	$\Downarrow *$	logical-arc(s)
η^3_{ij}	$\{\bigcirc \Diamond\!\!\!\!-\, A \vee \oslash \Diamond\!\!\!\!-\, A, N^3_{ij}\}$	$1 \leq j \leq m'_i, 1 \leq i \leq m$
		$\vdash N^2_i \leftrightarrow \bigvee_{j=1}^{m'_i} N^3_{ij}$
		$Pr(\eta^3_{ij}) = (\bigcirc \Diamond\!\!\!\!-\, A \vee \oslash \Diamond\!\!\!\!-\, A)$
	$\Downarrow$	logical arc
η^4_{ij}	$\{\bigcirc \Diamond\!\!\!\!-\, A, N^3_{ij}\}$	left child
	$\Downarrow *$	logical-arc(s)
η^5_{ijk}	$\{\bigcirc \Diamond\!\!\!\!-\, A, N^{x5}_{ijk}\}$	$1 \leq k \leq m^x_{ij}$
		state node
		$\vdash N^3_{ij} \leftrightarrow \vee_{k=1}^{m^x_{ij}} N^{x5}_{ijk}$
	$\Downarrow$	x-arc
η^6_{ijk}	$\{\Diamond\!\!\!\!-\, A, N^{x6}_{ijk}\}$	$\in ROwx'(\eta)$

Now we reason:

$\vdash \bigwedge_{k=1}^{m^x_{ij}} (N^{x5}_{ijk} \rightarrow \bigcirc N^{x6}_{ijk})$	1	*Above*
$\vdash \bigvee_{k=1}^{m^x_{ij}} (N^{x5}_{ijk}) \rightarrow (\bigvee_{k=1}^{m^x_{ij}} \bigcirc N^{x6}_{ijk})$	2	$PL, 1$
$\vdash N^3_{ij} \rightarrow (\bigvee_{k=1}^{m^x_{ij}} \bigcirc N^{x6}_{ijk})$	3	$Above, 2$
$\vdash N^3_{ij} \rightarrow \bigcirc (\bigvee_{k=1}^{m^x_{ij}} N^{x6}_{ijk})$	4	$T7, 3$
$\vdash \vee_{k=1}^{m^x_{ij}} N^{x6}_{ijk} \rightarrow T$	5	Defn.T
$\vdash \bigcirc \vee_{k=1}^{m^x_{ij}} N^{x6}_{ijk} \rightarrow \bigcirc T$	6	$R2, 5$
$\vdash N^3_{ij} \rightarrow \bigcirc T$	7	$PL, 4, 6$
$\vdash \bigwedge_{i=1}^{m} \bigwedge_{j=1}^{m'_i} (N^3_{ij} \rightarrow \bigcirc T)$	8	$PL, 7$
$\vdash (\bigvee_{i=1}^{m} \bigvee_{j=1}^{m'_i} N^3_{ij}) \rightarrow \bigcirc T$	9	$PL, 8$
$\vdash \bigwedge_{\eta^1 \in ROwx'(\eta)} (N^1 \rightarrow \bigcirc T)$	10	$Above, 9$
$\vdash (\bigvee_{\eta^1 \in ROwx'(\eta)} N^1) \rightarrow \bigcirc T$	11	$Above, 9$
$\vdash T \rightarrow \bigcirc T$	12	Defn.$T, 11$

(c) Similarly, we show $T \rightarrow \oslash T$. Consider the expansion of $\eta^1 \in ROwx'(\eta)$:

η^1	$\{\Diamond\!\!\!\!-\, A, N^1\}$	$\in ROwx'(\eta)$
	$\Downarrow *$	logical arc(s)
η_i^2	$\{A \vee (\bigcirc \Diamond\!\!\!\!-\, A \vee \oslash \Diamond\!\!\!\!-\, A), N_i^2\}$	$1 \leq i \leq m$
		$\vdash N^1 \leftrightarrow \bigvee_{i=1}^{m} N_i^2$
	$\Downarrow *$	logical-arc(s)
η_{ij}^3	$\{\bigcirc \Diamond\!\!\!\!-\, A \vee \oslash \Diamond\!\!\!\!-\, A, N_{ij}^3\}$	$1 \leq j \leq m_i', 1 \leq i \leq m$
		$\vdash N_i^2 \leftrightarrow \bigvee_{j=1}^{m_i'} N_{ij}^3$
		$Pr(\eta_{ij}^3) = (\bigcirc \Diamond\!\!\!\!-\, A \vee \oslash \Diamond\!\!\!\!-\, A)$
	$\Downarrow$	logical arc
η_{ij}^4	$\{\oslash \Diamond\!\!\!\!-\, A, N_{ij}^3\}$	right child
	$\Downarrow *$	logical-arc(s)
η_{ijk}^5	$\{\oslash \Diamond\!\!\!\!-\, A, N_{ijk}^{w5}\}$	$1 \leq k \leq m_{ij}^w$
		state node
		$\vdash N_{ij}^3 \leftrightarrow \vee_{k=1}^{m_{ij}^w} N_{ijk}^{w5}$
	$\Downarrow$	w-arc
η_{ijk}^6	$\{\Diamond\!\!\!\!-\, A, N_{ijk}^{w6}\}$	$\in ROwx'(\eta)$

Now we reason:

$\vdash \bigwedge_{k=1}^{m_{ij}^w} (N_{ijk}^{w5} \rightarrow \oslash N_{ijk}^{w6})$	1	*Above*
$\vdash \bigvee_{k=1}^{m_{ij}^w} (N_{ijk}^{w5}) \rightarrow (\bigvee_{k=1}^{m_{ij}^w} \oslash N_{ijk}^{w6})$	2	$PL, 1$
$\vdash N_{ij}^3 \rightarrow (\bigvee_{k=1}^{m_{ij}^w} \oslash N_{ijk}^{w6})$	3	$Above, 2$
$\vdash N_{ij}^3 \rightarrow \oslash (\bigvee_{k=1}^{m_{ij}^w} N_{ijk}^{w6})$	4	$T8, 3$
$\vdash \vee_{k=1}^{m_{ij}^w} N_{ijk}^{w6} \rightarrow T$	5	Defn.T
$\vdash \oslash \vee_{k=1}^{m_{ij}^w} N_{ijk}^{w6} \rightarrow \oslash T$	6	$R3, 5$
$\vdash N_{ij}^3 \rightarrow \oslash T$	7	$PL, 4, 6$
$\vdash \bigwedge_{i=1}^{m} \bigwedge_{j=1}^{m_i'} (N_{ij}^3 \rightarrow \oslash T)$	8	$PL, 7$
$\vdash (\bigvee_{i=1}^{m} \bigvee_{j=1}^{m_i'} N_{ij}^3) \rightarrow \oslash T$	9	$PL, 8$
$\vdash \bigwedge_{\eta^1 \in ROwx'(\eta)} (N^1 \rightarrow \oslash T)$	10	$Above, 9$
$\vdash (\bigvee_{\eta^1 \in ROwx'(\eta)} N^1) \rightarrow \oslash T$	11	$Above, 9$
$\vdash T \rightarrow \oslash T$	12	Defn.$T, 11$

(d) Now we show $\neg(N^1 \wedge \Diamond\!\!\!\!-A)$:

$\vdash T \rightarrow \neg A$	1	*Above*
$\vdash \boxplus T \rightarrow \boxplus \neg A$	2	$Ax1, RE, 1$
$\vdash T \rightarrow (\bigcirc T \wedge \oslash T)$	3	*Above*
$\vdash \boxplus(T \rightarrow (\bigcirc T \wedge \oslash T))$	4	$RE, 3$
$\vdash N^1 \rightarrow (T \wedge \boxplus(T \rightarrow (\bigcirc T \wedge \oslash T)))$	5	$PL, 4$
$\vdash N^1 \rightarrow \boxplus T$	6	$Ax10, 5$
$\vdash N^1 \rightarrow \boxplus \neg A$	7	$PL, 6, 2$
$\vdash \neg(N^1 \wedge \Diamond\!\!\!\!-A)$	8	$PL, 7$ ∎

(e) Finally letting $\eta_1 = \eta, \eta \in ROwx'(\eta)$ we have $\vdash \neg\eta$.

Corollary 5 (Completeness) $(\models \phi) \Rightarrow (\vdash \phi)$

Proof:

$$\begin{array}{ll} & \models \phi \\ \Rightarrow & \{\neg\phi\} \text{ is closed} \\ \Rightarrow & \vdash \neg(\neg\phi) \\ \Rightarrow & \vdash \phi \end{array}$$ ∎

Chapter 4

Propositional Ordinal Tree Logic

In the last chapter we saw how PDTL can be used to represent dense time information. We now restrict the dense time models of PDTL to have certain 'stability' and 'recurrence' properties. These properties are formalized in this chapter and we give a exponential time decision procedure for the resulting logic. These models give rise to temporal data structures which we develop in subsequent chapters.

4.1 Ordinal Trees

We consider a restricted set of omega tree models called ordinal trees and its logic: the Propositional Ordinal Tree Logic (POTL).

Definition 3 (Stability) *A proposition $p \in PROP$ is called* stable *in an omega tree T if*

$$\begin{aligned}&\exists n \geq 0, \forall (k_0, \ldots, k_n) \in \mathcal{N}^+, \forall j \geq 0, \forall \overline{m} \in \mathcal{N}^*, \\ &\quad p \in s(k_0, \ldots, k_n) \quad \textit{iff} \quad p \in s(k_0, \ldots, k_n + j, \overline{m})\end{aligned}$$

If a proposition p is stable in T then it follows that:

$$\exists n \geq 0, \forall (k_0, \ldots, k_n) \in \mathcal{N}^+, \quad T((k_0, \ldots, k_n)) \models (\boxplus p \vee \boxplus \neg p)$$

and $T((k_0, \ldots, k_n))$ is called a constant *tree.*

Definition 4 (Recurrence) *An omega tree T is called* recurring *if:*

$$\begin{aligned}&\forall \overline{k} \cdot k' \in \mathcal{N}^+, \exists k'' \geq k', \ \exists m \geq 0, \ \forall i \geq 0, \\ &\quad T(\overline{k} \cdot (k'' + i)) = T(\overline{k} \cdot (k'' + i + m))\end{aligned}$$

Definition 5 (Ordinal Tree) *An ordinal tree model for POTL is an omega tree T where:*

1. *All the propositions are stable in T.*
2. *T is recurring.*

Ordinal trees have interval based rather than point based models. For example, the formula $\boxplus(\Diamond\!\!\!\!-\, A \wedge \Diamond\!\!\!\!-\, \neg A)$ has an omega tree model but no ordinal tree models. Figure 4.1 shows an ordinal tree for $PROP = \{p_1, p_2\}$.

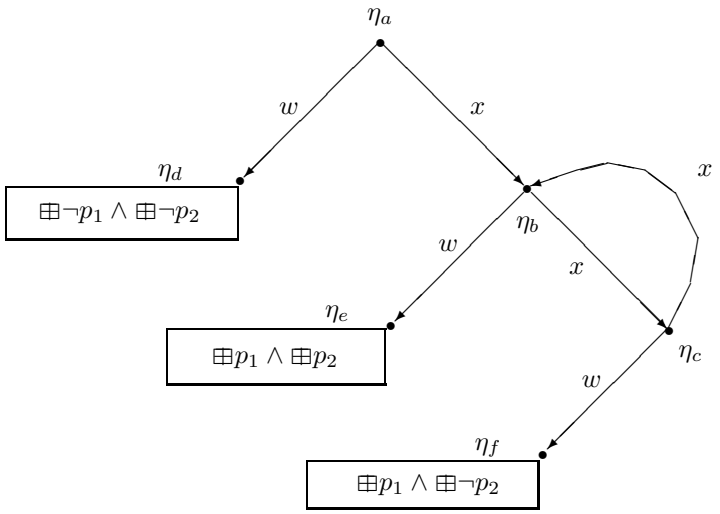


Figure 4.1: An ordinal tree

Ordinal trees are represented as finite binary trees possibly with back-x-arcs, with the following properties:

1. Each *leaf node* η is labeled by a formula from the set Σ, where

$$\begin{aligned} \sigma(A) &= \bigwedge(\{\boxplus p : p \in A \cap PROP\} \cup \{\boxplus\neg p : p \in PROP - A\}) \\ \Sigma &= \{\sigma(A) : A \subseteq PROP\} \end{aligned}$$

2. Each non-leaf node has two outgoing arcs (x and w-arcs).
3. A back arc from node η' to η is allowed only if $\eta' \in Rx(\eta)$, Here $Rx(\eta)$ is the set of nodes reachable from η by traversing one or more x-arcs. If $\eta \in Rx(\eta)$, then η is known as a *loop node*.
4. A node that is neither a *leaf node* nor a *loop node*, is known as a *split node*.

In figure 4.1, η_d, η_e, and η_f are the leaf nodes. η_a is a split node, and η_b and η_c are loop nodes. Note that each leaf node $[\eta]_w$ represents an interval between η and $[\eta]_x$ and hence an ordinal tree represents a nested sequence of intervals.

An ordinal tree T can be unrolled to get an omega tree as follows:

- A back arc from η to η' in T is unrolled by copying $T(\eta')$ below η.
- A leaf node η is replaced by the unique PDTL tree for η.

4.1.1 Language of Ordinal Trees

We introduce a set of *loop* operators: $\{[\![k]\!] : k \geq 1\}$, with the following semantics:

$$s(\overline{m} \cdot n) \models [\![k]\!]\phi \quad \leftrightarrow \quad \forall i \geq 0 : s(\overline{m} \cdot (n + ik)) \models \phi, \quad k \geq 1$$

The language L of formulas, whose elements represent ordinal trees over the propositions $PROP$ is inductively defined as follows:

$$\begin{array}{lcl} \sigma \in \Sigma & \rightarrow & \sigma \in L \\ t_l, t_r \in L & \rightarrow & (\oslash t_l, \bigcirc t_r) \in L \\ t_0, \ldots, t_k \in L & \rightarrow & [\![k+1]\!](\oslash t_0, \bigcirc \oslash t_1, \ldots, \bigcirc^k \oslash t_k) \in L,\ k \geq 0 \end{array}$$

The first condition includes all the leaf nodes in L. The second condition includes all ordinal trees whose roots are split nodes, and the third condition includes all ordinal trees whose roots are loop nodes.

Lemma 6 *1. Every ordinal tree can be represented by a formula in L.*

2. Every formula in L represents an ordinal tree.

3. The following formulas are equivalent in L:

$$\begin{array}{lcll} \boxplus\sigma & \leftrightarrow & (\oslash\boxplus\sigma, \bigcirc\boxplus\sigma) & \sigma \in \Sigma \\ \boxplus\sigma & \leftrightarrow & [\![k+1]\!](\oslash\boxplus\sigma, \bigcirc \oslash \boxplus\sigma, \ldots, \bigcirc^k \oslash \boxplus\sigma), & \sigma \in \Sigma,\ k \geq 0 \end{array}$$

Example 14 (Ordinal tree representation) The ordinal tree in figure 4.1 is given by the following formula from L:

$$(\oslash(\boxplus\neg p \wedge \boxplus\neg p_2), \bigcirc[\![2]\!](\oslash(\boxplus p_1 \wedge \boxplus p_2), \bigcirc \oslash (\boxplus p_1 \wedge \boxplus\neg p_2)))$$

4.2 Tableau and Satisfiability

The PDTL tableau procedure will be extended to check if the formula ϕ has any ordinal tree models.

4.2.1 Stable Nodes

When we build a PDTL tableau τ for ϕ, a state node η will now have a third child $[\eta]_s$, called its *stable* (s-) child, besides the two usual children $[\eta]_w$ and $[\eta]_x$. If $\eta = \{\bigwedge_i l_i, \bigwedge_j \oslash\phi_j, \bigwedge_k \bigcirc\psi_k\}$ then $[\eta]_s = \{\bigwedge_i \boxplus l_i, \bigwedge_j \phi_j\}$.

4.2.2 Closing Nodes

Define $ROswx$ descendants of an open node η as follows:

$$
\begin{array}{rcl}
ROswx(\eta) & = & \{[\eta]_l, [\eta]_r\} \cup ROswx([\eta]_l) \cup ROswx([\eta]_r) \\
 & & \text{if } \eta \text{ is a logical node} \\
ROswx(\eta) & = & \{[\eta]_x, [\eta]_w\} \cup ROswx([\eta]_w) \cup ROswx([\eta]_x) \\
 & & \text{if } \eta \text{ is a state node, and } [\eta]_s \text{ is closed} \\
ROswx(\eta) & = & \{[\eta]_x, [\eta]_w, [\eta]_s\} \cup ROswx([\eta]_w) \cup ROswx([\eta]_x) \cup ROswx([\eta]_s) \\
 & & \text{if } \eta \text{ is a state node, and } [\eta]_s \text{ is open} \\
swx\text{-leaf-scc}(\eta) & \text{iff} & \forall\mu(\mu \in ROswx(\eta) \rightarrow (\eta \in ROswx(\mu))) \text{ and} \\
 & & \forall\mu((\mu \in ROswx(\eta) \wedge \mu \text{ is a state node }) \rightarrow ([\eta]_s \text{ is open}))
\end{array}
$$

$ROswx(\eta)$ is the set of open nodes reachable from η by any path, and this set is a swx-strongly connected component (with every state node in it having three open children) when swx-leaf-scc(η) is true.

The closing algorithm applied is the same as that for PDTL for the closing conditions (1) to (4), however condition (5) now reads:

> Close η if swx-leaf-scc(η) and $\Diamond\!\!\!\!-\psi \in \eta$ and $\psi \notin \bigcup ROswx(\eta)$.

and the closing of a s-child does not affect its parent state node.

To ensure that the tableau for ϕ expands completely, we include the formula $\bigwedge_{p \in PROP(\phi)} \boxplus(p \vee \neg p)$ along with ϕ during tableau building. This formula is valid both in ordinal trees and omega trees and hence does not affect the satisfiability of ϕ.

4.2.3 State Node Tableau

The tableau τ is shortened to a *state node tableau* Δ by removing the logical nodes as follows:

- The nodes of Δ are the open state nodes of τ.
- For $\mu_1, \mu_2 \in \Delta$, if $\mu_2 \in RO([\mu_1]_x)$ in τ, then an x-arc is drawn from μ_1 to μ_2,

- For $\mu_1, \mu_2 \in \Delta$, if $\mu_2 \in RO([\mu_1]_w)$ then a w-arc is drawn from μ_1 to μ_2.
- For $\mu_1, \mu_2 \in \Delta$, if $\mu_2 \in RO([\mu_1]_s)$ then a s-arc is drawn from μ_1 to μ_2.
- Note that a node in Δ may have many x, w and s children. Hence $[\mu]_x$, $[\mu]_w$, and $[\mu]_s$ will denote sets when $\mu \in \Delta$.

4.2.4 Marking Nodes

Given a tableau Δ for ϕ, the nodes of Δ will be marked by a sequence of functions $\beta_i : \Delta \mapsto 2^L, i \geq 0$, where L is the ordinal tree language over the propositions occurring in ϕ. The marking function β_i for a large enough i, maps a node of the tableau to a set of ordinal tree models for that node.

A node $\eta \in \Delta$ having an ordinal tree model will be called *marked* at stage i, if $\beta_i(\eta)$ is non-empty. The marking will proceed bottom up: first marking nodes that have *constant* ordinal tree models, and then proceeding upwards marking nodes that can form ordinal trees from the nodes already marked. A node $\eta \in \Delta$ is *constant* if it satisfies the following condition:

$$constant(\eta) \stackrel{\text{def}}{=} \forall p \in PROP(\phi)\,((\boxplus p \in \eta) \vee (\boxplus \neg p \in \eta))$$

The s-children ensure that all swx-leaf-scc nodes have ordinal tree models. That is, for any node $\eta \in \Delta$: swx-leaf-scc(η) implies that $\bigcup ROswx(\eta)$ does not contain both p and $\neg p$ for all $p \in PROP$, and therefore η has an ordinal tree model that satisfies the formula $\sigma(\eta)$ given below:

$$\sigma(\eta) = \bigwedge \left(\begin{array}{l} \{\boxplus p : p \in (PROP \cap \bigcup ROswx(\eta))\} \cup \\ \{\boxplus \neg p : p \in (PROP - \bigcup ROswx(\eta))\} \end{array} \right)$$

and we have

$$swx\text{-leaf-scc}(\eta) \rightarrow constant(\eta)$$

Initially the constant nodes are marked: $\sigma(\eta) \in \beta_0(\eta)$ if $constant(\eta)$, otherwise $\beta_0(\eta) = \emptyset$. The initial marking β_0 is extended to $\beta_i, 0 < i \leq |\Delta|$ by the *marking algorithm* given in table 4.1. The marking can be slightly simplified by using the equivalences in L. Note that β_i is monotonic, i.e. $\beta_i(\eta) \subseteq \beta_{i+1}(\eta), i \geq 0$.

Definition 6 (satisfiable-x-loop) *The condition* sat-x-loop$(\eta_0, \ldots, \eta_k)$ *holds if the following three conditions are satisfied:*

1. $\eta_0 \in [\eta_k]_x$.
2. $\eta_{i+1} \in [\eta_i]_x, 0 \leq i < k$.

For all $\eta \in \Delta$ do

$\beta_0(\eta) := \{\sigma(\eta) : constant(\eta)\}$

Endfor

For $i := 1$ to $|\Delta|$ do

For all $\eta \in \Delta$ do

$\beta_{i+1}(\eta) := \beta_i(\eta)$

$\cup\{\ (\oslash t', \bigcirc t'')\ :$

$t' \in \beta_i(\eta'),\ t'' \in \beta_i(\eta''),$

$\eta' \in ([\eta]_w \cup [\eta]_s),\ \eta'' \in [\eta]_x\}$

$\cup\{\ [\![k+1]\!](\oslash t_0, \ldots, \bigcirc^k \oslash t_k)\ :$

$t_j \in \beta_i(\eta'_j),\ \ \eta'_j \in ([\eta_j]_w \cup [\eta_j]_s),\ 0 \leq j \leq k,$

sat-x-loop$(\eta_0, \ldots, \eta_k),\ \eta_0 = \eta\}$

Endfor

Endfor

Table 4.1: Marking Algorithm

3. $\forall \phi : \Diamond\phi \in (\eta_0 \cup \ldots \cup \eta_k) \rightarrow \phi \in (\eta_0 \cup \ldots \cup \eta_k)$.

Theorem 7 *ϕ has an ordinal tree model iff $\beta_{|\Delta|}(\{\phi\}) \neq \emptyset$, where $|\Delta|$ is the size of the POTL tableau Δ for ϕ.*

Proof: ($\Leftarrow$) The mark of the node at stage i (i.e. $\beta_i(\eta)$) itself represents a set of ordinal tree models starting at that node. The important part to note is that $\Diamond$ properties are satisfied during the construction of loops.

($\Rightarrow$) Let T be an ordinal tree model satisfying ϕ. Since T is also an omega tree, by completeness of PDTL, there is an embedding of T, $m : \mathcal{N}^+ \mapsto \Delta$ which uses only the w and x children of Δ. We now show how we can re-embed T ($m' : \mathcal{N}^+ \mapsto \Delta$) so that the constant nodes of T eventually embed in constant nodes of Δ. Once this is done, we can show all the nodes of Δ in which nodes of T embed get marked. Hence the root of Δ also gets marked.

The difficult part is to show that if $T(\overline{i} \cdot i)$ is a constant tree then $m(\overline{i} \cdot i) \in \Delta$ gets marked.

Let $T(\overline{i} \cdot i)$ be a constant tree. If $constant(m(\overline{i} \cdot i))$ then $\sigma(m(\overline{i} \cdot i)) \in \beta_0(\overline{i} \cdot i) \neq \emptyset$. Otherwise consider the chain of x-children $\{s(\overline{i} \cdot i + n) : n \geq 0\}$ in T until $m(\overline{i} \cdot i + n)$ repeats:

T	$s(\overline{i} \cdot i)$	$\ldots$	$s(\overline{i} \cdot i + j)$	$\ldots$	$s(\overline{i} \cdot i + j + k)$	$s(\overline{i} \cdot i + j + k + 1)$
Δ	$m(\overline{i} \cdot i)$	$\ldots$	$m(\overline{i} \cdot i + j)$	$\ldots$	$m(\overline{i} \cdot i + j + k)$	$m(\overline{i} \cdot i + j + k + 1)$
Δ	μ_i	$\ldots$	μ_{i+j}	$\ldots$	μ_{i+j+k}	μ_{i+j}

Since $T(\overline{i} \cdot i)$ is a constant tree, μ_i will have a marked open s-child μ'_i such that $constant(\mu'_i)$. This stable child was created In general this holds for every μ'_n for $i \leq n \leq (i + j + k)$.

The constant subtrees $\{T(\overline{i} \cdot i + n \cdot 0) : n \geq 0\}$ may as well be embedded in the marked constant nodes $\{\mu'_n : i \leq n \leq i + j + k\}$ as follows:

$$\begin{array}{lcll} m'(\overline{i} \cdot n \cdot 0) & = & \mu'_n & i \leq n \leq i + j + k \\ m'(\overline{i} \cdot n \cdot 0) & = & \mu'_{i+j+(n-i-j)\ mod\ (k+1)} & n > i + j + k \end{array}$$

Moreover the constant trees $\{T(\overline{i} \cdot n) : n \geq i\}$ will be embedded in the parents of the constant nodes: $\{\mu_n : i \leq n \leq i + j + k\}$.

By the marking algorithm:

- $[\![k + 1]\!](\oslash\sigma(\mu'_{i+j}), \ldots, \bigcirc^k \oslash \sigma(\mu'_{i+j+k})) \in \beta_1(m(\overline{i} \cdot i + j))$, this mark simplifies to $\sigma(\mu'_{i+j})$.
- $(\oslash\sigma(\mu'_{i+j-1}), \bigcirc t) \in \beta_2(m(\overline{i} \cdot i + j - 1))$, for some $t \in \beta_1(m(\overline{i} \cdot i + j))$, and this mark too simplifies to $\sigma(\mu'_{i+j-1})$.

- Eventually $\sigma(\mu_i') \in \beta_{j+1}(m(\overline{i} \cdot i + 0))$.

Hence $\beta_{j+1}(m(\overline{i} \cdot i)) \neq \emptyset$, that is, if a constant subtree of T is embedded in μ_i then μ_i is marked.

Once the constant nodes of T are embedded in marked nodes, the remaining nodes of Δ where non-constant subtrees of T are embedded get marked in a finite number of steps. Finally the root of Δ gets marked.

Observe that if no new node gets marked at a stage i, then no new nodes will get marked at any later stage $j, j \geq i$. Hence at least one node gets marked at every stage of marking and $|\Delta|$ steps should be enough to mark the root.

∎

Corollary 8 *The problem of testing whether a formula ϕ is satisfiable in POTL is decidable in exponential time.*

Proof: The complexity of the marking algorithm is $O(|\tau|^2(|E| + |\tau|))$, where $|E|$ is the number of edges in τ, $|\tau| \leq 2^{4|\phi|}$ and $|E| \leq 2^{8|\phi|}$. Note that the marking algorithm need only track whether the β_i are empty or not. ∎

Example 15 (POTL tableau) The formula $\boxplus(p \vee \neg p)$ has an ordinal tree model, the stable swx-leaf-sccs in its tableau (figure 4.2) are η_c and η_f. The tableau has been simplified for exposition. The simplified marking of its nodes is given below:

	η_a	η_b	η_c	η_d	η_e	η_f
β_0	$\emptyset$	$\emptyset$	C	$\emptyset$	$\emptyset$	F
β_1	C	F	C	$\emptyset$	$\emptyset$	F
β_2	C	F	C	C	F	F

where $\boxplus p \in C$ and $\boxplus \neg p \in F$.

Example 16 The formula $\boxplus(p \vee \neg p) \wedge \boxplus \diamond p \wedge \boxplus \diamond \neg p$ has no ordinal tree models, because it has no swx-leaf-sccs. In particular the nodes η_c and η_f of figure 4.2 will be closed because η_c (correspondingly η_f) form a swx-leaf-scc which does not satisfy $\neg p$ (respectively p).

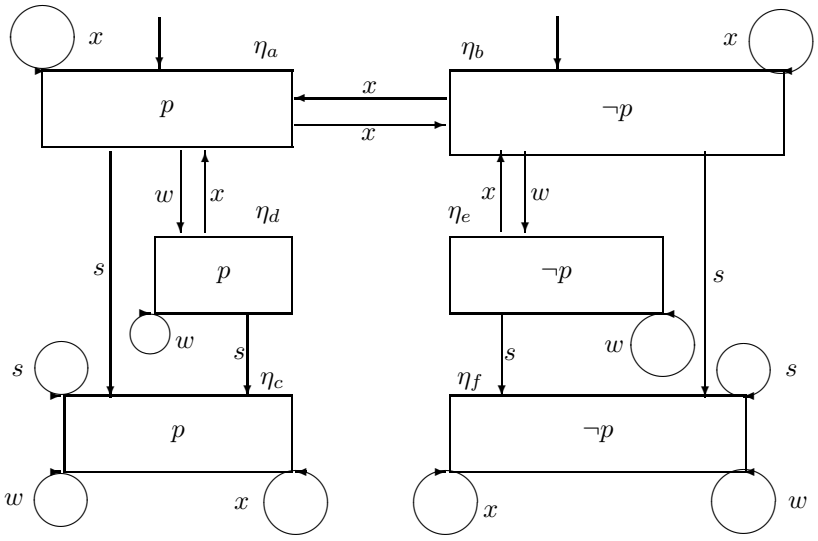


Figure 4.2: Ordinal tree tableau for $\boxplus(p \vee \neg p)$

Chapter 5

Decidability by Interpretations

This chapter explores the relation between our logics and conventional logics with respect to the decidability problem. This allows us to get a feel for what the decision problem for our logic is like. It also shows that the decision procedure we have presented using the tableau method is superior in terms of complexity than a (more simpler) decidability result that can be obtained via interpretation.

We can decide PDTL and POTL by interpreting them in the following logics:

[S2S] The Monadic Second Order Theory of Two Successor Functions[Rab69, Rab77, Tho90].

[PDL] Propositional Dynamic Logic[Par81, Har84, KT90].

5.1 Monadic Second Order Logic of Two Successor Functions

The monadic second order theory of two successor functions is popularly known as S2S. The infinite binary tree defined below serves as a structure for interpreting its sentences [Rab69, Rab77, Tho90].

Definition 7 *(Infinite Binary Tree) The structure* $T =< \{0,1\}^*, \Lambda, r_0, r_1 >$ *is an infinite binary tree rooted at* Λ*. The set of all finite strings (*$\{0,1\}^*$*), on the alphabet* $\{0,1\}$ *represent the nodes of the tree, and the two successor functions* r_0 *and* r_1 *map a node to its left and right child respectively. Hence* $r_0(u) = u \cdot 0$ *and* $r_1(u) = u \cdot 1$*.*

5.1.1 Definable sets in S2S

The set A is an infinite path from the root, if the root Λ is in the path A and for every node u in the path exactly one of its child is in the path, and if a node v is not in the path then none of its

children is in the path:

$$\begin{aligned} path(A) \stackrel{\text{def}}{=} \; & \Lambda \in A \\ & \wedge \forall u(u \in A \rightarrow (r_0(u) \in A \leftrightarrow r_1(u) \notin A)) \\ & \wedge \forall v(v \notin A \rightarrow (r_0(v) \notin A \wedge r_1(v) \notin A)) \end{aligned}$$

Node u is the ancestor of node v, if every infinite path through v also passes through u:

$$u \leq v \stackrel{\text{def}}{=} \forall A : (path(A) \wedge (v \in A)) \rightarrow (u \in A)$$

Node u lexicographically precedes node v, if either u is an ancestor of v, or they have a common ancestor z such that $r_0(z)$ is an ancestor of u and $r_1(z)$ is an ancestor of v:

$$u \preceq v \stackrel{\text{def}}{=} u \leq v \vee \exists z(r_0(z) \leq u \wedge r_1(z) \leq v)$$

The set A is finite, if every subset B of it has a minimum v_1 and maximum v_2 under the ordering '$\preceq$':

$$fin(A) \stackrel{\text{def}}{=} \forall B(B \subseteq A \rightarrow \exists v_1, v_2(v_1, v_2 \in B \wedge \forall z(z \in B \rightarrow v_1 \preceq z \preceq v_2)))$$

To pick the smallest set A, among the sets that satisfy a given property $\Phi(X)$, we introduce the following syntactic abbreviation:

$$\mu A : \Phi(A) \stackrel{\text{def}}{=} A : (\Phi(A) \wedge \forall B(\Phi(B) \rightarrow A \subseteq B))$$

5.1.2 Sets defined by Regular Expressions

Let $\Pi = \{r_0, r_1, r_0^-, r_1^-\}$ be the set of program symbols that relate adjacent nodes in the tree. We now define $R(re, u)$, the set of nodes reachable from u by paths given by the regular expression re (constructed using the operations $\{\cdot, ^*, +\}$ on Π).

$$\begin{aligned} R(r_0, u) & \stackrel{\text{def}}{=} \{r_0(u)\} \\ R(r_1, u) & \stackrel{\text{def}}{=} \{r_1(u)\} \\ R(r_0^-, v) & \stackrel{\text{def}}{=} \{u\}, \text{ if } u = r_0(v) \\ & \stackrel{\text{def}}{=} \emptyset, \text{ otherwise} \\ R(r_1^-, v) & \stackrel{\text{def}}{=} \{u\}, \text{ if } u = r_1(v) \\ & \stackrel{\text{def}}{=} \emptyset, \text{ otherwise} \\ R(re_1 + re_2, u) & \stackrel{\text{def}}{=} R(re_1, u) \cup R(re_2, u) \\ R(re_1 \cdot re_2, u) & \stackrel{\text{def}}{=} \mu B : z \in B \leftrightarrow \exists v(v \in R(re_1, u) \wedge z \in R(re_2, v)) \\ R(re^*, u) & \stackrel{\text{def}}{=} \mu B : (u \in B) \wedge \forall v(v \in B \rightarrow R(re, v) \subseteq B) \end{aligned}$$

For example, the set of nodes reachable from u by applying r_1 any number of times is:

$$R(r_1^*, u) = \mu B : u \in B \wedge \forall v(v \in B \rightarrow r_1(v) \in B)$$

and set of all nodes reachable from u is

$$R((r_0 + r_1)^*, u) = \mu B : u \in B \wedge \forall v(v \in B \rightarrow (r_0(v) \in B) \wedge (r_1(v) \in B))$$

5.1.3 Decidability of S2S

Theorem 9 *(Rabin) S2S is decidable in nonelementary time.*

Given any sentence ϕ of S2S, Rabin [Rab69] constructs a nondeterministic-tree-automata $\mathcal{A}_\phi$, such that the tree language $\mathcal{L}_\phi$ accepted by $\mathcal{A}_\phi$ corresponds exactly to the set of models for ϕ. Unsatisfiability of ϕ is then decided by a procedure that checks whether $\mathcal{L}_\phi$ is empty. Rabin shows the construction of $\mathcal{A}_{\phi\wedge\psi}$, $\mathcal{A}_{\neg\phi}$ and $\mathcal{A}_{\exists X\phi}$, using $\mathcal{A}_\phi$ and $\mathcal{A}_\psi$. He also shows how to decide whether $\mathcal{L}_\phi$ is empty. The construction of $\mathcal{A}_{\neg\phi}$ from $\mathcal{A}_\phi$ involves an exponential blow-up in the size of the automata. Hence the time complexity of the decision procedure for S2S is non-elementary.

5.1.4 Decidability by Interpretation in S2S

The logics PDTL and POTL can be decided by interpretations in S2S. A PDTL formula ϕ, is transformed to a S2S formula $t_{PDTL}(\phi)$, such that ϕ is satisfiable in PDTL if and only if $t_{PDTL}(\phi)$ is satisfiable in S2S. We associate a set variable P_i with each PDTL proposition p_i occurring in the formula and impose the following condition on the P_i:

$$pdtl(P) \stackrel{\text{def}}{=} \forall u : (u \in P) \leftrightarrow (r_0(u) \in P)$$

The translation of a PDTL formula to an S2S formula is:

$$\begin{array}{lcl}
t_{PDTL}(\phi) & = & \exists P_1 \ldots P_n(\bigwedge_{i=1}^{n} pdtl(P_i) \wedge t(\Lambda, \phi)) \\
 & & \text{where } PROP(\phi) = \{p_1, \ldots, p_n\} \\
t(u, \mathcal{F}) & = & \mathcal{F} \\
t(u, p_i) & = & u \in P_i, \quad \text{if } p_i \in PROP \\
t(u, \phi \rightarrow \psi) & = & t(u, \phi) \rightarrow t(u, \psi) \\
t(u, \neg\phi) & = & t(u, \phi \rightarrow \mathcal{F}) \\
t(u, \oslash\phi) & = & t(r_0(u), \phi) \\
t(u, \bigcirc\phi) & = & t(r_1(u), \phi) \\
t(u, \Box\phi) & = & \forall v(v \in R(r_1^*, u) \rightarrow t(v, \phi)) \\
t(u, \boxplus\phi) & = & \forall v(v \in R((r_0 + r_1)^*, u) \rightarrow t(v, \phi))
\end{array}$$

The PDTL translation is now modified to decide satisfiability of POTL formulas. We include the condition that the propositions stabilize on all paths A having infinitely many r_0 transitions:

$$\begin{array}{lcl}
inf_{r_0}(A) & \stackrel{\text{def}}{=} & path(A) \wedge \exists B(\forall u(u \in B \leftrightarrow \{u, r_0(u)\} \subseteq A) \wedge \neg fin(B)) \\
potl(P) & \stackrel{\text{def}}{=} & pdtl(P) \wedge \forall A\,(inf_{r_0}(A) \rightarrow (fin(A \cap P) \vee fin(A - P))) \\
t_{POTL}(\phi) & = & \exists P_1 \ldots P_n(\bigwedge_{i=1}^{n} potl(P_i) \wedge t(\Lambda, \phi)) \\
 & & \text{where } PROP(\phi) = \{p_1, \ldots, p_n\}
\end{array}$$

Lemma 10

$$\begin{aligned} \models_{PDTL} \phi &\quad\leftrightarrow\quad \models_{S2S} t_{PDTL}(\phi) \\ \models_{POTL} \phi &\quad\leftrightarrow\quad \models_{S2S} t_{POTL}(\phi) \end{aligned}$$

The following relation totally orders all the nodes:

$$\begin{aligned} u <_t v &\quad\stackrel{\text{def}}{=}\quad v \in R(r_0^* \cdot r_1 \cdot (r_0 + r_1)^* + (r_0^- + r_1^-)^* \cdot r_0^- \cdot r_1 \cdot (r_0 + r_1)^*, u) \\ u =_t v &\quad\stackrel{\text{def}}{=}\quad v \in R((r_0)^* + (r_0^-)^*, u) \end{aligned}$$

This can be used to define the temporal operator *henceforth*.

5.2 Decidability by Interpretation in PDL

Given a PDTL formula ϕ, we translate it to a PDL formula $t'_{PDTL}(\phi)$, such that ϕ is satisfiable in PDTL if and only if $t'_{PDTL}(\phi)$ is satisfiable in PDL:

$$\begin{aligned} t'_{PDTL}(\phi) &= \textstyle\bigwedge_{p \in PROP(\phi)} [(r_0 + r_1)^*] \begin{pmatrix} t(\phi) \wedge \\ p_i \leftrightarrow [r_0] p_i \wedge \\ <r_0>(p_i \vee \neg p_i) \wedge \\ <r_1>(p_i \vee \neg p_i) \end{pmatrix} \\ t(\mathcal{F}) &= \mathcal{F} \\ t(p) &= p, \quad p \in PROP \\ t(\phi \rightarrow \psi) &= t(\phi) \rightarrow t(\psi) \\ t(\neg\phi) &= t(\phi \rightarrow \mathcal{F}) \\ t(\oslash\phi) &= [r_0] t(\phi) \\ t(\bigcirc\phi) &= [r_1] t(\phi) \\ t(\Box\phi) &= [r_1^*] t(\phi) \\ t(\boxplus\phi) &= [(r_0 + r_1)^*] t(\phi) \\ t(\Diamond\phi) &= < r_1^* > t(\phi) \\ t(\diamondplus\phi) &= < (r_0 + r_1)^* > t(\phi) \end{aligned}$$

Lemma 11

$$\models_{PDTL} \phi \quad\leftrightarrow\quad \models_{PDL} t'_{PDTL}(\phi)$$

Chapter 6

Dense Time Logic Programming

Having introduced some logical systems for dense time, we now explore the utility of our systems for developing practical programming systems.

In this chapter we show how ordinal trees can be used as temporal data structures in a conventional logic programming language, and a *dense time logic programming* (DTLP) paradigm can built around this temporal data structure.

Temporal horn clauses based on ordinal trees allow us to store temporal facts, rules and queries. A DTLP interpreter implemented in prolog manipulates temporal horn clauses to prove temporal queries from temporal facts and rules.

6.1 Omega Trees

An omega tree is an infinite binary tree representing the rational line of time (see figure 6.1). A node of the omega tree represents a closed-open interval of time. Its root corresponds to the whole interval of time, starting from the time point zero. If a node η corresponds to an interval starting at time point t, then its right child (*next*-child or x-child) $[\eta]_x$ corresponds to sub-interval beyond the 'next' time point; and its left child (*within*-child or w-child) $[\eta]_w$ corresponds to the sub-interval between t and the *next* time point. Hence an omega tree represents a nested sequence of intervals. The arc connecting η to $[\eta]_x$ (respectively $[\eta]_w$) is called an x-arc (respectively w-arc).

Given a node η of a tree t, we denote the subtree rooted at η by $t(\eta)$, and we can pick a particular subnode of η by applying the operators *next* $(_x)$ and *within* $(_w)$ on it. For example in figure 6.1 the root is η_1 and $\eta_2 = [\eta_1]_w, \eta_3 = [\eta_1]_x, \eta_6 = [\eta_3]_w = [\eta_1]_{xw}$.

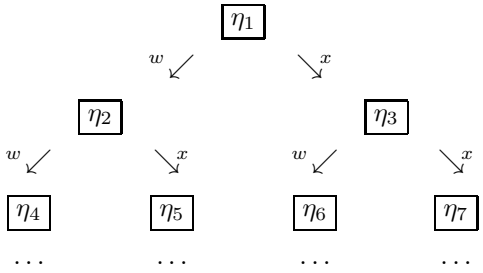


Figure 6.1: Omega tree

6.1.1 Labeling nodes and Variables

The nodes of the omega tree are labeled by formulas of some object language Σ to assert facts (or queries) about those time intervals. A labeled omega tree t is a mapping from the nodes of t to Σ. Here Σ will be the set of prolog clauses over some fixed language. We use '`T`' to indicate an unlabeled node.

Prolog variables occurring in different positions in the same tree refer to the same entity, and hence are the usual *rigid* or global variables of temporal programming languages. While identical prolog variables in different trees are different. We use `X, Y, Z` for rigid variables. We extend DTLP with flexible variables later on, and will use the symbols `A, B, C` for them.

6.2 Ordinal Trees

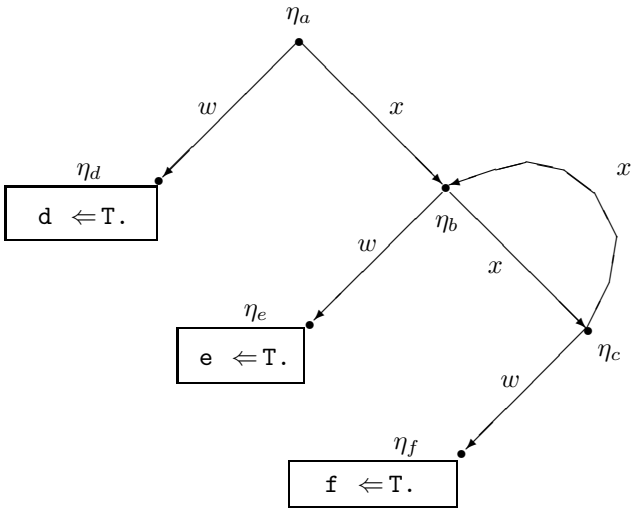


Figure 6.2: An ordinal tree

Omega trees with regular labelings (called ordinal trees) can be finitely represented by labeled directed graphs as shown in figure 6.2. Such graphs have three types of nodes: *split*, *leaf* and *loop* nodes:

1. A *Split*-node is the usual node of the omega tree. A split node η breaks up an interval into two sub-intervals: the present and the future, accessible respectively by its two children: $[\eta]_w$ and $[\eta]_x$. In figure 6.2, η_a is a split node.

2. If every descendant of a node η is labeled by the same formula, then η is made into a *leaf*-node. Each *leaf* node $[\eta]_w$ represents the initial subinterval of η up to the subinterval $[\eta]_x$. In figure 6.2, η_d, η_e and η_f are the leaf nodes. It is sufficient to label only the leaf nodes.

3. A *loop* is created when there is periodicity among the x-descendants of the omega tree, that is:

$$\exists k \geq 0 \; \forall i \geq 0 \; t([\eta]_{x^i}) = t([\eta]_{x^{i+k}})$$

   Then the tree $t([\eta]_{x^k})$ is replaced by a back-x-arc to η. The nodes $[\eta]_{x^i}, 0 \leq i < k$ are called *loop-nodes*. In figure 6.2, η_b and η_c are loop nodes.

Example 17 (Nested loops) The ordinal tree in figure 6.3 shows nested loops, the activity represented by tree $t(\eta_c)$ occurs repetitively from the second state onwards, which itself consists of a repetition of two alternating sub-activities at η_e and η_g.

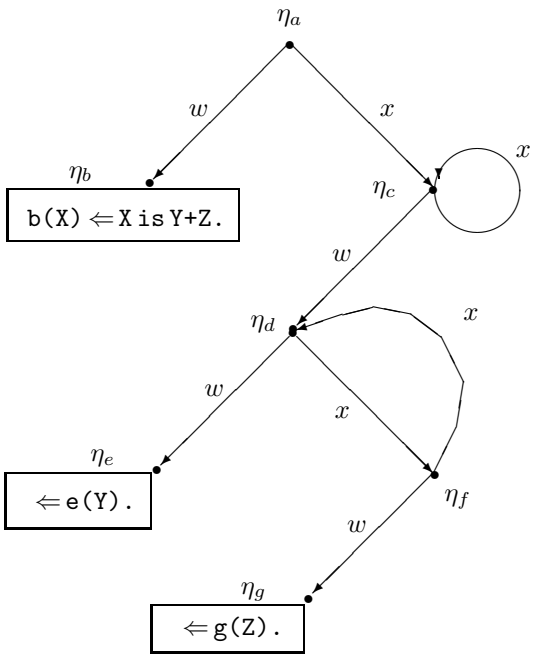


Figure 6.3: Nested loops

6.3 Representing Ordinal Trees in Prolog

Ordinal trees can be represented in prolog using nested lists, and by declaring `next` ($\bigcirc$) and `within` ($\oslash$) as prefix operators. Loops are represented by the binary functor `loop(i,_)` ($[\![i]\!]$), where i denotes the size of the loop.

The prolog clause labeling a leaf node is written as itself, We use the symbol '$\Leftarrow$' (read *if*) to differentiate it from prolog's '`:-`'. A tree whose root is a split node η, is written as: $[\oslash L_1, \bigcirc L_2]$, where L_1 and L_2 represent $t([\eta]_w)$ and $t([\eta]_x)$ respectively. A *loop* of $k+1$ nodes: $[\eta]_{x^i, 0 \leq i \leq k}$ where $t([\eta]_{x^{k+1}}) = t(\eta)$, is written as:

$$[\![k+1]\!]([\oslash L_0, \bigcirc \oslash L_1, \ldots, \bigcirc^k \oslash L_k])$$

where the L_i represents $t([\eta]_{x^i w})$ for $0 \leq i \leq k$. Note that $\bigcirc^i$ denotes $\bigcirc$ repeated i times. $\bigcirc^0$ is written merely for clarity and is not present in the corresponding clause.

Example 18 The tree in figure 6.2 can be written as:

```
[ ⊘d ⇐ T,
  ◯[[2]]( [
    ◯⁰⊘e ⇐ T,
    ◯¹⊘f ⇐ T])]
```

Example 19 The tree in figure 6.3 can be written as:

```
[ ⊘b(X) ⇐ X is Y + Z,
  ◯[[1]]( [
    ⊘[[2]]( [
      ◯⁰⊘ ⇐ e(Y),
      ◯¹⊘ ⇐ f(Z) ])])]
```

6.4 Standardizing Ordinal Tree Representation

The operators $\oslash$, $\bigcirc$, and $[\![i]\!]$ distribute over their list arguments, moreover $[\![i]\!]$ and $\bigcirc$ commute.

Given a labeled ordinal tree, its representation is said to be in *standard form* when the operators are distributed over their list arguments, and the $[\![i]\!]$ is always followed by $\oslash$. Each element of the resulting list is called a *temporal-predicate*, which expresses a fact (or goal) about a node of the ordinal tree. It is sufficient to label only the leaf nodes of the ordinal trees and the '`T`' labels can

be conveniently omitted from the list. The resulting list is called a *temporal horn clause* if it has at most one non-negated temporal predicate (called the *head*), and zero or more negated temporal predicates (called the *body*). If the body is absent, then the temporal clause is just a list of *temporal facts*. For convenience the missing body is replaced by `T`. If the head is absent, it is a *temporal query*. If it has a body and a single head, it is a *temporal rule*.

Example 20 (Sequence of events, standardizing) Consider the ordinal tree for the following facts: 'The ball hit the wall before rolling into the drain.'

```
[ ⊘( hit(ball,wall) ⇐),
  ◯[ ⊘( rolling(ball,ground) ⇐),
    ◯( in(ball,drain) ⇐)]]
```

by distributing and rearranging the operators, we get the following three temporal facts:

```
⊘hit(ball,wall) ⇐T,
◯⊘rolling(ball,ground) ⇐T,
◯◯in(ball,drain) ⇐T.
```

Example 21 (Infinite sub-activity, standardizing) 'The ball fell from the table and bounced before rolling into the drain.'

```
[ ⊘( falling(ball,table) ⇐),
  ◯[ ⊘⟦2⟧( [
      ⊘( moving(ball,up) ⇐),
      ◯⊘( moving(ball,down) ⇐) ] )
    ◯[ ⊘( rolling(ball,ground) ⇐),
      ◯( in(ball,drain) ⇐)]]]
```

standardizing, we get the following temporal clauses:

```
⊘falling(ball,table) ⇐T.
◯⊘⟦2⟧ ⊘moving(ball,up) ⇐T.
◯⊘◯⟦2⟧ ⊘moving(ball,down) ⇐T.
◯◯⊘rolling(ball,ground) ⇐T.
◯◯◯in(ball,drain) ⇐T.
```

6.5 Temporal Clauses

Temporal clauses are of the following three types: facts, rules and goals.

6.5.1 Facts

A fact ordinal tree is labeled by prolog facts. By standardizing the ordinal tree, we get a set of temporal facts. In the examples that follow assume that time is divided into an infinite sequence of days.

Example 22 'It will rain the whole day and there will be a flood in the latter half of the day.'

$\oslash rain \Leftarrow$ `T.`
$\oslash\bigcirc flood \Leftarrow$ `T.`

Example 23 (Loop information) 'It rains 10 units every Tuesday.'

$\bigcirc\bigcirc[\![7]\!]\ \oslash rain(10) \Leftarrow$ `T.`

Example 24 (Using cut) 'He is always late, except on Tuesdays.'

$\bigcirc\bigcirc[\![7]\!]\oslash late \Leftarrow$ `!,fail.`
$late \Leftarrow$ `T.`

However it is not possible to assert temporally existential facts. For example, it is not possible to assert: 'On some day there will be rain'.

6.5.2 Queries

A temporal query is a list of temporal goals, where each temporal goal is a temporal predicate.

Example 25 (Simple query) 'Was the janitor absent on Wednesday?'

`?=` $\bigcirc\bigcirc\bigcirc\oslash absent(janitor)$`.`

Example 26 (Query with rigid variables) 'Will there be rain on Monday and a flood on Tuesday, and will the rain be twice the amount of the flood?'

`?=` $\bigcirc\oslash rain(X)$`, ` Y `is` X `div 2,`
$\bigcirc\bigcirc\oslash flood(Y)$`.`

6.5.3 Rules

Temporal rules are of three types, classified according to the nodes (of the query ordinal tree) where they can be applied:

1. **Universal Rules.** These can be applied at any node. Universal rules in a sense have instantaneous effect, and are used to express time-independent rules. They are prefixed by the operator `every` ($\boxplus$). Universal rules have no temporal structure because a tree universal rule leads to degeneracy as it can be applied recursively *within* a subinterval.

   Example 27 (Universal rule) 'The bulb is on, whenever the switch is on.'

   $\boxplus$`(` $on(bulb)$ $\Leftarrow on(switch)$ `)`

   Example 28 The following universal rule is obviously time independent:

   $\boxplus$`(` $odd(X)$ $\Leftarrow$`( 1 is` X `mod 2 ))`

2. **Anchored Rules.** These can be applied only to the root of a query. It expresses temporal relations over fixed time intervals, and is useful in asserting generalized temporal facts.

   Example 29 The gas pressure (P) is k times the temperature (T) over the initial sub-interval:

   $\oslash pr(P)$ $\Leftarrow \oslash temp(T)$`, ` $\oslash$`( P is k * T ).`

   which is equivalent to:

   $\oslash pr(P)$ $\Leftarrow \oslash temp(T)$`, P is k * T.`

3. **Level Rules.** These can be applied on any node of a query at specific level only, where the *level* of a node is defined as the number of w-arcs between the node and the root. Such rules are of the form:

   $\Box(\oslash\Box)^i$ `(` $\bigcirc^j$ $\oslash$`Q` $\Leftarrow \bigcirc^k$ $\oslash$`B ),` $\quad i, j, k \geq 0$`.`

   where i is the level at which this rule is applicable.

   Example 30 (Level 0 rule, causal rule) 'Whenever it rains the janitor is absent the next day.' This rule is applicable on the level of days. To indicate that the rule is applicable on *all* days we prefix it by the operator `all` ($\Box$):

   $\Box$`(`$\bigcirc\oslash absent(janitor)$ $\Leftarrow \oslash rain$ `).`

   This is also an example of a *causal rule* as there is no body to the future of the head. Similarly a rule applicable on all hours of all days would be prefixed by '`all within all`' ($\Box\oslash\Box$).

Example 31 (Level 1 rule) 'Whenever it rains more than 20 units for a full hour, the buses are late for the next one hour.'

$$\Box\oslash\Box(\bigcirc\oslash late(bus) \Leftarrow \oslash rain(X),\ X \geq 20\ ).$$

6.6 Temporal Resolution

Temporal resolution tries to prove a given query from a set of temporal facts and rules. Given a list of temporal goals, we try to prove the goals from the left to the right. A temporal goal is proved if any one of the following holds:

- It is established by a temporal fact over the same or a larger interval.
- It is established by a temporal rule over the same or a larger interval, and whose temporal goals in turn are provable.
- Its interval can be broken up, and the goal can be proved in each of the broken interval.

A sample session with the DTLP interpreter is given in appendix A. The transfer of temporal information between a goal and a head is effected by a mechanism called *aligning*, which we explain in the next section.

6.6.1 The Aligning Algorithm

The algorithm *align*, when given a temporal goal, and a temporal rule or fact, tries to align the intervals of the goal and the head of a rule. If the goal and the head overlap then the goal is broken up into two sets: the *overlapping* and the *non-overlapping* sets. If the head of the rule and the overlapping goal unify then the body of the rule along with the non-overlapping goal is returned.

The algorithm considers the following cases:

	$\boxplus R$	$\Box R$	$\oslash R$	H	$\oslash H$	$\bigcirc H$	$[\![j]\!]\oslash H$
Q	$E1$	$L1$	$L5$	$A1$	$A5$	$A9$	$A13$
$\oslash Q$	$E2$	$L2$	$L6$	$A2$	$A6$	$A10$	$A14$
$\bigcirc Q$	$E3$	$L3$	$L7$	$A3$	$A7$	$A11$	$A15$
$[\![i]\!]\oslash Q$	$E4$	$L4$	$L8$	$A4$	$A8$	$A12$	$A16$

Where R denotes a rule, H a head of a rule, and Q a query. There are 28 cases involving the

structure of the goal and the head. The following identities are used during aligning:

$$\begin{aligned} \Box A &\leftrightarrow (A \wedge \bigcirc \Box A) \\ \boxplus A &\leftrightarrow (A \wedge \bigcirc \boxplus A \wedge \oslash \boxplus A) \\ [\![n]\!] A &\leftrightarrow (A \wedge \bigcirc^n [\![n]\!] A) \end{aligned}$$

Aligning $[\![i]\!] \oslash C$ with $\bigcirc D$, requires that the loop be unrolled. Loop events are *unrolled*, so that the first few occurrences of the repetitive events are made explicit.

Example 32 (Loop Unrolling) 'The bouncing ball':

$[\![2]\!]\ \oslash moving(ball, down)\ \Leftarrow \mathtt{T},$
$\bigcirc [\![2]\!]\ \oslash moving(ball, up)\ \Leftarrow \mathtt{T}.$

is the same as: 'The ball fell down and started bouncing':

$\oslash moving(ball, down) \Leftarrow \mathtt{T},$
$\bigcirc [\![2]\!]\ \oslash moving(ball, up)\ \Leftarrow \mathtt{T},$
$\bigcirc\bigcirc [\![2]\!]\ \oslash moving(ball, down)\ \Leftarrow \mathtt{T}.$

6.6.2 Aligning Examples

Example 33 (Universal rule) *Cases E1,E2,E3,E4*

`Query:` ?= $\bigcirc\bigcirc[\![3]\!]\ \oslash\bigcirc\oslash$Q.

`Universal Rule:` $\boxplus$(Q $\Leftarrow$B).

`Proved:` $\bigcirc\bigcirc[\![3]\!]\ \oslash\bigcirc\oslash$Q.

`ToProve:` $\bigcirc\bigcirc[\![3]\!]\ \oslash\bigcirc\oslash$B.

Example 34 (Level 0 rule) *Case L1*

`Query:` ?= Q

`Level Rule:` $\Box$($\oslash$Q $\Leftarrow\oslash$B).

`Proved:` $[\![1]\!]\ \oslash$Q.

`ToProve:` $[\![1]\!]\ \oslash$B.

Example 35 (Level 1 rule) *Cases L3,L6*

`Query:` ?= $\bigcirc\bigcirc\oslash\bigcirc\bigcirc\oslash$Q.

`Level Rule:` $\Box\ \oslash\Box($ $\bigcirc\oslash$Q $\Leftarrow\oslash$B).

`Proved:` $\bigcirc\bigcirc\oslash\bigcirc\bigcirc\oslash$Q.

`ToProve:` $\bigcirc\bigcirc\oslash\bigcirc\oslash$B.

Cases L7,L8 do not occur.

Example 36 (Level 0 rule and a loop) *Case L4*

Query: ?= $[\![3]\!]\ \oslash$Q.

Level Rule: $\Box$($\oslash$Q $\Leftarrow\oslash$B).

Proved: $[\![3]\!]\ \oslash$Q.

ToProve: $[\![3]\!]\ \oslash$B.

Example 37 (Level 0 rule) *Case L2*

Query: ?= $\oslash$Q.

Level Rule: $\Box$($\oslash$Q $\Leftarrow\oslash$B).

Proved: $\oslash$Q.

ToProve: $\oslash$B.

Example 38 (Similar loops) *Cases A1,A6,A11,A16*

Query: ?= $\bigcirc[\![5]\!]\ \oslash$Q.

Anchored Rule: $\bigcirc[\![5]\!]\ \oslash$Q $\Leftarrow$B.

Proved: $\bigcirc[\![5]\!]\ \oslash$Q.

ToProve: B.

Example 39 (Anchored Rule) *Cases A2,A3,A4*

Query: ?= $\bigcirc[\![5]\!]\ \oslash$Q.

Anchored Rule: Q $\Leftarrow$B.

Proved: $\bigcirc[\![5]\!]\ \oslash$Q.

ToProve: B.

Example 40 (Query splitting on an Anchored Rule) *Cases A5,A9*

Query: ?= Q.

Anchored Rule: $\bigcirc\oslash$Q $\Leftarrow$B.

Proved: $\bigcirc\oslash$Q.

ToProve: $\oslash$Q, $\bigcirc\bigcirc$Q, B.

Example 41 (Query splitting into a loop) *Case A13*

Query: ?= Q.

Anchored Rule: $[\![5]\!]\ \oslash$Q $\Leftarrow$B.

Proved: $[\![5]\!]\ \oslash$Q.

ToProve: $\bigcirc[\![5]\!]\ \oslash$Q, $\bigcirc\bigcirc[\![5]\!]\ \oslash$Q,
$\bigcirc\bigcirc\bigcirc[\![5]\!]\ \oslash$Q, $\bigcirc\bigcirc\bigcirc\bigcirc[\![5]\!]\ \oslash$Q, B.

Example 42 (Rule loop unrolling) *Cases A14,A15*

Query: ?= ◯◯◯◯◯⊘Q.

Anchored Rule: ⟦5⟧ ⊘Q ⇐B.

Proved: ◯◯◯◯◯⊘Q.

ToProve: B.

Example 43 (Unequal loops) *Case A16*

Query: ?= ⟦2⟧ ⊘Q.

Anchored Rule: ⟦3⟧ ⊘Q ⇐B.

Proved: ⟦6⟧ ⊘Q.

ToProve: ◯◯⟦6⟧ ⊘Q,
◯◯◯◯⟦6⟧ ⊘Q, B.

Example 44 (Query loop unrolling) *Case A8*

Query: ?= ⟦3⟧ ⊘Q.

Anchored Rule: ⊘Q ⇐B.

Proved: ⊘Q.

ToProve: ◯◯◯⟦3⟧ ⊘Q, B.

Example 45 (Query loop unrolling) *Case A12*

Query: ?= ⟦3⟧ ⊘Q.

Anchored Rule: ◯◯◯⊘Q ⇐B.

Proved: ◯◯◯⊘Q.

ToProve: ⊘Q, ◯◯◯◯◯◯⟦3⟧ ⊘Q, B.

6.7 Extensions to DTLP

In this section we discuss several extensions to DTLP. The interpreter supports *cuts* exactly like prolog, and can answer existential queries.

6.7.1 Existential Queries

The expressive power of a temporal query can be extended if one can ask temporally existential questions like 'Which day did it rain?', or 'Did it rain for any amount of time on any day?' These queries can be classified into two types: *sometime* queries which ask an existential temporal question at a fixed level, and *any-time* queries which asks about an existential temporal event at any level.

Sometime and *any-time* queries are prefixed by `some` ($\Diamond$) and `any` ($\Diamond\!\!\!\!-$) respectively. These two operators do not distribute over their list arguments, since we get a weaker query on distributing '$\Diamond$' or '$\Diamond\!\!\!\!-$' over a list of goals. The operators $\Diamond$ and $\Diamond\!\!\!\!-$ must not be embedded inside other temporal operators, i.e. $[\![i]\!]$, $\bigcirc$, $\oslash$ cannot enclose $\Diamond$ or $\Diamond\!\!\!\!-$. However note that $\Diamond$ commutes with $\bigcirc$. The DTLP interpreter expands $\Diamond$ and $\Diamond\!\!\!\!-$ to a string from the sets $\{\bigcirc^i : i \geq 0\}$ and $\{\{\bigcirc, \oslash\}^i : i \geq 0\}$ respectively, until the query can be proved. In effect the interpreter utilizes the following identities:

$$\begin{aligned} \Diamond A &\leftrightarrow (A \vee \bigcirc\Diamond A) \\ \Diamond\!\!\!\!- A &\leftrightarrow (A \vee \bigcirc\Diamond\!\!\!\!- A \vee \oslash\Diamond\!\!\!\!- A) \end{aligned}$$

Such queries never fail as the interpreter keeps trying again and again with different expansions.

Example 46 (Compound sometime query) 'Was there rain on any day, followed immediately by a flood on the next day?'

```
?= ◇( ( ⊘rain, ◯⊘flood ) )
```

Example 47 (Sometime loop query) 'Did the ball fall from the table and start bouncing?'

```
?= ◇( ( ⊘fall(ball,table),
  ◯[[2]] ⊘moving(ball,down),
  ◯◯[[2]] ⊘moving(ball,up) ) ).
```

Example 48 (Query with a cut) 'Did it flood the day after the first rains?'

```
?= ◇(( ⊘rain,!,◯⊘flood )).
```

Example 49 (Any query) 'Did it rain and flood together at any time?'

```
?= ◇-(( rain, flood )).
```

6.7.2 Unanswerable Queries

Example 50 (Any query within a loop) 'Does it rain daily for any amount of time?'

```
?= [[1]] ⊘◇- rain.
```

The only way to answer this is to unroll the loop and prove the infinite sequence of sub-queries. The interpreter cannot answer this query.

6.7.3 Flexible Variables

The variables appearing in DTLP programs up to now have been rigid variables, that is, once instantiated they have a fixed value at all times. However there is need for time dependent variables, here called *flexible variables*. These variables take different values over different times.

Example 51 (Need for flexible variable) Consider the following two facts and the query:

```
⊘rain( 20 ) ⇐T.
◯rain( 35 ) ⇐T.
?= rain( X ).
```

The interpreter will break up the query and will be unable to prove it:

```
?= ⊘rain( X ), ◯rain( X ).
  Proved:  ⊘rain( 20 )
  Cannot prove:  ◯rain( 20 )
```

Example 52 'Does it rain more that 10 units daily ?' The following query expresses it incorrectly:

```
?= ⟦1⟧ ⊘rain(X), X > 10.
```

Unrolling the query we see that the variable `X` is causing the problem:

```
?= ⊘rain(X), ◯⊘rain(X), ...◯^n⊘rain(X),
  ◯^{n+1}⟦1⟧ ⊘rain(X), X > 10.
```

To remedy this situation we introduce a new set of flexible variables, denoted by the symbols `A, B,...`. We will continue using `X, Y, Z` for rigid variables. A separate ordinal tree will be used to store the values of each flexible variable over time.

Now, the query '`Q(A),R(A)`' with the temporal goals `Q(A)` and `R(A)`, over a common interval (both containing the flexible variable `A`) will be broken up as follows:

```
?= ⊘Q(A0), R(A0), ◯Q(A1), R(A1).
```

Whenever we create new flexible variables, other goals that constrain the flexible variable have to be duplicated for each instance of the flexible variable.

The loop query above with flexible variable will now be unrolled as:

```
?= ⊘rain(A0), A0 > 10,
  ◯⊘rain(A1), A1 > 10, ...
  ◯^n⊘rain(AN), AN > 10,
  ◯^{n+1}[[1]]⊘rain(A), A > 10.
```

Equating a flexible variable to a rigid variable makes the flexible variable behave like a rigid variable.

Another way of handling flexible variables would be to use a separate ordinal tree for each flexible variable to store its values over time.

6.7.4 Temporal Terms

We can get more expressive power by allowing relations between *temporal terms*, especially between adjacent values of a flexible variable within a *loop*. We can express many interesting arithmetical sequences, such as 'the height reached by the a bouncing ball in a bounce is half its previous height':

```
[[1]](⊘height(H), ◯⊘H is ((⊘H)/2)).
```

Where ◯⊘H is a temporal term indicating the value of H at the beginning of the *next* interval.

6.7.5 Clock Trees

Till now, ordinal trees divide each interval into an infinite sub-sequence of intervals. This is useful in modeling abstract computations with subroutine calls, however most hierarchies of time encountered in real life have finite division of time at each level.

In this section, we will slightly change the interpretation of ordinal trees so that at certain levels a time interval will be divisible only into a finite number of subintervals. This finiteness is indicated by an axiom schema, called *finite level axioms*, for each finite level.

Example 53 (Axiom schema) There are 60 minutes in *all* hours of *all* days:

$$\Box\oslash\Box((\oslash\bigcirc^{n}A) \leftrightarrow (\bigcirc^{n\ div\ 60}\oslash\bigcirc^{n\ mod\ 60}A)), \qquad n \geq 0$$

This is meta-temporal information, and needs to be stored separately by the interpreter. The aligning predicate now includes an input informing it of the current level, and number of preceding '◯'. When it reaches $\oslash\bigcirc^{60}A$, it rewrites it as $\bigcirc\oslash A$.

Consider the example: 'The clock interrupts the CPU every 7 minutes', with a hour containing 60 minutes. The correct way to interpret this *loop* is to *unroll* it across the hours.

6.7.6 Operator Grammars

Following Wolper [Wol83], DTLP can be extended by a deterministic-context-free grammar based operators. For example, $[\![3]\!]A$ can be written as $L_3(A)$, along with the following production rule:

$$L_3(A) ::= (A \wedge \bigcirc\bigcirc\bigcirc L_3(A))$$

for expanding the non-terminal L_3. This lets us define all regular omega-sequences of events. However it also lets us define omega tree models that do not have any ordinal tree models, as the next example shows:

Example 54 (Dense Mix) The following grammar gives an omega tree model that is not an ordinal tree.

$$\begin{aligned} L_+(A) &::= (A \wedge \oslash L_+(A) \wedge \bigcirc L_-(A)) \\ L_-(A) &::= (\neg A \wedge \oslash L_-(A) \wedge \bigcirc L_+(A)) \end{aligned}$$

In particular the model of $L_+(A)$ satisfies the formula:

$$\boxplus(\diamond\!\!\!\!\!\!\diamond A \wedge \diamond\!\!\!\!\!\!\diamond \neg A)$$

This formula claims that there is a dense mix of A and $\neg A$ in every sub-tree of the omega tree.

Chapter 7

Conclusion

In this report we showed how reasoning about time can be automated. We used the notion of refinement to impose a hierarchy of levels on time, thereby allowing the reasoner to treat time in a modular fashion at several levels. We then developed the several logics of dense time and showed how they can be decided in exponential time using tableau based methods. By embedding the temporal modalities of dense time logics in prolog, we developed a temporal logic programming language based on temporal horn clause.

We introduced Propositional Dense Time Logic (PDTL), which allows us to reason about nested sequences of events. We also gave an exponential time tableau based decision procedure for it and a complete axiomatization for it.

We next looked at an interesting subset of models called ordinal trees, where all the propositions are stable. The logic of ordinal trees called Propositional Ordinal Tree Logic (POTL) was also shown to be decidable in exponential time by extending the techniques used for PDTL. Ordinal trees are interval based as they allow only finite level of refinement, and they seem to be a good bridge between point based and interval based temporal logics.

Ordinal trees being finite can be directly embedded and used as a data structure in a logic programming language, in particular we extended prolog with ordinal trees for temporal logic programming. We defined a *temporal clause* to be an ordinal tree with its nodes labeled with prolog clauses (in a restricted way). A temporal clause is used to store temporal-facts, causal-rules and temporal-queries. We describe *temporal-resolution* based on *aligning* to answer temporal queries from temporal facts and (causal) rules. We implemented the Dense Time Logic Programming (DTLP) interpreter in prolog. This experimental interpreter can handle cuts and can also prove existential queries.

Future Work

Future work will include axiomatization of POTL and relating it to CTL*, PDL (with *converse*) and other dense time logics. We will try to define a *strong-until* operator over intervals. *Strong-until* requires operators that can access the parent node. The language of S2S allows many interesting properties to be defined at the cost of increased decision complexity. It may be possible to obtain omega tree models as a limit of ordinal tree models.

The expressive power of PDTL can be extended by allowing grammar based temporal operators. We will further investigate the expressive powers of this logic, and its use in temporal logic programming.

We will also be considering logics dealing with finitely nested finite sequences of time points, i.e. we allow only finite subsequences of time points at each level. Each level of time could correspond to some natural periodic events, like 'hours', 'minutes', and 'seconds' of a clock. For example, we could specify a day to consists of 24 hours, and make $\oslash \bigcirc^{24} \phi$ invalid or even better let $\oslash \bigcirc^{24} \phi \leftrightarrow \bigcirc \oslash \phi$. Such a representation is particularly useful in modeling synchronous digital circuits and in historical databases.

We will be implementing *clock trees* to deal with finitely nested finite sequences of time points, i.e. finite subsequences of time points at each level. Each level of time could correspond to some natural periodic events, like 'hours', 'minutes', and 'seconds' of a clock.

All the variables in DTLP are rigid, we plan to allow flexible variables as they increase the expressive power of queries. We need to re-examine the existing model theory in light of new extensions and applications.

Bibliography

[AFMO85] T. Aoyagi, M. Fujita, and T. Moto-Oka. Temporal logic programming language *tokio*: programming in *tokio*. In *Logic Programming 85*, pages 128–137, 1985. LNCS 221.

[AH90] R. Alur and T. Henzinger. Real-time logics: Complexity and expressiveness. In *Logic in Computer Science*, pages 390–401, 1990.

[Ahm93] Mohsin Ahmed. Concurrent temporal logic programming: Based on guarded temporal horn clauses. In *IEEE Kharagpur section technical conference on Applications of Parallel and Distributed Processing, Kharagpur, India*, January 1993.

[All84] J. F. Allen. Towards a general theory of action and time. *Artificial Intelligence*, 23(2):123–154, 1984.

[AM87] M. Abadi and Z. Manna. Temporal logic programming. In *International Symposium on Logic Programming San Francisco, CA*, pages 4–16, 1987.

[AV93a] Mohsin Ahmed and G. Venkatesh. Dense time logic programming. In *Second Symposium on Logical Formalizations of Common-sense Reasoning, Austin, TX*, January 1993.

[AV93b] Mohsin Ahmed and G. Venkatesh. A propositional dense time logic: Based on nested sequences. In *TAPSOFT (CAAP), Orsay, France*, April 1993.

[AV93c] Mohsin Ahmed and G. Venkatesh. Reasoning about motion: Product theories of motion. In *IEEE TENCON, Beijing, China*, October 1993.

[BAPM83] M. Ben-Ari, A. Pnueli, and Z. Manna. The temporal logic of branching time. *Acta Informatica*, 20:207–226, 1983.

[Bar77] J. Barwise. Introduction to first order logic. In J. Barwise, editor, *Handbook of Mathematical Logic*, pages 5–46. North Holland, Amsterdam, 1977.

[Bar87] H. Barringer. The use of temporal logic in compositional specification of concurrent systems. In A. Galton, editor, *Temporal Logics and their applications*, pages 53–90. Academic Press, 1987.

[Bau89] M. Baudinet. Temporal logic programming is complete and expressive. In *16th POPL, Austin, TX*, pages 267–280, 1989.

[BKP86] H. Barringer, R. Kuiper, and A. Pnueli. A really abstract concurrent model and its temporal logic. In *13th POPL, St. Petersberg Beach, FL*, pages 173–183, 1986.

[Boo79] George Boolos. *The Unprovability of Consistency*. Cambridge University Press, Cambridge, 1979.

[BS84] Robert Bull and Krister Segerberg. Basic modal logic. In D. Gabbay and F. Guenther, editors, *Handbook of Philosophical Logic*, volume II, pages 1–88. D. Reidel, Dordrecht, Holland, 1984.

[Bur84] J. P. Burgess. Basic tense logic. In D. Gabbay and F. Guenther, editors, *Handbook of Philosophical Logic*, volume II, pages 89–133. D. Reidel, Dordrecht, Holland, 1984.

[Che80] Brian Chellas. *Modal Logic - An Introduction*. Cambridge University Press, Cambridge, 1980.

[CM84] W. F. Clocksin and C. S. Mellish. *Programming in Prolog*. Springer Verlag, 2 edition, 1984.

[Eme90] E. Allen Emerson. Temporal and modal logic. In J. v. Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 996–1072. Elsevier, 1990.

[End72] H. B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, 1972.

[FKTMO85] M. Fujita, S. Kono, H. Tanaka, and T. Moto-Oka. Tokio: logic programming language based on temporal logic and its compilation to prolog. In *3rd Int. Conf. on Logic Programming*, pages 695–709. Springer Verlag, 1985. LNCS 225.

[Gab73] D. Gabbay. A survey of decidability results for modal, tense and intermediate logics. In Suppes et. al., editor, *Logic, Methodologies and Philosophy of Science IV*, pages 29–43. North Holland, Amsterdam, 1973.

[Gab75] D. Gabbay. Decidability results in non-classical logics - i. *Annals of Mathematical Logic*, 8:237–295, 1975.

[Gab87] D. Gabbay. Modal and temporal logic programming. In A. Galton, editor, *Temporal Logics and their applications*, pages 195–237. Academic Press, 1987.

[Gab91] D. Gabbay. Modal and temporal logic programming - ii. In T. Dodd, editor, *Logic Programming*, pages 82–123. Intellect, Oxford, 1991.

[Gal87a] J. Gallier. *Logic for Computer Science*. John Wiley, Singapore, 1987.

[Gal87b] A. Galton. Temporal logic and computer science: An overview. In A. Galton, editor, *Temporal Logics and their applications*, pages 1–52. Academic Press, 1987.

[GPSS80] D. Gabbay, A. Pnueli, S. Shelah, and S. Stavi. The temporal analysis of fairness. In *7th POPL, Las Vegas, NE*, pages 163–173, 1980.

[Hal87] R. Hale. Temporal logic programming. In A. Galton, editor, *Temporal logics and their applications*, pages 91–119. Academic Press, 1987.

[Ham71] C. L. Hamblin. Instants and intervals. *Studium Generale*, 24:127–134, 1971.

[Har84] D. Harel. Propositional dynamic logic. In D. Gabbay and F. Guenther, editors, *Handbook of Philosophical Logic*, volume II, pages 497–604. D. Reidel, Dordrecht, Holland, 1984.

[KS86] R. A. Kowalski and M. J. Sergot. A logic-based calculus of events. *New Generation Computing*, 4:67–95, 1986.

[KT90] D. Kozen and J. Tiuryn. Logics of programs. In J. V. Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 165–186. Elsevier, 1990.

[Lam90] L. Lamport. A temporal logic of actions. TR 57, Digital, 1990.

[Llo84] J. W. Lloyd. *Foundations of Logic Programming*. Springer Verlag, 1984.

[LPZ85] O. Lichtenstein, A. Pnueli, and L. Zuck. The glory of the past. In *Proc. Conf. Logics of Programs*, pages 196–218. Springer Verlag, 1985. LNCS 193.

[McD82] D. McDermott. A temporal logic for reasoning about processes and plan. *Cognitive Science*, 6:101–155, 1982.

[Mos86] B. C. Moszkowski. *Executing Temporal Logic Programs*. Cambridge University Press, Cambridge, 1986.

[Par81] R. Parikh. Propositional dynamic logics of programs: A survey. *LNCS 125*, pages 102–144, 1981. LNCS 125.

[Pri67] A. N. Prior. *Past, present and future*. Oxford: Claredon Press, 1967.

[Rab69] Michael O. Rabin. Decidability of second-order theories and automata on infinite trees. *Transactions of AMS*, 141:1–35, July 1969.

[Rab77] Michael O. Rabin. Decidable theories. In J. Barwise, editor, *Handbook of Mathematical Logic*, pages 595–630. North Holland, Amsterdam, 1977.

[Ram89] Allan Ramsay. *Formal Methods in Artificial Intelligence*. Cambridge University Press, Cambridge, 1989.

[RDMMS92] Y. S. Ramakrishna, L. K. Dillion, L. E. Moser, and P. M. Melliar-Smith. An automata-theoretic decision procedure for future interval logic. In *12th FSTTCS, New Delhi, INDIA*, pages 51–67. Springer Verlag, 1992. LNCS 652.

[San93] Erik Sandewall. Causal qualification and structure-based ramification. In *Second Symposium on Logical Formalizations of Common-sense Reasoning, Austin, TX*, January 1993.

[Sar87] F. Sardi. Three recent approaches to temporal reasoning. In A. Galton, editor, *Temporal Logics and their applications*, pages 121–168. Academic Press, 1987.

[SC85] A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. *Journal of ACM*, 3(32):733–749, July 1985.

[SH88] A. Szalas and L. Holenderski. Incompleteness of first order temporal logic with *until*. *Theoretical Computer Science*, 57, 1988.

[Sho89] Y. Shoham. *Reasoning about change*. MIT Press, 1989.

[Sow84] J. F. Sowa. *Conceptual Structures*. Addision Wesley, 1984.

[Sza87a] A. Szalas. Arithmetical axiomatization of first order temporal logic. *Information Processing Letters*, 26, 1987.

[Sza87b] A. Szalas. A complete axiomatic characterization of first order temporal logic of linear time. *Theoretical Computer Science*, 54, 1987.

[Tho90] W. Thomas. Automata on infinite trees. In J. V. Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 165–191. Elsevier, 1990.

[vB83] J. F. A. K. van Benthem. *The Logic of Time*. D. Reidel, Dordrecht, Holland, 1983.

[vB89] J. F. A. K. van Benthem. Time, logic and computation. In J. W. deBakker, W.-P. deRoever, and G. Rozenberg, editors, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, pages 1–49. Springer Verlag, 1989. LNCS 354.

[Wol83] P. Wolper. Temporal logic can be more expressive. *Information and Control*, 56:72–99, 1983.

Appendix A

Testing the DTLP Interpreter

A.1 Test Programs

A.1.1 Program 1

```
/*

 DTLP fact/rule/query test, file t1, 05-Oct-92 */
clear.
/* ========== Facts */
within b <= true.
next within d <= true.
next next next within y <= true.

/* ========== Anchored rules */
next within a <=
          within b,
          next next within c,
          next next next within x.

/* ========== Level-1 rule */
all( next within c <= within d ).

/* ========== Universal rule */
every( x <= y ).

/* ========== Queries for testing */

/* Query-0, test facts */
within b, within next b, within loop(1, b).

/* Query-1, test universal rule */
next next next within x.

/* Query-2, test level rule */
next next within c.

/* Query-3, test anchored/level/universal rules */
next within a.

/* Query-4, Variable = Goal, and execute modified Variable */
```

```
X = within d,
next X.

/*

*/
```

A.1.2 Program 2

```
/*

 DTLP backtracking test, file t2, 05-Oct-92 */
clear.
every( a <= b ).
within b <= true.
next within a <=
          next next within c,
          within d.
next next c <= true.
within within d <= true.
within next d <= true.
all( next a <= within loop(6,within e ) ).
next within loop( 3, within f ) <= true.
every( e <= f ).

/* Query "a" is proveable */
a.

/*

*/
```

A.1.3 Program 3

```
/*

 DTLP loop test, file t3, 07-Oct-92 */
clear.
/* ========== facts */
a <= true.
loop( 2, within b) <= true.
loop( 3, within c) <= true.
loop( 5, within d) <= true.
loop( 2, within e) <= true.
next loop( 2, within e) <= true.

/* Q0 */
loop(2, within a ).

/* Q1 */
loop(6, within (b,c) ).

/* Q2 */
loop(2, within (b,c) ).

/* Q3 */
```

```
loop( 1, within e).

/* Q4 */
a,e.

/*

*/
```

A.1.4 Program 4

```
/*

 DTLP cut test, file t4, 07-Oct-92 */
clear.
/* === Facts */
within a(0) <= !, fail.
a(X) <= true.

/* Query escapes !,fail and succeeds */
next a(0).
/* Query fails because of !,fail */
a(0).
/* Query escapes !,fail and succeeds */
within a(1).

every( b(X) <= c(X) ).
within c(1) <= !, fail.
c(2) <= true.

/* Query fails, because of choice of c(1) is fixed by ! */
b(1).
/* Query succeeds, as it picks only c(2) */
b(2).

/*

*/
```

A.1.5 Program 5

```
/*

 DTLP any/some test, file t5, 14-Oct-92 */
/* Part 1 */
clear.
/* === Facts */
within a(5) <= true.
within next a(4) <= true.
within b(4) <= true.

/* Any Query: simple */
any( b(4) ).

/* Any Query: two simple */
any( a(X) ), any( b(Y) ).
```

```
/* Compound Any Query with backtracking */
any( ( a(X), b(X) ) ).

/* Part 2 */
clear.
next next within d <= true.
next next next next within b <= true.

/* Simple Some queries */
some within d.
/*

*/
```

A.2 Trace of Test Programs

A.2.1 Trace 1

```
+-------------------------------------------+
| MS-DOS Prolog-1             Version 2.2   |
| Copyright 1983      Serial number: 0001040 |
| Expert Systems Ltd.                       |
| Oxford U.K.                               |
+-------------------------------------------+

?- [dtlp].
..... Dense Time Logic Interpreter .....
..... Loading ...
Commands are:
   help / exit / clear / show / [file] /
   every(..) / all(..) / head <= body / query
dtlp consulted.
?- dtlp.
?= [t1].
Input: .(t1,[])
?= Input: clear
Output: clear
/* ==================== */
?= Input: <=(within(b),true)
Temporal Fact: within b<=true
Output: asserted
/* ==================== */
?= Input: <=(next(within(d)),true)
Temporal Fact: next within d<=true
Output: asserted
/* ==================== */
?= Input: <=(next(next(next(within(y)))),true)
Temporal Fact: next next next within y<=true
Output: asserted
/* ==================== */
?= Input: <=(next(within(a)),,(within(b),,(next(next(within(c))),
        next(next(next(within(x)))))))
Anchored Rule: next within a
        <=within b,next next within c,
        next next next within x
Output: asserted
```

```
/* ==================== */
?= Input: all(<=(next(within(c)),within(d)))
Level Rule: all next within c<=within d
Output: asserted
/* ==================== */
?= Input: every(<=(x,y))
Universal Rule: every x<=y
Output: asserted
/* ==================== */
?= Input: ,(within(b),,(within(next(b)),within(loop(1,b))))
 anchored aligning: within b<=true+(within next b,within loop(1,b))
 anchored aligning: within next b<=true+within loop(1,b)
 anchored aligning: within loop(1,b)<=true
Output: proved
/* ==================== */
?= Input: next(next(next(within(x))))
 universal aligning: next next next within x<=next next next within y
 anchored aligning: next next next within y<=true
Output: proved
/* ==================== */
?= Input: next(next(within(c)))
 level aligning: next next within c<=next within d
 anchored aligning: next within d<=true
Output: proved
/* ==================== */
?= Input: next(within(a))
 anchored aligning: next within a
        <=within b,next next within c,next next next within x
 anchored aligning: within b
        <=true+(next next within c,next next next within x)
 level aligning: next next within c
        <=next within d+next next next within x
 anchored aligning: next within d<=true
 universal aligning: next next next within x<=next next next within y
 anchored aligning: next next next within y<=true
Output: proved
/* ==================== */
?= Input: ,(=(_80,within(d)),next(_80))
 Prolog proved head of: (within d=within d)+next within d
 anchored aligning: next within d<=true
Output: proved
/* ==================== */
?= Input: end_of_file
Output: exit
/* ==================== */
Output: read
/* ==================== */
?= ^Z
Input: end_of_file
Output: exit
/* ==================== */

yes
?- ^Z
```

A.2.2 Trace 2

```
?- [dtlp].
```

```
..... Dense Time Logic Interpreter .....
..... Loading ...
Commands are:
   help / exit / clear / show / [file] /
   every(..) / all(..) / head <= body / query
dtlp consulted.
?- dtlp.
?= [t2].
Input: .(t2,[])
?= Input: clear
Output: clear
/* ====================== */
?= Input: every(<=(a,b))
Universal Rule: every a<=b
Output: asserted
/* ====================== */
?= Input: <=(within(b),true)
Temporal Fact: within b<=true
Output: asserted
/* ====================== */
?= Input: <=(next(within(a)),,(next(next(within(c))),within(d)))
Anchored Rule: next within a<=next next within c,within d
Output: asserted
/* ====================== */
?= Input: <=(next(next(c)),true)
Temporal Fact: next next c<=true
Output: asserted
/* ====================== */
?= Input: <=(within(within(d)),true)
Temporal Fact: within within d<=true
Output: asserted
/* ====================== */
?= Input: <=(within(next(d)),true)
Temporal Fact: within next d<=true
Output: asserted
/* ====================== */
?= Input: all(<=(next(a),within(loop(6,within(e)))))
Level Rule: all next a<=within loop(6,within e)
Output: asserted
/* ====================== */
?= Input: <=(next(within(loop(3,within(f)))),true)
Temporal Fact: next within loop(3,within f)<=true
Output: asserted
/* ====================== */
?= Input: every(<=(e,f))
Universal Rule: every e<=f
Output: asserted
/* ====================== */
?= Input: a
 universal aligning: a<=b
 anchored aligning: within b<=next b
 Failed and backtracking on: next b
 Failed and backtracking on: b
 anchored aligning: next within a
        <=within a,next next a,next next within c,within d
 universal aligning: within a
        <=within b+(next next a,next next within c,within d)
 anchored aligning: within b<=true
 universal aligning: next next a
```

```
       <=next next b+(next next within c,within d)
 Failed and backtracking on: next next b
 level aligning: next next a
       <=next within loop(6,within e)+(next next within c,within d)
 universal aligning: next within loop(6,within e)
       <=next within loop(6,within f)
 anchored aligning: next within loop(6,within f)<=true
 anchored aligning: next next within c<=true+within d
 anchored aligning: within within d<=within next d
 anchored aligning: within next d<=true
Output: proved
/* ==================== */
?= Input: end_of_file
Output: exit
/* ==================== */
Output: read
/* ==================== */
?= ^Z
Input: end_of_file
Output: exit
/* ==================== */

yes
?- ^Z
```

A.2.3 Trace 3

```
?- [dtlp].
..... Dense Time Logic Interpreter .....
..... Loading ...
Commands are:
   help / exit / clear / show / [file] /
   every(..) / all(..) / head <= body / query
dtlp consulted.
?- dtlp.
?= [t3].
Input: .(t3,[])
?= Input: clear
Output: clear
/* ==================== */
?= Input: <=(a,true)
Temporal Fact: a<=true
Output: asserted
/* ==================== */
?= Input: <=(loop(2,within(b)),true)
Temporal Fact: loop(2,within b)<=true
Output: asserted
/* ==================== */
?= Input: <=(loop(3,within(c)),true)
Temporal Fact: loop(3,within c)<=true
Output: asserted
/* ==================== */
?= Input: <=(loop(5,within(d)),true)
Temporal Fact: loop(5,within d)<=true
Output: asserted
/* ==================== */
?= Input: <=(loop(2,within(e)),true)
Temporal Fact: loop(2,within e)<=true
```

```
Output: asserted
/* ===================== */
?= Input: <=(next(loop(2,within(e))),true)
Temporal Fact: next loop(2,within e)<=true
Output: asserted
/* ===================== */
?= Input: loop(2,within(a))
 anchored aligning: loop(2,within a)<=true
Output: proved
/* ===================== */
?= Input: loop(6,within(,(b,c)))
 anchored aligning: loop(6,within b)<=true+loop(6,within c)
 anchored aligning: loop(6,within c)<=true
Output: proved
/* ===================== */
?= Input: loop(2,within(,(b,c)))
 anchored aligning: loop(2,within b)<=true+loop(2,within c)
 anchored aligning: loop(6,within c)
        <=next next loop(6,within c),
        next next next next loop(6,within c)
 Failed and backtracking on: next next loop(6,within c)
        +next next next next loop(6,within c)
 Failed and backtracking on: loop(2,within c)
 Failed and backtracking on: loop(2,within b)+loop(2,within c)
Output: failed
/* ===================== */
?= Input: loop(1,within(e))
 anchored aligning: loop(2,within e)<=next loop(2,within e)
 anchored aligning: next loop(2,within e)<=true
Output: proved
/* ===================== */
?= Input: ,(a,e)
 anchored aligning: a<=true+e
 anchored aligning: loop(2,within e)<=next loop(2,within e)
 anchored aligning: next loop(2,within e)<=true
Output: proved
/* ===================== */
?= Input: end_of_file
Output: exit
/* ===================== */
Output: read
/* ===================== */
?= show.
Input: show
 Temporal_Fact  : <=(a,true)
 Temporal_Fact  : <=(loop(2,within(b)),true)
 Temporal_Fact  : <=(loop(3,within(c)),true)
 Temporal_Fact  : <=(loop(5,within(d)),true)
 Temporal_Fact  : <=(loop(2,within(e)),true)
 Temporal_Fact  : <=(next(loop(2,within(e))),true)
Output: shown
/* ===================== */
?= ^Z
Input: end_of_file
Output: exit
/* ===================== */

yes
?- ^Z
```

A.2.4 Trace 4

```
?- [dtlp].
..... Dense Time Logic Interpreter .....
..... Loading ...
Commands are:
   help / exit / clear / show / [file] /
   every(..) / all(..) / head <= body / query
dtlp consulted.
?- dtlp.
?= [t4].
Input: .(t4,[])
?= Input: clear
Output: clear
/* ==================== */
?= Input: <=(within(a(0)),,(!,fail))
Anchored Rule: within a(0)<=!,fail
Output: asserted
/* ==================== */
?= Input: <=(a(_76),true)
Temporal Fact: a(_76)<=true
Output: asserted
/* ==================== */
?= Input: next(a(0))
 anchored aligning: next a(0)<=true
Output: proved
/* ==================== */
?= Input: a(0)
 anchored aligning: within a(0)<=next a(0),!,fail
 anchored aligning: next a(0)<=true+(!,fail)
 Cut and continuing on: !+fail
 Failed and backtracking on: fail
Output: failed
/* ==================== */
?= Input: within(a(1))
 anchored aligning: within a(1)<=true
Output: proved
/* ==================== */
?= Input: every(<=(b(_79),c(_79)))
Universal Rule: every b(_79)<=c(_79)
Output: asserted
/* ==================== */
?= Input: <=(within(c(1)),,(!,fail))
Anchored Rule: within c(1)<=!,fail
Output: asserted
/* ==================== */
?= Input: <=(c(2),true)
Temporal Fact: c(2)<=true
Output: asserted
/* ==================== */
?= Input: b(1)
 universal aligning: b(1)<=c(1)
 anchored aligning: within c(1)<=next c(1),!,fail
 Failed and backtracking on: next c(1)+(!,fail)
 Failed and backtracking on: c(1)
 Failed and backtracking on: b(1)
Output: failed
/* ==================== */
?= Input: b(2)
```

```
 universal aligning: b(2)<=c(2)
 anchored aligning: c(2)<=true
Output: proved
/* ==================== */
?= Input: end_of_file
Output: exit
/* ==================== */
Output: read
/* ==================== */
?= ^Z
Input: end_of_file
Output: exit
/* ==================== */

yes
?- ^Z
```

A.2.5 Trace 5

```
?- [dtlp].
..... Dense Time Logic Interpreter .....
..... Loading ...
Commands are:
   help / exit / clear / show / [file] /
   every(..) / all(..) / head <= body / query
dtlp consulted.
?- dtlp.
?= [t5].
Input: .(t5,[])
?= Input: clear
Output: clear
/* ==================== */
?= Input: <=(within(a(5)),true)
Temporal Fact: within a(5)<=true
Output: asserted
/* ==================== */
?= Input: <=(within(next(a(4))),true)
Temporal Fact: within next a(4)<=true
Output: asserted
/* ==================== */
?= Input: <=(within(b(4)),true)
Temporal Fact: within b(4)<=true
Output: asserted
/* ==================== */
?= Input: any(b(4))
 anchored aligning: within b(4)<=next b(4)
 Failed and backtracking on: next b(4)
 Failed and backtracking on: b(4)
 Trying: any within b(4)
 anchored aligning: within b(4)<=true
Output: proved
/* ==================== */
?= Input: ,(any(a(_77)),any(b(_80)))
 anchored aligning: within a(5)<=next a(5)
 Failed and backtracking on: next a(5)
 anchored aligning: within next a(4)<=next a(4),within within a(4)
 Failed and backtracking on: next a(4)+within within a(4)
 Failed and backtracking on: a(_77)
```

```
 Trying: (any within a(_77))+any b(_80)
 anchored aligning: within a(5)<=true
 anchored aligning: within b(4)<=next b(4)
 Failed and backtracking on: next b(4)
 Failed and backtracking on: b(_80)
 Trying: any within b(_80)
 anchored aligning: within b(4)<=true
Output: proved
/* ==================== */
?= Input: any(,(a(_79),b(_79)))
 anchored aligning: within a(5)<=next a(5)+b(5)
 Failed and backtracking on: next a(5)
 anchored aligning: within next a(4)
        <=(next a(4),within within a(4))+b(4)
 Failed and backtracking on: next a(4)+within within a(4)
 Failed and backtracking on: a(_79)+b(_79)
 Trying: any within(a(_79),b(_79))
 anchored aligning: within a(5)<=true+within b(5)
 Failed and backtracking on: within b(5)
 anchored aligning: within next a(4)<=within within a(4)+within b(4)
 Failed and backtracking on: within within a(4)
 Failed and backtracking on: within a(_79)+within b(_79)
 Trying: any next(a(_79),b(_79))
 Failed and backtracking on: next a(_79)+next b(_79)
 Trying: any within within(a(_79),b(_79))
 anchored aligning: within within a(5)<=true+within within b(5)
 Failed and backtracking on: within within b(5)
 Failed and backtracking on: within within a(_79)+within within b(_79)
 Trying: any next within(a(_79),b(_79))
 Failed and backtracking on: next within a(_79)+next within b(_79)
 Trying: any within next(a(_79),b(_79))
 anchored aligning: within next a(5)<=true+within next b(5)
 Failed and backtracking on: within next b(5)
 anchored aligning: within next a(4)<=true+within next b(4)
 anchored aligning: within next b(4)<=true
Output: proved
/* ==================== */
?= Input: clear
Output: clear
/* ==================== */
?= Input: <=(next(next(within(d))),true)
Temporal Fact: next next within d<=true
Output: asserted
/* ==================== */
?= Input: <=(next(next(next(next(within(b))))),true)
Temporal Fact: next next next next within b<=true
Output: asserted
/* ==================== */
?= Input: some(within(d))
 Failed and backtracking on: within d
 Trying: some next within d
 Failed and backtracking on: next within d
 Trying: some next next within d
 anchored aligning: next next within d<=true
Output: proved
/* ==================== */
?= Input: end_of_file
Output: exit
/* ==================== */
```

```
Output: read
/* ===================== */
?= ^Z
Input: end_of_file
Output: exit
/* ===================== */

yes
?- ^Z
```

Appendix B

DTLP Interpreter Listing

```
/*

 * ================================================
 * Dense Time Logic Programming, prolog86/PC,
 * 24-Oct-92, 17-Mar-93
 * Mosh@cse.iitb.ernet.in
 * ================================================
 */
?-        write('..... Dense Time Logic Interpreter ..... '), nl,
          write('..... Loading ...'), nl,

          op( 255, fx, every ),
          op( 255, fx, all ),
          op( 255, fx, some ),
          op( 255, fx, any   ),
          op( 254, xfy, <= ),
          /* op( 253, xfy, ',' ), */
          op( 20, fx, next ),
          op( 20, fx, within ).
/* Now: ( every next a <= b, c ) == every(next(a)<=(b,c)) */
/* ================================================
 * @dtlp topcall read-eval loop
 */
dtlp:-
          repeat,
          write('?= '), read( Input ),
          write('Input: '), display( Input ), nl,
          dtlp( Input, Output ),
          write('Output: '), display( Output ), nl,
          write('/* ==================== */'), nl,
          Output = exit.
/* ================================================ */
/* @dtlp( Rule/Query:i, Output:o:true/false/asserted ) */
dtlp( help, help ):-!,
          write('Commands are:'), nl,
          write('   help / exit / clear / show / [file] / '), nl,
          write('   every(..) / all(..) / head <= body / query '), nl.
dtlp( exit, exit ):-!.
dtlp( end_of_file, exit ):-!.
dtlp( clear, clear ):-!, retractall( ruledata(_,_) ).
```

```
dtlp( show, retry ):-
          ruledata( RuleType, Data ),
          write(' '), write( RuleType ), write(' : '),
          display( Data ), nl,
          fail.
dtlp( show, shown ):-!.

dtlp( [FileName], read ):-
          see( FileName ), dtlp, seen, !.
dtlp( [FileName], file_error ):-!.

/*
 * @ruledata( RuleType:
 *  {Universal_Rule,Level_Rule,Temporal_Fact,Anchored_Rule}:i,
 *  Rule/Data:{every(H<=B),all(R),H<=B}:i )
 *
 *  Used for Storing Facts and Rules.
 */
dtlp( every(Rule), asserted ):-!,
          write('Universal Rule: '), write( every(Rule)), nl,
          assertz( ruledata( 'Universal_Rule', every(Rule) ) ).
dtlp( all(Rule), asserted ):-!,
          write('Level Rule: '), write( all(Rule) ), nl,
          assertz( ruledata( 'Level_Rule    ', all(Rule) ) ).
dtlp( Head<=true, asserted ):-!,
          write('Temporal Fact: '), write( Head<=true ), nl,
          assertz( ruledata( 'Temporal_Fact ', Head<=true ) ).
dtlp( Head<=Body, asserted ):-!,
          write('Anchored Rule: '), write( Head<=Body ), nl,
          assertz( ruledata( 'Anchored_Rule ', Head<=Body ) ).
/*
 * @anyplus( s(A):i, s'(A):o ), where s, s' in {within,next}^*,
 *  and s' is enumerated after s, such that:
 *  value( within ) = 0, value( next ) = 1, and
 *  value( s ) = binaryvalue( reverse( s ) )
 */
anyplus( within(A), next(A) ) :-!.
anyplus( next(A), within(B) ) :-!, anyplus( A, B ).  /* carry */
anyplus( A, within(A) ).
/* ================================================== */
/*
 * @prolog( Temporally_Goal_For_Prolog_Call:i )
 *  succeeds, if a prolog call can prove it.
 */
prolog( within(Goal) ):-!, prolog( Goal ).
prolog( next(Goal)   ):-!, prolog( Goal ).
prolog( loop(_,Goal) ):-!, prolog( Goal ).
prolog( true ):- !.                /* var(Goal), Goal = true */
prolog( Goal ):- integer( Goal ),!, fail.
prolog( Goal ):- real( Goal ),!, fail.
prolog( Goal ):- call( Goal ).
/* ================================================== */
/*
 * @normalize: normalize/flatten/distribute operators before aligning.
 *  Always succeeds. Leave rigid variables alone.
 * normalize( New:i, Old:i, FlatSeqOfTerms:o ) until New =Old.
*/
normalize( A, A, A ):-!.
normalize( X, _, X ):- var(X), !.
```

```
normalize( within(A), _, within(A) ):- var( A ), !.
normalize( within(true), _, true ):-!.
normalize( within((A,B)), _, D ):-!,
          normalize( (within(A),within(B)), 0, D ).
normalize( within(A), _, D ):-!,
          normalize( A, 0, C ),
          normalize( within(C), within(A), D).

normalize( next(A), _, next(A) ):- var( A ), !.
normalize( next(true), _, true ):-!.
normalize( next((A,B)), _, D ):-!,
          normalize( (next(A),next(B)), 0, D ).
normalize( next(A), _, D ):-!,
          normalize( A, 0, C ),
          normalize( next(C), next(A), D).

normalize( loop(I,A), _, loop(I,A) ):- var( A ), !.
normalize( loop(_,true), _, true ):-!.
normalize( loop(I,(A,B)), _, D ):-!,
          normalize( (loop(I,(A)),loop(I,(B))), 0, D ).
normalize( loop(I,next(A)), _, D ):-!,
          normalize( next(loop(I,(A))), 0, D ).
normalize( loop(I,A), _, D ):-!,
          normalize( A, 0, C ),
          normalize( loop(I,C), loop(I,A), D).

normalize( (X,Y), _, (X,Y) ):- var( X ), var( Y ), !.
normalize( (X,B), _, (X,D) ):- var( X ), !, normalize( B, 0, D ).
normalize( (A,Y), _, (C,Y) ):- var( Y ), !, normalize( A, 0, C ).

normalize( (A,true), _, D ):-!, normalize( A, 0, D ).
normalize( (true,B), _, D ):-!, normalize( B, 0, D ).

normalize( ((A,B),C), _, D ):-!, normalize( (A,(B,C)), 0, D ).

normalize( (A,B), _, D ):-!,
          normalize( A, 0, A1 ),
          normalize( B, 0, B1 ),
          normalize( (A1,B1), (A,B), D ).

normalize( A, _, A ).

/*
 *  Aligning
 *  @align( AlignType:o, Goal:i, Rule:i, ToProve:o, Proved:o )
 */
align( universal, G, every(UnivRule), ToProve, Proved ):-!,
          universal( G, UnivRule, ToProve, Proved ).
align( level, G, all(LevelRule), ToProve, Proved ):-!,
          level( G, all(LevelRule), ToProve, Proved ).
align( anchored, G, (H<=B), (MovingQ,AnchoredQ), Proved ):-
          anchor( G,(H<=B), AnchoredQ, MovingQ, Proved ).
/*
 * Universal Rules: cases E1,E2,E3,E4
 * @universal( Goal:i, UnivRule:i, MovingQuery:o, Proved:o )
 */
universal( within(G), UnivRule, within(B), within(Proved) ):-
          universal( G, UnivRule, B, Proved).
```

```
universal( next(G), UnivRule, next(B), next(Proved) ):-
          universal( G, UnivRule, B, Proved).
universal( loop(I,G), UnivRule, loop(I,B), loop(I,Proved) ):-
          universal( G, UnivRule, B, Proved).
universal( G, ( G <= B ), B, G ):-!.
/*
 * Level Rules
 * @level( Goal:i, LevelRule:i, MovingQuery:o, Proved:o )
 */
/* === case L2, get rid of 'all' */
level( within(G), all(LevelRule), Mq, Proved ):-!,
          level( within(G), LevelRule, Mq, Proved ).
/* === case L3 */
level( next(G), all(next(H)<=B), Mq, next(Proved) ):-!,
          level( G, all(H<=B), Mq, Proved ).
level( next(G), all(LevelRule), next(Mq), next(Proved) ):-!,
          level( G, all(LevelRule), Mq, Proved ).
/* === case L4 */
level( loop(I,G), all(LevelRule), loop(I,Mq), loop(I,Proved) ):-!,
          level( G, LevelRule, Mq, Proved ).
/* === case L6 */
level( within(G), within(LevelRule), within(Mq), within(Proved) ):-!,
          level( G, LevelRule, Mq, Proved ).
/* === cases L7,L8,L5 donot occur */
/* === case L1, Q-catchall, keep testing */
level( Q, all(LevelRule), Mq, Proved ):-
          level( Q, LevelRule, Mq, Proved ).
        /* if above case does not work then try looping */
level( Q, all(LevelRule), loop(1,Mq), loop(1,Proved) ):-!,
          level( within(Q), LevelRule, Mq, Proved ).
/* === BaseLevelRule-catchall */
level( Q, ( H <= B ), ( Aq, Mq ), Proved ):-!,
          anchor( Q, (H <= B ), Aq, Mq, Proved ).
/*
 * Anchored Rules
 * @anchor( Query:i,AnchoredRule:i, AnchoredQuery:o,
 *          MovingQuery:o, Proved:o )
 */
/* === case A6 */
anchor( within(Q), (within(H) <= B), Aq, within(Mq), within(Pd) ):-!,
          anchor( Q, (H <= B), Aq, Mq, Pd ).
/* === case A7, A10 */
anchor( next(_), (within(_) <= _), _, _, _ ):-!, fail.
anchor( within(_), (next(_) <= _), _, _, _ ):-!, fail.
/* === case A11 */
anchor( next(Q), (next(H) <= B), Aq, next(Mq), next(Pd) ):-!,
          anchor( Q, (H <= B), Aq, Mq, Pd ).
/* === case A8 */
anchor( loop(I,Q), (within(H) <= B), Aq, (Mq,UnrollQ), Pd ):-!,
          anchor( Q, (within(H) <= B), Aq, Mq, Pd ),
          unroll( I, loop(I,Q), UnrollQ ).
/* === case A12 */
anchor( loop(I,Q), (next(H) <= B), Aq, (Q,Mq), Pd ):-!,
          unrollhalt1( loop(I,Q), next(H) ),
          unroll( I, loop(I,Q), UnrollQ ),
          anchor( UnrollQ, (next(H) <= B), Aq, Mq, Pd ).
/* === case A14 */
anchor( within(Q), (loop(J,H) <= B), Aq, Mq, Pd ):-!,
          anchor( within(Q), (H <= B), Aq, Mq, Pd ).
```

```
/* === case A15 */
anchor( next(Q), (loop(J,H) <= B), Aq, Mq, Pd ):-!,
          unrollhalt1( next(Q), loop(J,H) ),
          unroll( J, loop(J,H), UnrollQ ),
          anchor( next(Q), (UnrollQ <= B), Aq, Mq, Pd ).
/* === case A16, two cases: a:I = J, b:I <> J */
anchor( loop(I,Q), (loop(I,H) <= B), Aq, loop(I,Mq), loop(I,Pd) ):-!,
          anchor( Q, (H <= B), Aq, Mq, Pd ).
anchor( loop(I,Q), (loop(J,H) <= B), Aq, (loop(Lcm,Mq),Qlcm), loop(Lcm,Pd)):-
          I \= J, !,
          anchor( Q, (H <= B), Aq, Mq, Pd ),
          anchorlcm( Q, I, J, I, J, Qlcm, Lcm ).
/* === case A2,  H catchall */
anchor( within(Q), (H <= B), Aq, within(Mq), within(Pd) ):-!,
          anchor( Q, (H <= B), Aq, Mq, Pd ).
/* === case A3,  H catchall */
anchor( next(Q), (H <= B), Aq, next(Mq), next(Pd) ):-!,
          anchor( Q, (H <= B), Aq, Mq, Pd ).
/* === case A4,  H catchall */
anchor( loop(I,Q), (H <= B), Aq, loop(I,Mq), loop(I,Pd) ):-!,
          anchor( Q, (H <= B), Aq, Mq, Pd ).
/* === case A5,  Q catchall */
anchor( Q, (within(H) <= B), Aq, (next(Q),within(Mq)), within(Pd) ):-!,
          anchor( Q, (H <= B), Aq, Mq, Pd ).
/* === case A9,  Q catchall */
anchor( Q, (next(H) <= B), Aq, (within(Q),next(Mq)), next(Pd) ):-!,
          anchor( Q, (H <= B), Aq, Mq, Pd ).
/* === case A13, Q catchall */
anchor( Q, (loop(J,H) <= B), Aq, (loop(J,Mq),S), loop(J,Pd) ):-!,
          anchor( within(Q), (H <= B), Aq, Mq, Pd ),
          anchorcase13( J, loop(J,within(Q)), S ).
/* === case A1,  Q catchall, H catchall */
anchor( Q, (Q <= Aq), Aq, true, Q ).
/* ======================================================
 * @unroll( N:i, Loop:i, UnRolledLoop:o == next^N(Loop) )
 */
unroll( 0, A, A ).
unroll( N, A, next(B) ):- N >0, M is N - 1, unroll( M, A, B ).
/* ======================================================
 * @anchorcase13( N:integer:i, Query:i, Query':o )
 */
anchorcase13( 1, Q, true ):-!.
anchorcase13( 2, Q, next(Q) ):-!.
anchorcase13( J, Q, (next(Q),S) ):- J > 2, K is J -1,
          anchorcase13( K, next(Q), S ).
/* ====================================================== */
/* @anchorlcm(Query:i,I:i,J:i,In:i,Jm:i,LcmOfIJ:o,QueryLcm:o)
 *  method: repeat ( In:=In+I or Jm:=Jm+J )
 *                 until (In==Jm); return(In)
 */
anchorlcm( _, _, _, Lcm, Lcm, true, Lcm ):-!.
anchorlcm( Q, I, J, In, Jm, (loop(Lcm,UnQ),Qlcm), Lcm ):-
          In < Jm, !,
          InI is In + I,
          unroll( In, Q, UnQ ),
          anchorlcm( Q, I, J, InI, Jm, Qlcm, Lcm ).
anchorlcm( Q, I, J, In, Jm, Qlcm, Lcm ):-
          In > Jm, !,
          JmJ is Jm + J,
```

```
          anchorlcm( Q, I, J, In, JmJ, Qlcm, Lcm ).
/* ================================================== */
/* @unrollhalt1( Term1:i, Term2:i ) fail when
 *    unrolling would go in an infinite loop.
 *  Term1 == next^m loop( I, P ),
 *  Term2 == next^n loop( I, Q ),
 *  assume wlog: n >= m,
 *
 * unrollhalt1( Term1:i, Term2:i ):-
 *   remove "next^(n-m)" from Term1 and Term2.
 */
unrollhalt1( next(P), next(Q) ):- !,
          unrollhalt1( P, Q ).
unrollhalt1( next(P), loop( I, Q ) ):-!, /* goto step 2 */
          unrollhalt2( I, 1, P ).
unrollhalt1( loop( I, Q ), next(P) ):-!,
          unrollhalt2( I, 1, P ).
unrollhalt1( _, _ ).                    /* Otherwise */
/* ========================================================
 * unrollhalt2( I:integer:i, K:integer:i, Term2:i )
 *        K is #(‘next’ prefixed to Term2) mod I.
 */
unrollhalt2( I, I, P ):-!,
          unrollhalt2( I, 0, P ).
unrollhalt2( I, K1, next(P) ):-!,
          K2 is K1 + 1,
          unrollhalt2( I, K2, P ).

unrollhalt2( I, K, loop( J, P ) ):-!,       /* goto step 3 */
          unrollhalt3( I, K, J ).
unrollhalt2( _, _, _).                     /* Otherwise */
/* ========================================================
 * unrollhalt3( I:integer:i, K2:integer:i, J:integer:i )
 *        I,J: Loop sizes,
 *        K2 is #(next prefixes to Term2) mod I.
 *        K3 is #(next prefixes to Term2) mod (I and J).
 *        and then to step 4.
 */
unrollhalt3( I, K2, J ):-!,
         K3 is K2 mod J,
          unrollhalt4( I, K3, J ).
/* ========================================================
 *  @unrollhalt4( I:integer:i, K3:integer:i, J:integer:i )
 *   I,J: Loop sizes. K3 is the offset.
 *   Check if the two loop ever coincide.
 *     Coincide if K3 = 0,
 *     Never coincide if I=J or J/I or I/J.
 */
unrollhalt4( _, 0, _ ):-!.
unrollhalt4( I, _, I ):-!, fail.
unrollhalt4( I, _, J ):- 0 is I mod J, !, fail.
unrollhalt4( I, _, J ):- 0 is J mod I, !, fail.
unrollhalt4( _, _, _).                     /* Otherwise */

/* [e.pro] Query-Prover + cut */
dtlp( Q, O ):-
        normalize( Q, 0, NQ ),
        pl( NQ, O, C ),
        (O,C) \= (failed,backtrack), !.
```

```
dtlp( _, failed ).
/*
 * @pl( Q:Query:i, Out:{proved,failed}:o, Chop:{cut,backtrack}:o )
 *  Always succeeds
 * If Q contains a cut then it returns Chop = cut,
 *  now the caller shouldn't retry this Q.
 * Q=(H,T) or Q=H, where H in { true,failed,!,prolog,some,any,align }
 */
pl( true, proved, backtrack ):-!.
pl( failed, failed, backtrack ):-!.
pl( !, proved, cut ):- !, write(' Cutting at tail end.').
pl( Q, proved, backtrack ):-
        Q \= (_,_),
        prolog( Q ), !,
        write(' Prolog proved: '), write( Q ), nl.
pl( some(Q), O, C ) :-
        normalize( Q, O, R ),
        pl( R, O, C ),
        (O,C) \= (failed,backtrack).
pl( some(Q), O, C ) :-
        R = next(Q),
        write(' Trying: '), write( some(R) ), nl,
        pl( some(R), O, C ).

pl( any(Q), O, C ) :-
        normalize( Q, O, R ),
        pl( R, O, C ),
        (O,C) \= (failed,backtrack).
pl( any(Q), O, C ) :-
        anyplus( Q, R ),
        write(' Trying: '), write( any(R) ), nl,
        pl( any(R), O, C ).

pl( Q, O, C ):-
        Q \= (_,_),
        ruledata( RuleType, Rule ),
        align( AlignType, Q, Rule, ToProveQ, ProvedQ ),
        normalize( ToProveQ, 0, NormQ ),
          write(' '), write( AlignType ), write(' aligning: '),
          write( ProvedQ <= NormQ ), nl,
        pl( NormQ, O, C ),
        (O,C) \= (failed,backtrack).
pl( Q, failed, backtrack ):-
        Q \= (_,_), !,
        write(' Failed and backtracking on: '), write( Q ), nl.
/*
 *  Q = (_,_), ie. a query list.
 */
pl( (true,S), O, C ) :- !, pl( S, O, C ).
pl( (fail,S), failed, backtrack ) :- !,
        write(' Failed and backtracking on '), write( fail+S ), nl.
pl( (!,S), O, cut ) :- !,
        write(' Cut and continuing on: '), write( !+S ), nl,
        pl( S, O, _ ).
pl( (Q,S), O, C ) :-
        prolog( Q ), !,
        write(' Prolog proved head of: '), write( Q+S ), nl,
        pl( S, O, C ).
```

```
pl( (some(Q),S), Ox, Cx ) :-
        normalize( Q, 0, R ),
        pl( R, 0, C ),
        (0,C) \= (failed,backtrack),
        plx( 0, C, S, Ox, Cx ).
pl( (some(Q),S), Ox, Cx ) :-
        R = next(Q),
        write(' Trying: '), write( some(R)+S ), nl,
        pl( (some(R),S), 0, C ),
        (0,C) \= (failed,backtrack),
        plx( 0, C, S, Ox, Cx ).

pl( (any(Q),S), Ox, Cx ) :-
        normalize( Q, 0, R ),
        pl( R, 0, C ),
        (0,C) \= (failed,backtrack),
        plx( 0, C, S, Ox, Cx ).
pl( (any(Q),S), 0, C ) :-
        anyplus( Q, R ),
        write(' Trying: '), write( any(R)+S ), nl,
        pl( any(R), 0, C ),
        (0,C) \= (failed,backtrack),
        plx( 0, C, S, Ox, Cx ).

pl( (Q,S), Ox, Cx ):-
        ruledata( RuleType, Rule ),
        align( AlignType, Q, Rule, ToProveQ, ProvedQ ),
        normalize( ToProveQ, 0, NormQ ),
          write(' '), write( AlignType ), write(' aligning: '),
          write( ProvedQ <= NormQ+S ), nl,
        pl( NormQ, 0, C ),
        (0,C) \= (failed,backtrack),
        plx( 0, C, S, Ox, Cx ).

pl( (Q,R), failed, backtrack ):-
        write(' Failed and backtracking on: '), write( Q+R ), nl.
/*
 * @plx( LastOut:{proved,failed}:i, LastChop:{cut,backtrack}:i,
 *       QueryTail:i, FullOut:{proved,Out}:o,
 *       FullChop:{cut,backtrack}:o ).
 * Always succeeds.
 *  Prove TailQuery only if the first part was proved earlier.
 *  If the first part was cut & failed, then just go back.
 */
plx( failed, C, _, failed, C ):-!.
plx( proved, cut, S,     Ox, cut ):- !, pl( S, Ox, Cx ).
plx( proved,  _, S,     Ox,  Cx ):- pl( S, Ox, Cx ).

?- dtlp(help,_).

/*

*/
```

Summary

Almost all human reasoning involves the notion time, yet there is no agreement on the nature of time and on formalization of temporal reasoning.

Consider the statement: 'whenever it rains, the janitor is absent on the next day.' This simple statement assumes that time is divided into a sequence of days, where each day is related to the next one. Moreover it is a rule applicable on days. The notion of causality is implicit in the rule: the rain causing the janitor to be absent on the next day. Should we assume that occurrence of rain takes place over a time interval or at a time point during the day?

As another example, consider the sentence: 'whenever he threw the ball, it bounced and eventually came to rest.' This statement involves subtler properties of time. It may simply be looked upon as a rule: throwing the ball causes it to bounce for some time. Looking closely, we are able to refine the notion of bouncing as an infinite series of actions that terminates in a finite amount of time. Does it mean time is dense? If time is dense, can there be a time interval with a *dense mix* of events, that is, between any two time points in the interval, there is a point where the event occurred and also a point where the event did not occur? Is it sensible to have a sequence of time points that describe bouncing in reverse time?

These are some of the properties of time we capture in the logics proposed by us and we show how such temporal reasoning can be automated elegantly. Specifically we can model an infinite sequence of events, which may itself be refined into infinite sub-sequences of sub-events that take place in finite time.

We propose *propositional dense time logic* (PDTL) to model dense time. Instead of using the traditional infinite linear sequence of time points, we use an infinite binary tree to model time. We call this the *omega tree* model of time. The right branch of a node in the tree represents the traditional PTL model. While the left child of a node in the tree represents the refinement of the time interval between the node and its next right child. We give a propositional logic language extended with several modalities for expressing events over the omega tree. The modalities *next* and *within* allow us to talk respectively of the right and left child of a given node. The modality *always* (and its dual *sometimes*) let us talk of all the right children together, and they have identical semantics to that in PTL. The modality *every* (and its dual *any*) let us talk of whole sub-trees at a time. In this way, we can talk about both intervals and points over dense time. The sub-trees refine time intervals, and they even allow the representation of infinite subsequences of events in a finite interval of time.

Given a formula of PDTL, we show how a tableau can be built for it, which is exponential in the size of the formula. Nodes of the tableau are closed when we show that the node is unsatisfiable. Eventually if the root of the tableau is still open, we claim that the formula is satisfiable and show how an omega tree model for it can be extracted from the tableau.

We also give several rules of inference and an axiomatization for PDTL. Using this we prove completeness of PDTL.

The division of time into several levels has to be fixed before hand. We claim this division could reflect the natural hierarchy imposed on time by the problem domain and should be based on the

observed periodicity in events over time. Moreover PDTL is ideal for modeling computations where 'uninteresting' subroutine calls can be abstracted in a single step.

It is easy to write a formula in PDTL that asserts a dense mix of an event and its negation over an interval. The models for dense mix are difficult to handle and this phenomenon is rarely encountered in modeling real world events. To filter out such models, we define the notion of *stability* and *recurrence* of events over omega trees. Omega trees which satisfy these properties are called *ordinal trees*.

Ordinal trees are omega trees, where only a finite amount of refinement takes place (on the left branch), and the right branch eventually becomes periodic. Ordinal trees are much easier to handle, have a cleaner representation, can model many temporal scenarios elegantly and lend themselves to temporal logic programming based on horn clauses.

We define *propositional ordinal tree logic* (POTL), still using the language of PDTL. However POTL imposes stricter conditions on the models. The POTL decision procedure extends the PDTL tableau procedure; during the tableau construction we continuously check if a stable ordinal tree can be embedded in a node. The closing of nodes is similar to that of PDTL, except that now we extract ordinal tree models from the tableau 'bottom-up'. Nodes that have stable ordinal tree models are combined to produce larger ordinal tree models until a model for the root is found. we call it the *marking algorithm*. We then give a proof of correctness of the marking algorithm.

We relate PDTL to *propositional dynamic logic* (PDL), and POTL to *monadic second order logic with two successor functions* (S2S). We show how our decidability results can also be obtained by interpretation of PDTL and POTL in PDL and S2S respectively. However we claim that our methods are more intuitive and the modalities have direct temporal interpretation.

We finally show how ordinal trees can be used as temporal data structures in a horn clause logic programming framework called *dense time logic programming* (DTLP). DTLP extends the temporal logic programming language *Templog* designed by Abadi and Manna.

Ordinal trees labeled with prolog predicates in a restricted way give rise to temporal facts, rules and queries in DTLP. We classify the rules according to their temporal scope into three types: *universal* rule, *level* rule, and *anchored* rule. The query answering mechanism called *temporal resolution* is an extension of normal prolog resolution to allow the transfer of temporal information between any two ordinal trees by a mechanism called *aligning*. Temporal resolution works on a query clause trying to align it with temporal rules and facts, expanding the query ordinal tree to fill in details and performing other book-keeping details until the whole query can be proved.

The DTLP interpreter is written in prolog and supports the traditional control mechanism of prolog like *cut* and proves a query left to right, and examines the rules and facts top to bottom. It is capable of answering existential temporal queries, i.e. queries that ask if something ever happened. All variables of DTLP are *rigid*. In future we plan to allow flexible variables by using ordinal trees themselves to store the time dependent values of the variables. Temporal terms offer another interesting area of future work. On the theoretical side, we are trying to axiomatize POTL, and obtain omega trees as a limit of ordinal trees. We are also investigating a logic where predicates operate on ordinal terms.

Acknowledgments

I take this opportunity to thank my guide Dr. G. Venkatesh. This report saw the light of day because of his able and patient guidance.

I thank Dr. R. Parikh, Dr. M. Mukund, Dr. R. Ramanujam, Dr. K. Lodaya, Dr. G. Sivakumar, Dr. A. Sanyal, Dr. D. Singh, Dr. M. Sohoni, Dr. S. Patkar for useful discussions. I thank several anonymous referees for helpful comments. I thank the proof readers: Maya, Mouli and Pagey.

I thank Dr. D. B. Phatak and Dr. S. S. S. P. Rao for their vision and efforts in making the department one of the best in the country. I thank the department faculty, staff, research scholars and students for creating an excellent atmosphere for research. I also thank Sohoni, Patkar, Mouli, Pagey, Satpathy, Gandhe, Babu, Chocks, Oberoi, Deshpande, Uday, Usha, Prakash, Ehab, Vasu, Mukesh, Vinod, Lakshmi, ER, Yagi, Khemani, Bala, Prerna, Jog, Chat, Subba, Raghavan, AKN, Ajit, Shashikant, YKDV, Ramnath, Vinod, Waghmere, Devi, Phadke, Chandran, Athvankar, Laksmi, Joshi, Nitin, PSR, Jacob, Siva, Sharat, Ramani, Rohit, Shuvam, Radha, Chris, Thiru, Venko, Leora, Kartha, Sabata, Santosh, Shabbir, Juzar, Mohammed, my parents: Sofiya and Shafi, and my wife Maya.

The sky is beautiful because of the stars that cannot be seen.

- Mohsin Ahmed.
(May 30, 2020)