DENSE TIME LOGICS

Seminar Report

Mohsin Ahmed
Guide: Dr. G.Venkatesh

Department of Computer Science and Engineering
Indian Institute of Technology, Bombay
May 15, 2018

Abstract

This report studies the extension of propositional linear time temporal logic (PTL) to propositional dense time logic (PDTL) and propositional ordinal tree logic (POTL). We finally describe a prolog implementation of a dense time logic programming (DTLP) system based on ordinal trees.

While a PTL model is a single sequence of states, a PDTL model, called an omega-tree, consists of a nested sequence of states. Two new operators, called *within* and *everywhere* are introduced to access nested sequences. Besides its application in describing activities for Artificial Intelligence PDTL can be used to represent more naturally procedural abstractions in control flow. We show PDTL to be decidable by using a tableau based method, and give a complete axiomatization for it.

PDTL's omega tree models are too rich as they allow a dense mixture of events which have no natural significance. We consider a subset of the omega tree models called ordinal trees which do not allow such mixtures. Their logic called Propositional Ordinal Tree Logic (POTL) is also be shown to be decidable in exponential time. POTL though point based represents interval information and is a good bridge between point based and interval based temporal logics.

We show how the ordinal tree data structure can be embedded within a conventional logic programming language with appropriate temporal operations to manipulate time, and finally we give a simple implementation of DTLP.

DTLP improves upon previous attempts to represent temporal information using temporal logics or event calculii by allowing both point and interval based temporal information to be expressed over a dense time line. Dense time is represented by a hierarchy of nested infinite sequences of points. An ordinal tree breaks up the rational line of time into such sequences of closed-open intervals. An *labeled ordinal tree* is an ordinal tree with its nodes labeled by prolog clauses. The prolog facts (or goals) at a node of the ordinal tree make assertions (or queries) about the time interval represented by that node. We use labeled ordinal tree to represent temporal facts, rules and queries. Causal rules are temporal rules where the causes precede the effects.

Temporal resolution tries to prove temporal queries from a set of temporal facts and rules. Temporal resolution *aligns* two labeled ordinal trees by re-structuring them to facilitate the transfer of temporal information between any two ordinal trees.

Keywords: Temporal logic, dense time, ordinal trees, temporal logic programming, clock trees, prolog.

Contents

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 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 a unknown number of bouncing events.

While there have been a number of studies in representing temporal information within a logical framework [**?**, **?**, **?**, **?**, **?**], 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 [**?**]. However here, we focus on the issue of representing dense time information.

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.

We introduce two new operators into the propositional linear time temporal logic (PTL) [**?**]. The new operators $\oplus$, read *within* and $\boxplus$, read *everywhere*, allow us to open up an interval and look inside it. The dual of $\boxplus$ is $\diamondplus$, read *somewhere*. This gives us the ability to represent dense time information. We can also express a behavior consisting of a sequence of events terminating in a limit point, for example a ball bouncing to rest can be represented by

$$\oplus(Up \wedge \Box(Up \leftrightarrow \bigcirc Down)) \wedge (\bigcirc Rest) \wedge \boxplus(Down \leftrightarrow \neg Up)$$

Where $\{Up, Down, Rest\}$ are propositional symbols.

Such a sequence can be a nested in a bigger sequence of events. For example the statement "Every time he hit the ball, the ball bounced and came to rest", can represented by

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

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

1.1 Propositional Dense Time Logic

We show that the resulting logic, called the Propositional Dense Time Logic (PDTL) is decidable in exponential time by using a tableaux based decision procedure which is similar to that used for deciding satisfiability of PTL formulas [**?**, **?**]. We also give a simple axiomatization of PDTL and show its completeness.

Though PDTL is reducible to the Deterministic Propositional Dynamic Logic (DPDL) [**?**] and hence its decidability follows from that of DPDL [**?**], PDTL's *within* and *next* operators have a direct temporal interpretation in terms of nested sequences. Moreover our tableaux based method is more appealing intuitively, and is a direct extension of the methods used for propositional temporal logic [**?**, **?**].

We find that PDTL is too expressive because it allows us to write a satisfiable sentence:

$$\boxplus(\diamond P \wedge \diamond \neg P)$$

which describes a dense mixture of states that satisfy P and that falsify P. Such formulas do not appear to have any useful application, but on the other hand, contribute to unwieldy models. We look at a subset of models, called ordinal trees, where such mixtures do not appear.

Ordinal trees are finite binary trees with back arcs allowed in a restricted way so that infinitely nested sequences do not appear. We show that the logic of formulas valid over ordinal trees (POTL) is decidable by extending the tableaux based procedure of PDTL by closing nodes which do not contain any ordinal tree models. Thus POTL is decidable in exponential time, as compared to deciding it in non-elementary recursive time by interpreting it in *S2S* logic of Rabin [**?**].

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, and a^k denotes a occurring k times. Omega trees distinguish between $a \cdot a^{\omega} \cdot b$ and $a^{\omega} \cdot b$, but a simple normalization procedure on ordinal trees demonstrates their equivalence.

1.2 Dense Time Logic Programming

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.

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.

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: 'What happened after the ball hit the wall?'. We also show how recursive computations can be modeled naturally using ordinal trees reflecting the natural procedural control mechanisms.

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.

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. $O(max(m, n))$.

Tree resolution works on a query ordinal tree clause trying to match it with temporal facts and rules, expanding the leafs of the query to fill in details and performing other book-keeping until the whole query ordinal tree clause can be proved. For example, if we are given the following two facts: 'Whenever it rained, the janitor was absent the next day' and 'It rains every Tuesday' then the following query will succeed: 'Was the janitor absent on Wednesday?' Notice that in most temporal rules the causes precede the effects, such rules are called *causal rules*.

Ordinal tree allows the representation of information that uses either a finite sequence of events or an infinite periodic sequence of events. However, sometimes it is more convenient to represent finite sequences also 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 hour at 5pm, every second day' and the rule: 'Whenever it rains for a full hour, the buses are late for the next one hour.' can 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.

1.3 Related Literature

The theory of dense time logics has also been studied by a number of researchers: Burgess describes *S4.3*, the modal logic of dense time in [**?**]. Gabbay [**?**] shows that *S4.3* can be interpreted in *S2S*, the *monadic second order logic of two successors*. Rabin gives an non-elementary time decision procedure for *S2S* [**?**, **?**]. Alur and Henzinger in [**?**] show a real time logic to be undecidable.

Several researchers have tried to model temporal phenomena. Dense time has been studied by Burgess in [**?**] as the modal logic $S4.3$, which can be interpreted in the monadic second order theory of two successors *S2S* [**?**]. *S2S* is decidable in non-elementary recursive time [**?**]. Alur and Henzinger show a real time logic to be undecidable [**?**]. We study propositional ordinal tree logic in [**?**] and show it to be decidable in exponential time (in the size of the formula).

Discrete time temporal logic programming has also been investigated by several researchers. Gabbay has proposed [**?**, **?**] temporal deduction systems based on horn clauses. Abadi and Manna show how prolog can be extended with PTL to obtain a temporal logic programming language called *Templog* [**?**]. The completeness and expressive power of *Templog* is studied by Baudinet [**?**]. Hale and Mostowski have implemented [**?**], an interval temporal logic programming language called *Tempura*. A *tempura formula* is also a interval temporal logic program and its execution gives rise to a model for the formula.

Logic of change has been widely studied in AI. Shoham has studied temporal phenomena from the model theoretic point of view, and describes his logic of change in terms of minimal models [**?**].

Gabbay [**?**, **?**] shows how temporal operators may be introduced in a conventional logic programming language and extends horn clause resolution to handle time. Abadi and Manna [**?**]; and Baudinet [**?**] describe *Templog*, an extension to prolog by propositional temporal logic (PTL) operators. Moszkowski and Hale [**?**] describe *Tempura*, an Interval Temporal Logic programming language.

1.4 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 tableaux method for PDTL.
- Practically interesting, as it gives rise to a temporal data structure that is useful in a logic programming language.
- It is based on logic programming like Gabbay's, and can be embedded in a language like Prolog.
- It takes the data structure approach to model time.
- It can model activities, which may contain an infinites 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.
- It models dense time.
- We have shown the propositional ordinal tree logic to be decidable in exponential time [**?**]. However, we believe it is more fruitful to use its models (ordinal trees) directly as a data structure than to try to build a temporal logic programming framework based on POTL on the lines of [**?**].

Chapter 2

Propositional Dense Time Logic

2.1 The Language

Propositional Dense Time Logic (PDTL) is an extension of the usual PTL. The language of PDTL contains the usual propositional logic symbols: propositions $\{p_1, \ldots, p_n\}$, logical connectives:$\neg, \rightarrow$, the usual temporal operators: $\Box$, $\bigcirc$, with the following new operators: $\boxplus$ read *everywhere* and $\oplus$, read *within*. We do not consider the *until* operator of PTL [**?**] in this paper.

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

$$L ::= p_1 | \ldots | p_n | \neg L | L \rightarrow L | \Box L | \bigcirc L | \boxplus L | \oplus L$$

The syntactic definitions are:

$$\Diamond A \stackrel{\text{def}}{=} \neg\Box\neg A \tag{2.1}$$

$$\diamondplus A \stackrel{\text{def}}{=} \neg\boxplus\neg A \tag{2.2}$$

$$A \vee B \stackrel{\text{def}}{=} \neg A \rightarrow B \tag{2.3}$$

$$A \wedge B \stackrel{\text{def}}{=} \neg(A \rightarrow \neg B) \tag{2.4}$$

$$A \leftrightarrow B \stackrel{\text{def}}{=} (A \rightarrow B) \wedge (B \rightarrow A) \tag{2.5}$$

2.2 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 mapping from the propositions to the truth values $\mathcal{T}$ and $\mathcal{F}$. A ω-sequence of states (the usual model for PTL [**?**, **?**]) is a sequence of states indexed by elements of $\aleph$. The model for PDTL is an infinitely nested sequence of states.

Notation:

1. $\aleph \stackrel{\text{def}}{=} \{0, 1, 2, \ldots\}$, the set of natural numbers.

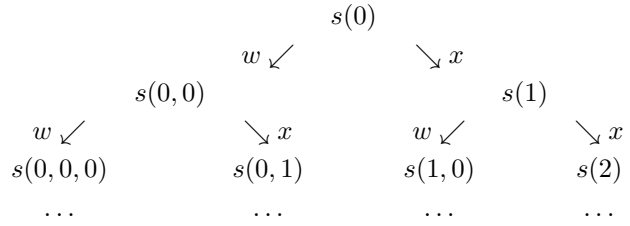

Figure 2.1: Omega-tree

2. $\aleph^+ \stackrel{\text{def}}{=} \{< k_0, \ldots, k_n > \mid k_i \in \aleph,\ 0 \leq i \leq n\}$

3. $\aleph^* \stackrel{\text{def}}{=} \aleph^+ \cup \{<>\}$

4. An element of $\aleph^*$ will be denoted by $\overline{k} =< k_0, \ldots, k_n >$,

5. The concatenation of $\overline{k}$ with i will be denoted by $\overline{k} \cdot i =< k_0, \ldots, k_n, i >$,

6. The concatenation of $\overline{k}$ with $\overline{m} =< m_0, \ldots, m_j >$, will be $\overline{k} \cdot \overline{m} =< k_0, \ldots, k_n, m_0, \ldots, m_j >$.

A omega-tree of states (a model for PDTL) is a collection of states indexed by $\aleph^+$, that is

$$T = \{s(\overline{k}) \mid \overline{k} \in \aleph^+\}$$

We could view this tree as an infinite binary tree rooted at $s(0)$ (see figure **??**). Every state has two children: the w and the x child states. We introduce a restriction that every state $s(\overline{k})$ and its w-child $s(\overline{k} \cdot 0)$ give the same truth assignment to the propositions. The x-child of $s(\overline{k} \cdot i)$ is $s(\overline{k} \cdot i + 1)$. The omega-tree rooted at $s(\overline{k})$ will be denoted by $T(\overline{k})$.

The sequence $\ll s(0), s(1), \ldots \gg$ 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 $\oplus$ operator. The states can be totally ordered as follows:

$$\left[s(\overline{k} \cdot i) < s(\overline{k} \cdot i \cdot \overline{m}) < s(\overline{k} \cdot i + 1)\right]\ \overline{k} \in \aleph^*,\ \overline{m} \in \aleph^+,\ i \in \aleph$$

Truth of formulas at the states is defined as follows:

$$T(\overline{k}) \models \neg A \quad \text{iff} \quad \text{not } (T(\overline{k}) \models A) \tag{2.6}$$

$$T(\overline{k}) \models A \rightarrow B \quad \text{iff} \quad \text{not } (T(\overline{k}) \models A) \text{ or } (T(\overline{k}) \models B) \tag{2.7}$$

$$T(k_1 \cdots k_n) \models \bigcirc A \quad \text{iff} \quad T(k_1 \cdots k_{(n-1)} \cdot k_n + 1) \models A \tag{2.8}$$

$$T(k_1 \cdots k_n) \models \Box A \quad \text{iff} \quad \forall j \geq k_n\, [T(k_1 \cdots k_{(n-1)} \cdot j) \models A] \tag{2.9}$$

The operator $\oplus A$ asserts that A holds in the w-child of the current state, and $\boxplus A$ asserts that A holds everywhere below the current state in the omega-tree model.

$$T(\overline{k}) \models \oplus A \quad \text{iff} \quad T(\overline{k} \cdot 0) \models A \tag{2.10}$$

$$T(k_1 \cdots k_n) \models \boxplus A \quad \text{iff} \quad \forall j \geq k_n,\ \forall \overline{m} \in \aleph^* \quad T(k_1 \cdots k_{(n-1)} \cdot j \cdot \overline{m}) \models A \tag{2.11}$$

2.3 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 1. It is always hot: $\boxplus Hot$

Example 2. There will be rain: $\diamondsuit Rain$

Example 3. If we want to assert that the rain will last over a time interval we assert: $\diamondsuit \oplus \boxplus Rain$.

Example 4. It rains daily: $\Box \oplus \diamondsuit Rain$. 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 5. It never rains for a full day: $\Box \oplus \diamondsuit \neg Rain$.

Example 6. Everyday the rains are followed by a flood: $\Box \oplus \diamondsuit(\oplus \boxplus Rain \wedge \bigcirc \boxplus Flood)$.

Example 7. If he takes a walk everyday, then someday he will get wet in the rain: $\Box \oplus \diamondsuit Walk \rightarrow \Diamond \oplus \diamondsuit(Walk \wedge Rain)$. Note that this is not a theorem. Consider in contrast the next example:

Example 8. Even though it rains daily, it is possible for him to take a walk everyday and not get wet in the rain: $\Box \oplus (\diamondsuit Walk \wedge \diamondsuit Rain \wedge \boxplus(Walk \rightarrow \neg Rain))$

PDTL offers us an interesting way of modeling qualitative temporal information 'event p occurred *many* times' in terms of '$\oplus\Box p$', which deliberately identifies 'many occurrences' with 'infinitely many occurrences in finite time' [**?**]. 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 is represented by $\bigcirc\phi$. If the statement is an abstraction of further statements (e.g. a procedure call), then the properties during the call can be written as $\oplus \bigcirc^i \psi$.

2.4 Satisfiability and Tableaux

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

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

Logical-Nodes. All other nodes are logical-nodes. A logical-node η has a subnode $[\eta]_l$ and, if required subnode $[\eta]_r$.

2.4.1 Tableaux Building

We build a tableaux 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, one formula the *principal formula* of η (denoted by $Pr(\eta)$) will be expanded to generate subnodes. Other formulas in η, called the side formulas are then copied unchanged to the subnode(s).

	$Pr(\eta) \in \eta$	$[\eta]_l$
1	$\Box A$	$A, \bigcirc\Box A$
2	$\Diamond A$	$A \vee \bigcirc\Diamond A$
3	$\boxplus A$	$A, \bigcirc\boxplus A, \oplus\boxplus A$
4	$\Diamond\!\!\!\!-A$	$A \vee (\bigcirc\Diamond\!\!\!\!-A \vee \oplus\Diamond\!\!\!\!-A)$
5	$\neg \bigcirc A$	$\bigcirc\neg A$
6	$\neg \oplus A$	$\oplus\neg A$
7	$\neg\Box A$	$\Diamond\neg A$
8	$\neg\Diamond A$	$\Box\neg A$
9	$\neg\boxplus A$	$\Diamond\!\!\!\!-\neg A$
10	$\neg\Diamond\!\!\!\!-A$	$\boxplus\neg A$

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

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

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

2.4.2 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*. If a node contains both A and $\neg A$, 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 $\Diamond\!\!\!\!-A$.

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 $\Diamond\!\!\!\!-A$ is unsatisfiable at a node, it will be carried over separately to both its x and w descendant.

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 $\Diamond\!\!\!\!-A$, if $\neg A$ holds at every reachable node. An easy way to do this is to check whether each leaf strongly connected component is self-fulfilling [**?**].

Define R, Rx and Rxw descendents of a node as follows:

$$\begin{array}{rcl}
R(\eta) & = & \{\eta\} \cup Rx([\eta]_l) \cup Rx([\eta]_r) \\
 & & \text{if } \eta \text{ is a logical node} \\
R(\eta) & = & \{\eta\} \\
 & & \text{otherwise} \\
Rx(\eta) & = & \{[\eta]_l, [\eta]_r\} \cup Rx([\eta]_l) \cup Rx([\eta]_r) \\
 & & \text{if } \eta \text{ is a logical node} \\
Rx(\eta) & = & \{[\eta]_x\} \cup Rx([\eta]_x) \\
 & & \text{if } \eta \text{ is a state node} \\
Rwx(\eta) & = & \{[\eta]_l, [\eta]_r\} \cup Rwx([\eta]_l) \cup Rwx([\eta]_r) \\
 & & \text{if } \eta \text{ is a logical node} \\
Rwx(\eta) & = & \{[\eta]_x, [\eta]_w\} \cup Rwx([\eta]_w) \cup Rwx([\eta]_x) \\
 & & \text{if } \eta \text{ is a state node} \\
x\text{-leaf-scc}(\eta) & \text{iff} & \forall\mu(\mu \in Rx(\eta) \rightarrow \eta \in Rx(\mu)) \\
wx\text{-leaf-scc}(\eta) & \text{iff} & \forall\mu(\mu \in Rwx(\eta) \rightarrow \eta \in Rwx(\mu))
\end{array}$$

$R(\eta)$ is the set of logical descendants of η, that is, nodes reachable from η through logical arcs. $Rx(\eta)$ 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 $Rwx(\eta)$ 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.

Close any node η that satisfies any one of the conditions:

1. $\psi, \neg\psi \in \eta$
2. All the logical children of η are closed
3. Either $[n]_x$ or $[n]_w$ is closed
4. $\Diamond\psi \in \eta$ and $\psi \notin \bigcup Rx(\eta) \cup \eta$
5. $\Diamond\!\!\!\!-\!\!\!\!\Diamond\psi \in \eta$ and $\psi \notin \bigcup Rwx(\eta) \cup \eta$

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

Proof: ($\Leftarrow$) Let $\{\phi\}$ be closed, we show that it is unsatisfiable. There are five reasons why it was closed, the first three are obvious, the fourth one indefinitely postpones satisfying $\Diamond\phi$, to its x-descendants, without any of them satisfying ϕ, similarly the fifth one indefinitely postpones satisfying a formula $\Diamond\!\!\!\!-\!\!\!\!\Diamond\phi$ to its x and w-descendants separately, without any of them satisfying ϕ.

($\Rightarrow$) 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.
- The set of state nodes in $R(\{\phi\})$ form the set of start nodes for Δ.
- For $\mu_1, \mu_2 \in \Delta$, if $\mu_2 \in R([\mu_1]_x)$ in τ, then an x arc is drawn from μ_1 to μ_2, similarly if $\mu_2 \in R([\mu_1]_w)$ 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.

A omega tree model for the formulas in a node η can be extracted from the set of open state nodes in Δ. Define the mapping $m : \aleph^+ \rightarrow \Delta$ as follows:

Let $m(0) := \eta$
If $m(\overline{k} \cdot k') = \mu$ then
  If not x-leaf-scc(μ) then
    Let $m(\overline{k} \cdot k' + 1)$ be any x-child of μ, and $m(\overline{k} \cdot k' \cdot 0)$ be any w-child of μ.
  If x-leaf-scc(μ) and not wx-leaf-scc(μ) then
    By the leaf condition there is a x-path $\ll \nu_0, \ldots, \nu_k \gg$ passing
    through every node in $Rx(\mu)$, such that $\mu = \nu_0 = \nu_k$, and
    ν_{j+1} is a x-child of ν_j, $\quad 0 \leq j < k$.
    For $j := 0$ to k do
      Let $m(\overline{k} \cdot k' + j) := \nu_j$
      and let $m(\overline{k} \cdot k' + j \cdot 0)$ be any $w-$ child of ν_j
  If wx-leaf-scc(μ) then
    By the leaf condition there is a wx-path $\ll \nu_0, \ldots, \nu_k \gg$ passing
    through every node in $Rwx(\mu)$, such that $\mu = \nu_0 = \nu_k$,
    and ν_{j+1} is a x-child or a w-child of ν_j, for $0 \leq j < k$.
    Let $\overline{l} \cdot l' := \overline{k} \cdot k'$
    For $j := 0$ to $k - 1$ do
      if ν_{j+1} is a x-child of ν_j then
        Let $m(\overline{l} \cdot l' + 1) := \nu_{j+1}$ and let $m(\overline{l} \cdot l' \cdot 0)$ be any w-child of ν_j.
        Let $\overline{l} \cdot l' := \overline{l} \cdot (l' + 1)$
      otherwise let
        $m(\overline{l} \cdot l' \cdot 0) := \nu_{j+1}$ and let $m(\overline{l} \cdot l + 1)$ be any x-child of ν_j.
        Let $\overline{l} \cdot l' := \overline{l} \cdot l' \cdot 0$

Define the mapping $m' : \aleph^+ \rightarrow 2^{literals}$ as:

$$m'(\overline{k}) = \bigcup_{i \geq 0} (m(\overline{k} \cdot \underbrace{0 \cdots 0}_{i \ times}) \cap literals)$$

The omega tree T can now be defined:

$$T = \{s(\overline{k}) | s(\overline{k}) = m'(\overline{k}) \cup \{\neg p_i | p_i \notin m'(\overline{k}), 0 \leq i \leq n\}\}, \quad \overline{k} \in \aleph^+$$

It is easy to see that $T \models \phi$ by induction on the structure of ϕ. 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 $j \in \aleph$ (respectively $T(\overline{k} \cdot k' \cdot \overline{j}) \models \psi$ for some $j \in \aleph^*$). This can be shown using the fact $T(\overline{k} \cdot k')$ was constructed using every node of the x-leaf-scc (wx-leaf-scc respectively) [**?**]. ∎

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 tableaux procedure takes $O(2^{c|\phi|})$ for some constant c. ∎

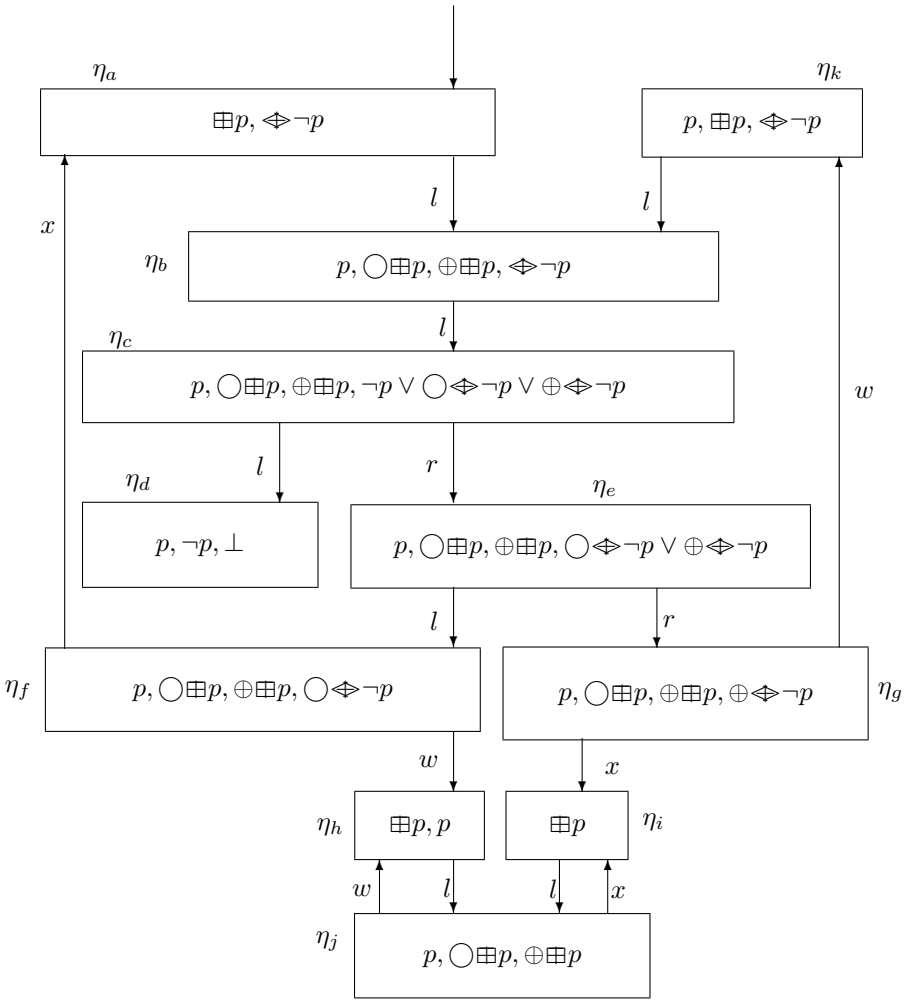

Figure 2.2: Tableaux for $\{\boxplus p, \Diamond\!\!\!\!-\!\!\!\!\triangleright \neg p\}$

2.4.3 Examples

We give a tableaux for the unsatisfiable formula: $\boxplus p \wedge \Diamond\!\!\!\!-\!\!\!\!\triangleright \neg p$ (figure **??**), and for the satisfiable formula: $\Box p \wedge \Diamond\!\!\!\!-\!\!\!\!\triangleright \neg p$ (figure **??**). In the tableaux for $\boxplus p \wedge \Diamond\!\!\!\!-\!\!\!\!\triangleright \neg p$, (figure **??**), we close η_a because $\Diamond\!\!\!\!-\!\!\!\!\triangleright \neg p \in \eta_a$ and $\neg p \notin \bigcup Rwx(\eta_a) \cup \eta_a$.

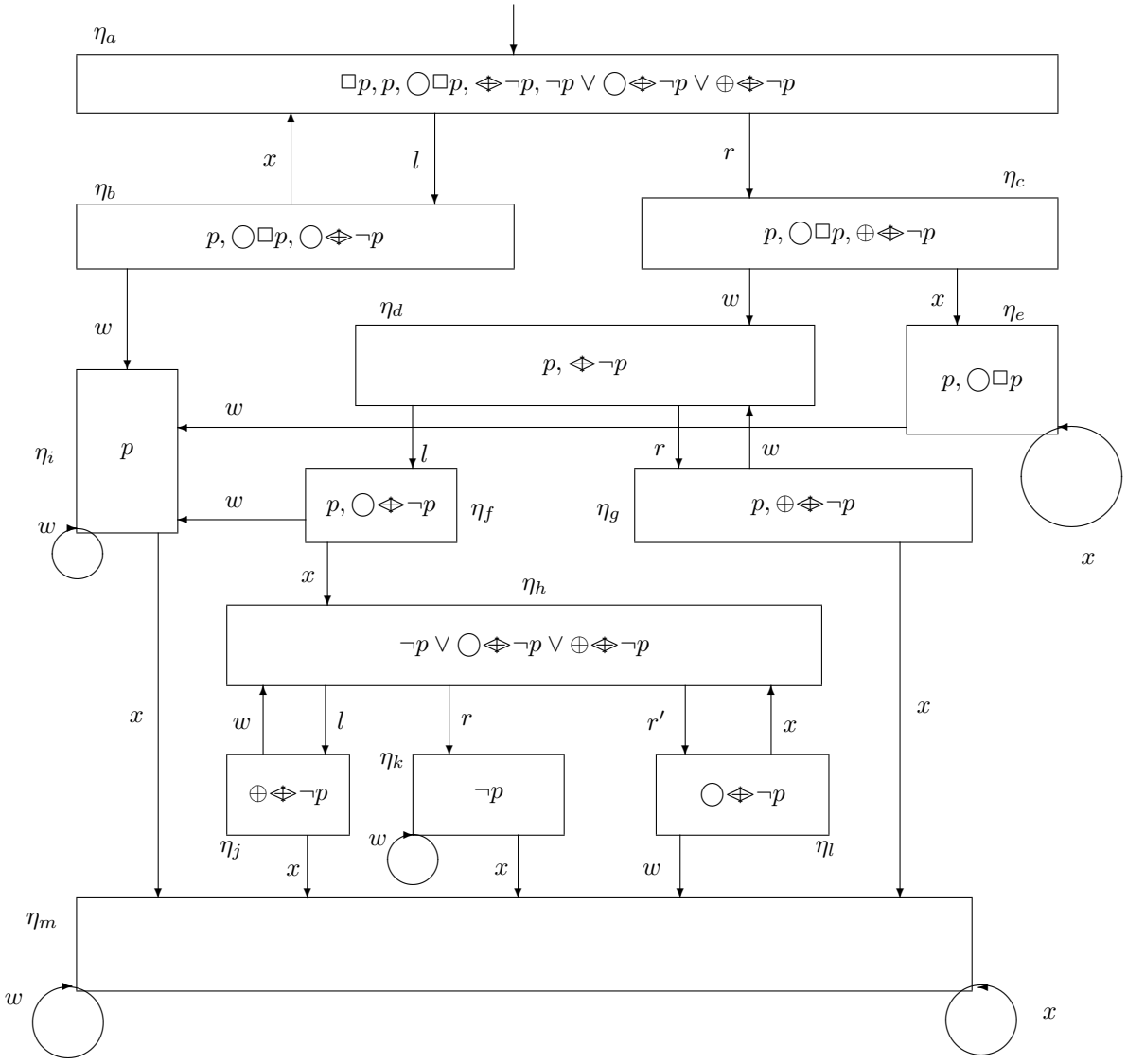

Figure 2.3: Tableaux for $\{\Box p, \Diamond\!\!\!\!-\neg p\}$

2.5 Completeness

Axioms:

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

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.

Some Theorems:

$$
\begin{array}{lll}
T1 & \vdash & \Box(A \wedge B) \leftrightarrow (\Box A \wedge \Box B) \\
T2 & \vdash & \bigcirc(A \wedge B) \leftrightarrow (\bigcirc A \wedge \bigcirc B) \\
T3 & \vdash & \boxplus(A \wedge B) \leftrightarrow (\boxplus A \wedge \boxplus B) \\
T4 & \vdash & \oplus(A \wedge B) \leftrightarrow (\oplus A \wedge \oplus B)
\end{array}
$$

Example 9. $\vdash \boxplus A \to \Box A$

Proof:

$$
\begin{array}{lll}
\vdash (\boxplus A \wedge A) \to (\bigcirc\boxplus A \wedge \bigcirc A) & 1 & Ax8 \\
\vdash (\boxplus A \wedge A) \to \bigcirc(\boxplus A \wedge A) & 2 & T2 \\
\vdash \Box((\boxplus A \wedge A) \to \bigcirc(\boxplus A \wedge A)) & 3 & RN, 2 \\
\vdash (\boxplus A \wedge A) \to \Box(\boxplus A \wedge A) & 4 & Ax9, 3 \\
\vdash \Box(\boxplus A \wedge A) \to (\Box\boxplus A \wedge \Box A) & 5 & T1 \\
\vdash (\boxplus A \wedge A) \to \Box A & 6 & PL, 4, 5 \\
\vdash \boxplus A \to A & 7 & Ax8 \\
\vdash \boxplus A \to \Box A & 8 & PL, 6, 7 \ \blacksquare
\end{array}
$$

Theorem 3. If $\{\phi\}$ is closed, then $\neg\phi$ is provable from the axioms.

Proof: We prove this by induction on the order in which the nodes are closed, assuming at each step that the theorem holds for nodes closed at a 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 A and $\neg A$, then it is unsatisfiable, and 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. If $[\eta]_r$ is absent, let $R = \bot$.

$$\begin{array}{lll} \vdash \neg L & 1 & Hyp. \\ \vdash \neg R & 2 & Hyp. \\ \vdash N \rightarrow (L \vee R) & 3 & PDTL \\ \vdash \neg N & 4 & PL, 1, 2, 3 \end{array}$$

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 $[\eta]_x$:

$$\begin{array}{lll} \vdash \neg X & 1 & Hyp. \\ \vdash \bigcirc \neg X & 2 & RE, Ax8, 1 \\ \vdash \neg \bigcirc X & 3 & Ax5, 2 \\ \vdash N \rightarrow \bigcirc X & 4 & PDTL \\ \vdash \neg N & 5 & PL, 3, 4 \end{array}$$

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 $[\eta]_w$.

$$\begin{array}{lll} \vdash \neg(L \wedge W) & 1 & Hyp. \\ \vdash \oplus\neg(L \wedge W) & 2 & RE, Ax8 \\ \vdash \neg \oplus (L \wedge W) & 3 & Ax6 \\ \vdash N \rightarrow (L \wedge \oplus W) & 4 & PDTL \\ \vdash N \rightarrow (\oplus L \wedge \oplus W) & 5 & Ax11, T4 \\ \vdash N \rightarrow \oplus(L \wedge W) & 6 & T4 \\ \vdash \neg N & 7 & PL, 6, 3 \end{array}$$

5. A node η was closed because $\Diamond A \in \eta$ has been indefinitely postponed. Let $P = \bigvee_{\eta_i \in Rx(\eta)}(N_i)$. We first show $\vdash P \rightarrow \neg A$.

   Let η' be a open node in $Rx(\eta)$ containing $\Diamond A$, it expands into $\bigvee_{i=1}^{m}(A \wedge N_i') \vee \bigvee_{i=1}^{m}(\bigcirc\Diamond A \wedge N_i')$ where $N' = \bigvee_{i=1}^{i=m} N_i'$. The nodes corresponding to $A \wedge N_i'$ are called alpha nodes and those corresponding to $\bigcirc\Diamond A \wedge N_i'$ are called the beta nodes.

   Since $A \notin \cup Rx(\eta)$, all α nodes are closed, therefore by induction hypothesis on closed nodes we have

$$\begin{array}{l} \vdash \bigwedge_{i=1}^{m}((A \wedge N_i') \rightarrow \bot) \\ \vdash \bigwedge_{i=1}^{m}(N_i' \rightarrow \neg A) \\ \vdash (\bigvee_{i=1}^{m} N_i') \rightarrow \neg A \\ \vdash N' \rightarrow \neg A \end{array}$$

   Since this holds for all nodes in $Rx(\eta)$, we have

$$\begin{array}{lll} \vdash P \rightarrow \neg A & 1 & PL \\ \vdash P \rightarrow \bigcirc P & 2 & PL, Rx(\eta) \\ \vdash \Box(P \rightarrow \bigcirc P) & 3 & RN \\ \vdash N \rightarrow (P \wedge \Box(P \rightarrow \bigcirc P)) & 4 & PL, 3 \\ \vdash N \rightarrow \Box P & 5 & Ax9, 4 \\ \vdash N \rightarrow \Diamond A & 6 & PL, Hyp. \\ \vdash N \rightarrow \Box\neg A & 7 & Ax1, 5, Hyp. \\ \vdash \neg N & 8 & PL, 6, 7 \end{array}$$

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 $T = \bigvee_{\eta_i \in Rwx(\eta)} N_i^*$, where $N_i^* = \wedge(\eta_i - \{\Diamond\!\!\!\!-A, \bigcirc\Diamond\!\!\!\!-A, \oplus\Diamond\!\!\!\!-A\})$. The formulas N_i^* are so chosen because we do not need $\Diamond\!\!\!\!-A$ to prove $\neg A$ and we need to show $T \rightarrow (\bigcirc T \wedge \oplus T)$.

   Let η' be a open node in $Rwx(\eta)$ containing $\Diamond\!\!\!\!-A$, it expands into $\bigvee_{i=1}^{m}(A \wedge N_i') \vee \bigvee_{i=1}^{m}(\bigcirc\Diamond\!\!\!\!-A \wedge N_i') \vee \bigvee_{i=1}^{m}(\oplus\Diamond\!\!\!\!-A \wedge N_i')$ where $N' = \bigvee_{i=1}^{i=m} N_i'$. The three corresponding sets of nodes are called the alpha, beta and gamma nodes. As seen before $\vdash N' \rightarrow \neg A$.

   Let μ be a beta (respectively gamma) node of η, then $\Diamond\!\!\!\!-A$ will not carry over to $[\mu]_w$ (respectively $[\mu]_x$). To prove $\neg A$ in the nodes of the $[\mu]_w$ (respectively $[\mu]_x$) subtree, we can use the nodes of the subtree at a gamma (respectively beta) node $\mu' \in Rwx(\eta)$ with the same formula as μ and where $\Diamond\!\!\!\!-A$ is carried over. Hence $\neg A$ is provable in the subtree of μ also.

$\vdash T \rightarrow \neg A$	1	*Above*
$\vdash T \rightarrow (\bigcirc T \wedge \oplus T)$	2	*Above*
$\vdash \boxplus(T \rightarrow (\bigcirc T \wedge \oplus T))$	3	RE
$\vdash N \rightarrow (T \wedge \boxplus(T \rightarrow (\bigcirc T \wedge \oplus T)))$	4	$PL, 3$
$\vdash N \rightarrow \boxplus T$	5	$Ax10, 4$
$\vdash N \rightarrow \Diamond\!\!\!\!-A$	6	PL
$\vdash N \rightarrow \boxplus\neg A$	7	$1, 5, Ax2$
$\vdash \neg N$	8	$PL, 6, 7$ ∎

Corollary 4. (Completeness) $(\models \phi) \Rightarrow (\vdash \phi)$

Proof:

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

Chapter 3

Propositional Ordinal Tree Logic

3.1 Ordinal Trees and POTL

PDTL allows formulas such as: $\boxplus(\Diamond P \wedge \Diamond \neg P)$ expressing a dense mix of P and $\neg P$, which do not seem to be useful. Therefore we consider a restricted set of models, called ordinal trees and its logic: the Propositional Ordinal Tree logic (POTL). Ordinal trees are interval based rather than point based models, and they capture Gabbay's notion of recurrence and persistence [**?**]. Figure **??** shows an ordinal tree for the a language with propositions $\{p_1, p_2\}$.

Ordinal trees are finite binary trees possibly with back arcs, with the following properties:

1. A back arc from node η' to η is allowed only if $\eta' \in Rx(\eta)$.
2. Each non-leaf node is a state node, i.e. a mapping from the propositions to the truth values $\mathcal{T}$ and $\mathcal{F}$.
3. Each leaf node η is labeled by a formula $\zeta \in \Sigma$, where

$$\Sigma \quad = \quad \{\boxplus l_1 \wedge \ldots \wedge \boxplus l_n | l_i = p_i \text{ or } l_i = \neg p_i, 0 < i \leq n\}$$

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.

3.2 Tableaux for POTL

The PDTL tableaux procedure will be extended to check if the formula ϕ has any ordinal tree models. 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 $\eta = \{\bigwedge_i l_i, \bigwedge_j \oplus \phi_j, \bigwedge_k \bigcirc \psi_k\}$ then $[\eta]_s = \{\bigwedge_i \boxplus l_i, \bigwedge_j \phi_j\}$.

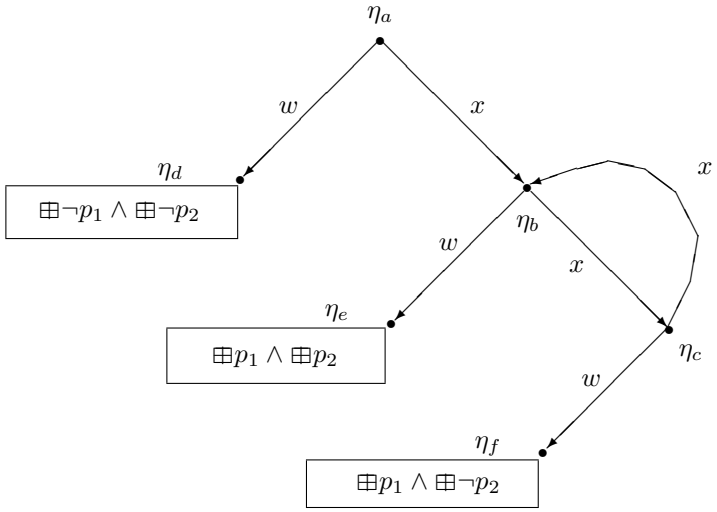

Figure 3.1: An ordinal tree

Define $Rsxw$ descendents of a node as follows:

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

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

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

Marking Nodes

Nodes that have an ordinal tree model will be marked by a sequence of functions $\beta_i, i \geq 0$ defined on τ. A node η will be called *marked* at stage i, if $\beta_i(\eta)$ is defined, that is $\beta_i(\eta) \neq \perp$. The marking will proceed bottom up: first marking nodes in all swx-leaf-sccs, and then proceeding upwards marking nodes that can form ordinal trees from the nodes already marked.

The s-children ensure that all swx-leaf-scc nodes have ordinal tree models. That is, for any node $\eta \in \tau$: swx-leaf-scc(η) implies that $\bigcup Rswx(\eta) \cap literals$ is consistent, and therefore η has an ordinal tree model that satisfies the formula $\sigma(\eta)$ given below:

$$\sigma(\eta) = \bigwedge \left(\begin{array}{ll} \{\boxplus l : l \in ((\bigcup Rswx(\eta)) \cap literals)\} & \cup \\ \{\boxplus\neg p_i : p_i \notin ((\bigcup Rswx(\eta)) \cap literals), 0 \leq i \leq n\} & \end{array} \right)$$

The marking function β_i for a large enough i, maps a node of the tableau to an ordinal tree model for that node. The range of β_i, $i \geq 0$ is the language $\mathcal{L}$ given below:

- For $\eta \in \tau$, if swx-leaf-scc(η) then $\eta \in \mathcal{L}$.
- If $t_1, t_2 \in \mathcal{L}$ then $t_1 \cdot t_2 \in \mathcal{L}$.
- If $t \in \mathcal{L}$ then $t^\omega \in \mathcal{L}$.

Each string in $\mathcal{L}$ represents an ordinal tree model, for example the ordinal tree in figure **??** is represented by $\eta_d \cdot (\eta_e \cdot \eta_f)^\omega$. Initially nodes in swx-leaf-sccs are marked: $\beta_0(\eta) = \eta$ if swx-leaf-scc(η), otherwise $\beta_0(\eta) = \bot$. The initial marking β_0 is extended to $\beta_i, 0 < i \leq |\tau|$ by the *marking algorithm* given in table **??**.

Theorem 5. ϕ has an ordinal tree model iff $\beta_{|\tau|}(\{\phi\}) \neq \bot$, where $|\tau|$ is the size of the tableau τ for ϕ.

Proof: ($\Rightarrow$) The mark of the node at stage i (i.e. $\beta_i(\eta)$) itself represents an ordinal tree model starting at that node.
($\Leftarrow$) Let T be an ordinal tree model satisfying ϕ, without loss of generality we can assume T is minimal, i.e. there are no ordinal tree models with fewer nodes than T satisfying ϕ. T can be embedded in τ, since τ contains all the information regarding all the models of ϕ. For each $\eta \in T$, let (η) be the corresponding node in τ. A leaf node of T will correspond to a node in a leaf node of τ. It is easy to see that for any $\eta \in T$, $(\eta) \in \tau$ will get marked by the marking algorithm by stage i, where i is the length of the largest path (without loops) from η to a swx-leaf-scc in T. ∎

Corollary 6. 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 $O(|E|+|\tau|)$, where $|E|$ is the number of edges in τ, $|\tau| \leq 2^{c|\phi|}$ and $|E| \leq 2^{2c|\phi|}$ for some constant c. ∎

Example 10. The formula $\boxplus(p \vee \neg p)$ has a ordinal tree model, the stable leaf nodes of its tableau (figure **??**) are η_c and η_f. The tableau has been simplified for exposition. The marking of its nodes is given below:

	η_a	η_b	η_c	η_d	η_e	η_f
β_0	$\bot$	$\bot$	η_c	$\bot$	$\bot$	η_f
β_1	$(\eta_c \cdot \eta_f)^\omega$	$(\eta_f \cdot \eta_c)^\omega$	η_c	$\bot$	$\bot$	η_f
β_2	$(\eta_c \cdot \eta_f)^\omega$	$(\eta_f \cdot \eta_c)^\omega$	η_c	$\eta_c \cdot (\eta_c \cdot \eta_f)^\omega$	$\eta_f \cdot (\eta_f \cdot \eta_c)^\omega$	η_f

Example 11. 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 **??** will be closed because η_c (correspondingly η_f) form a swx-leaf-scc which does not satisfy $\neg p$ (respectively p).

3.2.1 Equivalence of POTL Formulas

As we have seen up to now, the choice of time division into levels plays an important role in the way formulas are interpreted. However, there may be two different time divisions T and T', and two formulas ϕ based on T and ϕ' based on T' describing the same sequence of events. In which case ϕ and ϕ' are equivalent, but in PDTL (or POTL) we cannot prove $\phi \leftrightarrow \phi'$.

Table 3.1: Marking Algorithm

For $i := 1$ to $|\tau|$ do
  For all $\eta \in \tau$ do
    If $\beta_i(\eta) \neq \bot$ then $\beta_{i+1}(\eta) := \beta_i(\eta)$
    Else
      Case *node type* of η:
        Logical Node
          If $\beta_i([\eta]_l) \neq \bot$ then $\beta_{i+1}(\eta) := \beta_i([\eta]_l)$
          Elseif $\beta_i([\eta]_r) \neq \bot$ then $\beta_{i+1}(\eta) := \beta_i([\eta]_r)$
          Else $\beta_{i+1}(\eta) := \bot$
          Endif
        State Node
          If $\beta_i([\eta]_s) \neq \bot$ and $\beta_i([\eta]_x) \neq \bot$ then $\beta_{i+1}(\eta) := \beta_i([\eta]_s) \cdot \beta_i([\eta]_x)$
          Elseif $\beta_i([\eta]_w) \neq \bot$ and $\beta_i([\eta]_x) \neq \bot$ then $\beta_{i+1}(\eta) := \beta_i([\eta]_w) \cdot \beta_i([\eta]_x)$
          Elseif x-leaf-scc(η) then
            Let $\pi = \ll \eta_0, \ldots, \eta_m \gg$ be a path going
            through all the state nodes of the x-leaf-scc(η), where $\eta = \eta_0 = \eta_m$.
            For $j \in m$ do
              If $\beta_i([\eta_j]_s) \neq \bot$ then $\zeta_j := \beta_i([\eta_j]_s)$
              Elseif $\beta_i([\eta_j]_w) \neq \bot$ then $\zeta_j := \beta_i([\eta_j]_w)$
              Else $\zeta_j := \bot$
              Endif
            Endfor
            If $\forall j \in m : \zeta_j \neq \bot$ then $\beta_{i+1}(\eta) := (\zeta_0 \cdots \zeta_{m-1})^\omega$
            Else $\beta_{i+1}(\eta) := \bot$
            Endif
          Else $\beta_{i+1}(\eta) := \bot$
          Endif
      Endcase
    Endif
  Endfor
Endfor

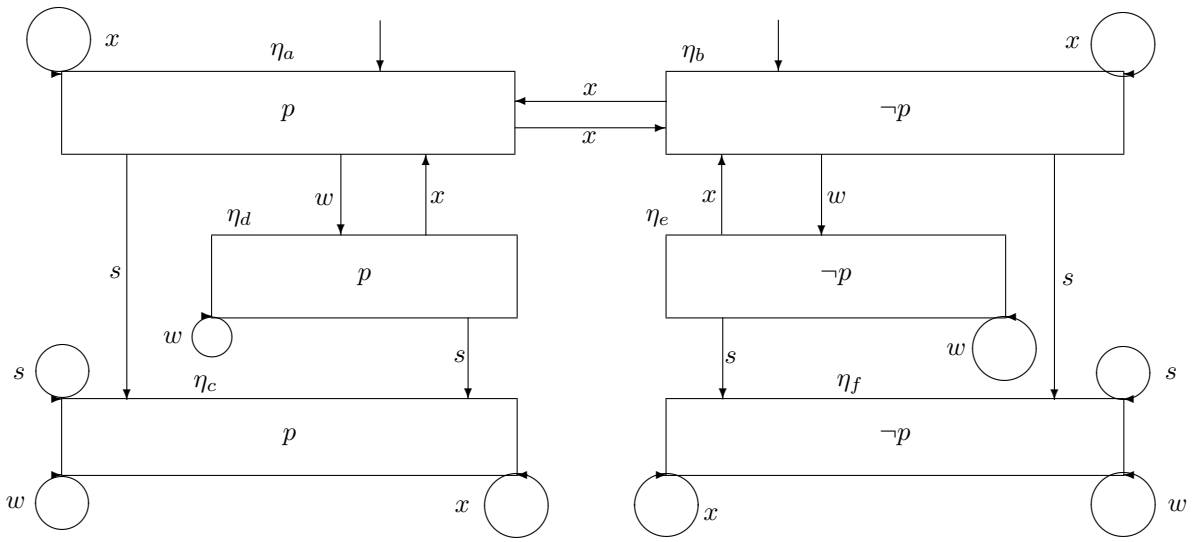

Figure 3.2: Ordinal tree tableau for $\boxplus(p \vee \neg p)$

	p	p	p	...		
$\oplus\Box p$	$s(0)$	...	...	...	$s(1)$	$s(2)$
	$s(0,0)$	$s(0,1)$	$s(0,2)$	...		
$p \wedge \bigcirc \oplus \Box p$	$s(0)$	$s(1)$	...	...	$s(2)$	$s(3)$
		$s(1,0)$	$s(1,1)$	...		

Figure 3.3: Event sequences corresponding to $\oplus\Box p$ and $p \wedge \bigcirc \oplus \Box p$

Example 12. The event sequences corresponding to $\oplus\Box p$ and $p \wedge \bigcirc \oplus \Box p$ are identical except for the way we have chosen to label the states (figure **??**). In the following sections we give a more refined method for showing the equivalence of POTL formulas.

3.2.2 Ordinal Strings and Ordinal Languages

Ordinal trees can be mapped to a set of strings called ordinal strings ($o(\Sigma)$) on the alphabet Σ. We define ordinal strings before defining the mapping.

$$\Sigma \subseteq o(\Sigma) \tag{3.1}$$

$$a \in o(\Sigma) \rightarrow a^{\omega} \in o(\Sigma) \tag{3.2}$$

$$a, b \in o(\Sigma) \rightarrow a \cdot b \in o(\Sigma) \tag{3.3}$$

An *ordinal language* is any subset of $o(\Sigma)$. We can associate with each ordinal tree an ordinal string using the mapping θ:

1. If η is a leaf, then $\theta(T(\eta)) = \sigma(\eta)$
2. If $\eta_1, \ldots, \eta_n, \eta_1$ is a circular path then

$$\theta(T(\eta_1)) = (\theta(T([\eta_1]_w)) \cdot \theta(T([\eta_2]_w)) \cdots \theta(T([\eta_n]_w)))^{\omega}$$

3. If $\eta \notin Rx([\eta]_x)$ and η is not a leaf node then

$$\theta(T(\eta)) = \theta(T([\eta]_w)) \cdot \theta(T([\eta]_x))$$

Example 13. The ordinal string corresponding to the ordinal tree in figure **??** is $(\boxplus\neg p_1 \wedge \boxplus\neg p_2) \cdot ((\boxplus p_1 \wedge \boxplus p_2) \cdot (\boxplus p_1 \wedge \boxplus\neg p_2))^\omega$, where $\Sigma = \{\boxplus p_1 \wedge \boxplus p_2, \boxplus\neg p_1 \wedge \boxplus p_2, \boxplus p_1 \wedge \boxplus\neg p_2, \boxplus\neg p_1 \wedge \boxplus\neg p_2\}$.

3.2.3 Normalizing Ordinal Strings

As there are many representations for the same ordinal string we define a normalization operation on them. The string $b \cdot b^\omega$ says that a event b was followed by an infinite sequence of the same event, which is the same as b^ω.

As another example of normalization, consider the infinite sequence representing a bouncing ball. This can be written as $(down \cdot up)^\omega$, or equivalently we can say that the ball fell down and started bouncing as $down \cdot (up \cdot down)^\omega$. In this case we choose $(down \cdot up)^\omega$ as the normal form.

The following rules normalize ordinal strings:

1. $a^\alpha \Rightarrow a$, where $a \in \Sigma$.
2. $a^\alpha \cdot a^\beta \Rightarrow a^\beta$ where $\alpha + \beta = \beta$ in ordinal addition [**?**].
3. $a \cdot (b \cdot c)^\alpha \Rightarrow (b \cdot c)^\alpha$ where $\alpha \geq \omega$ and $(a \cdot b) \Rightarrow b$.
4. $(a \cdot b \cdot c)^\alpha \Rightarrow (a \cdot b)^\alpha$ where $\alpha \geq \omega$ and $(c \cdot a) \Rightarrow a$.
5. $(a \cdot b)^\omega \cdot (b \cdot a)^\alpha \Rightarrow a \cdot (b \cdot a)^\alpha$ where $\alpha \geq \omega^2$.
6. $b \cdot (a \cdot b)^\omega \Rightarrow (b \cdot a)^\omega$

Clearly this operation can be done directly on an ordinal tree to reduce it to a *normal* ordinal tree.

Example 14. $(\boxplus P) \cdot (\boxplus P)^\omega \Rightarrow (\boxplus P)$

Example 15. $a^{\omega^2} \cdot a^{\omega^3} \Rightarrow a^{\omega^3}$

Example 16. $b \cdot (b^\omega \cdot a)^\omega \Rightarrow (b^\omega \cdot a)^\omega$

Example 17. $(a^\omega \cdot b \cdot a)^\omega \Rightarrow (a^\omega \cdot b)^\omega$

Example 18. $(a \cdot b)^\omega \cdot (b \cdot a)^{\omega^3} \Rightarrow a \cdot (b \cdot a)^{\omega^3}$

Example 19. $b^\omega \cdot b \cdot (b^\omega \cdot b)^{\omega^2} \Rightarrow b^{\omega^3}$

Example 20. $b \cdot (a \cdot b)^\omega \cdot (b \cdot a)^{\omega^2} \Rightarrow (b \cdot a) \cdot (b \cdot a)^{\omega^2} \Rightarrow (b \cdot a)^{\omega^2}$

Example 21. $b \cdot (a \cdot b)^\omega \cdot (c \cdot b \cdot a)^{\omega^2} \Rightarrow b \cdot a \cdot (c \cdot b \cdot a)^{\omega^2}$

Example 22. $b \cdot (a \cdot c \cdot b)^\omega \Rightarrow (b \cdot a \cdot c)^\omega$

Equivalence of Ordinal Tree Formulas

As we have seen, the models extracted from POTL formulas are ordinal trees. We can convert these models into ordinal strings and then normalize them. Two formulas (whether or not equivalent under POTL) represent the same infinite sequence of events, if the normalized ordinal sequences extracted from the two sets of ordinal trees models are equal.

Chapter 4

Dense Time Logic Programming

In this chapter we use the tools developed in the previous chapter to develop a practical temporal logic programming system. The system *DTLP* (Dense Time Logic Programming) extends prolog with ordinal trees to represent temporal information. The implementation of DTLP in prolog is given in Appendix A.

We first explain the theoretical basis of omega and ordinal trees, and introduce new notation for representing ordinal trees in prolog.

4.1 Omega Trees

An omega tree is an infinite binary tree representing the rational line of time (see figure **??**). A node of the omega tree represents a closed-open interval of time. Its root (Λ) corresponds to the whole interval of time, starting from the time point zero. If a node η corresponds to a interval staring 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 **??** the root is $\eta_1 = \Lambda$ and $\eta_2 = [\eta_1]_w, \eta_3 = [\eta_1]_x, \eta_6 = [\eta_3]_w = [\eta_1]_{xw}$.

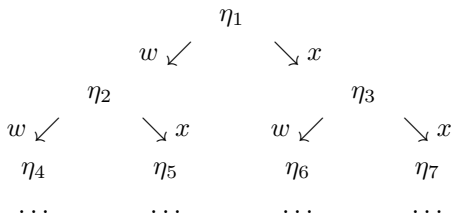

Figure 4.1: Omega-tree

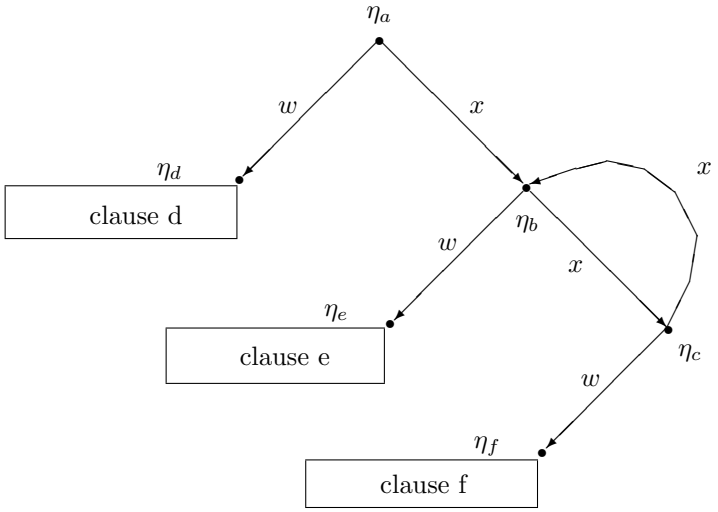

Figure 4.2: An Ordinal-tree

Labeling nodes

The nodes of the omega tree can be labeled by formulae of some object language Σ to assert facts (or queries) about those time intervals. A labeled omega tree t is a mapping σ from the nodes of t to Σ, and is denoted by t_σ. In this paper Σ will be the set of prolog clauses over some fixed language, and the subscript σ will be omitted. We use `true` 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.

4.2 Ordinal Trees

Omega trees with regular labelings (called ordinal trees) can be finitely represented by labeled directed graphs as shown in figure **??**. 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 **??**, 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 **??**, η_d, η_e and η_f are the leaf nodes.

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

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

   Then the tree $t([\eta]_{x^k})$ is replaced by an back-x-arc to η. The nodes $[\eta]_{x^i}, 0 \leq i < k$ are called *loop-nodes* and are not allowed to be labeled. In figure **??**, η_b and η_c are loop nodes.

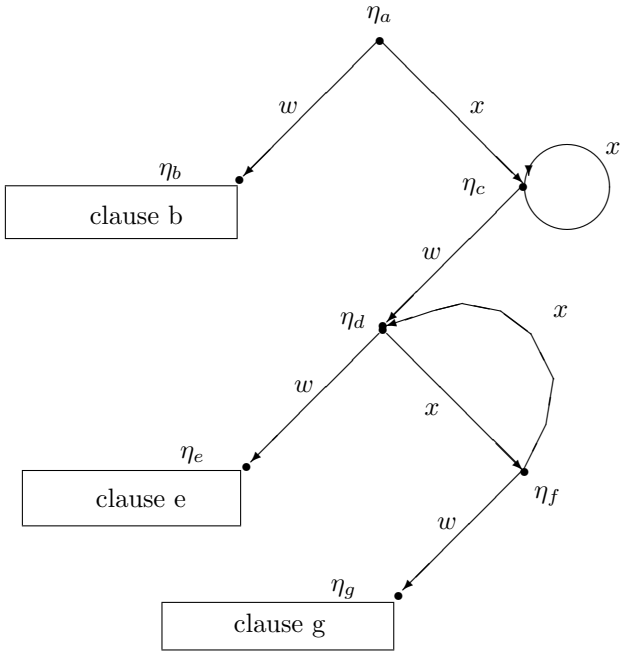

Figure 4.3: Nested Loops

Example 23. *(Nested loops)* The ordinal tree in figure **??** 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 given by clause-e and clause-g.

4.3 Representing Ordinal Trees

Ordinal trees can be represented in prolog using nested lists, and by declaring '`next`' and '`within`' as prefix operators. Loops are represented by the binary functor '`loop`', with its first argument denoting the size of the loop.

The clause labeling a leaf node is written as itself. A tree whose root is a split node η with label C, is written as:

$$[\ C,\ \texttt{within}\ L_1,\ \texttt{next}\ L_2\ ]$$

where $t([\eta]_w)$ and $t([\eta]_x)$ have been written as L_1 and L_2 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:

$$\texttt{loop}\,(k+1, [\,\texttt{next}^{\,0}\,\texttt{within}\,L_0,\,\texttt{next}^{\,1}\,\texttt{within}\,L_1, \ldots,\,\texttt{next}^{\,k}\,\texttt{within}\,L_k\,]\,)$$

where the lists L_i represents $t([\eta]_{x^i w})$ for $0 \leq i \leq k$.

Note that $\texttt{next}^{\,i}$ denotes `next` repeated i times. $\texttt{next}^{\,0}$ is written merely for clarity and is not present in the corresponding clause.

Example 24. The tree in figure **??** can be written as:

$$
\begin{array}{l}
[\sigma(\eta_a), \\
\quad \texttt{within}\,\sigma(\eta_d), \\
\quad \texttt{next loop}\,(2, [\\
\quad\quad \texttt{next}^{\,0}\,\texttt{within}\,\sigma(\eta_e), \\
\quad\quad \texttt{next}^{\,1}\,\texttt{within}\,\sigma(\eta_f)\,]\,)\,]
\end{array}
$$

where $\sigma(\eta)$ denotes the label of η.

Example 25. The tree in figure **??** can be written as:

$$
\begin{array}{l}
[\sigma(\eta_a), \\
\quad \texttt{within}\,\sigma(\eta_b), \\
\quad \texttt{next loop}\,(1, [\\
\quad\quad \texttt{next}^{\,0}\,\texttt{within loop}\,(2, [\\
\quad\quad\quad \texttt{next}^{\,0}\,\texttt{within}\,\sigma(\eta_e), \\
\quad\quad\quad \texttt{next}^{\,1}\,\texttt{within}\,\sigma(\eta_f)\,]\,)\,]\,)\,]
\end{array}
$$

4.4 Standardizing Ordinal Tree Representation

The operators '`within`', '`next`', and '`loop`' distribute over their list arguments, moreover the operators '`loop`' and '`next`' commute. The logical operator '$\Leftarrow$' is meta-temporal and hence commutes with '`within`', '`next`', and '`loop`'.

Given an labeled ordinal tree, its representation is said to be in *standard form* when the operators are distributed over their list arguments, and the '`loop`' is always followed by the '`within`'. In this form, temporal expressions with '`true`' labels can be conveniently omitted from the list. Each element of the resulting list is called a *temporal clause*, if it has at most one head. A missing body of a temporal clause is replaced by '`true`'.

Example 26. *(Sequence of events, standardizing)* Consider the ordinal tree for the following facts: 'The ball hit the wall before rolling into the drain.'

$$
\begin{array}{l}
[\,\texttt{true}\,, \\
\quad \texttt{within}\,(\mathit{hit}(\mathit{ball}, \mathit{wall}) \Leftarrow), \\
\quad \texttt{next}\,[\,\texttt{true}\,, \\
\quad\quad \texttt{within}\,(\mathit{rolling}(\mathit{ball}, \mathit{ground}) \Leftarrow), \\
\quad\quad \texttt{next}\,(\mathit{in}(\mathit{ball}, \mathit{drain}) \Leftarrow)\,]\,]
\end{array}
$$

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

$$
\begin{array}{l}
\texttt{within}\,(\mathit{hit}(\mathit{ball}, \mathit{wall})) \Leftarrow \texttt{true}\,, \\
\texttt{next within}\,(\mathit{rolling}(\mathit{ball}, \mathit{ground})) \Leftarrow \texttt{true}\,, \\
\texttt{next next}\,(\mathit{in}(\mathit{ball}, \mathit{drain})) \Leftarrow \texttt{true}\,.
\end{array}
$$

Example 27. *(Infinite sub-activity, standardizing)* 'The ball fell from the table and bounced before rolling into

the drain.'

$$
\begin{array}{l}
[\,\texttt{true}\,,\\
\quad \texttt{within}\,(falling(ball, table) \Leftarrow\ ),\\
\quad \texttt{next}\,[\,\texttt{true}\,,\\
\quad\quad \texttt{within loop}\,(2, [\\
\quad\quad\quad \texttt{next}^{\,0}\,\texttt{within}\,(moving(ball, up) \Leftarrow\ ),\\
\quad\quad\quad \texttt{next}^{\,1}\,\texttt{within}\,(moving(ball, down) \Leftarrow\ )]\,)\\
\quad\quad \texttt{next}\,[\,\texttt{true}\,,\\
\quad\quad\quad \texttt{within}\,(rolling(ball, ground) \Leftarrow\ ),\\
\quad\quad\quad \texttt{next}\,(in(ball, drain) \Leftarrow\ )\,]\,]\,]
\end{array}
$$

standardizing, we get the following temporal clauses:

$$
\begin{array}{l}
\texttt{within}\, falling(ball, table) \Leftarrow \texttt{true}\,,\\
\texttt{next within loop}\,(2,\ \texttt{within}\, moving(ball, up)) \Leftarrow \texttt{true}\,,\\
\texttt{next within next loop}\,(2,\ \texttt{within}\, moving(ball, down)) \Leftarrow \texttt{true}\,,\\
\texttt{next next within}\, rolling(ball, ground) \Leftarrow \texttt{true}\,,\\
\texttt{next next next}\, in(ball, drain) \Leftarrow \texttt{true}\,.
\end{array}
$$

4.5 Temporal Clauses

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

1. A **fact** ordinal tree is labeled by prolog facts. By standardizing the ordinal tree, we get a set of temporal facts.

   Example 28. *(Simple Facts)* 'The ball hit the wall before rolling into the drain' is represented by the following three facts:

   $$
   \begin{array}{l}
   \texttt{within}\, hit(ball, wall) \Leftarrow \texttt{true}\\
   \texttt{next within}\, rolling(ball, ground) \Leftarrow \texttt{true}\\
   \texttt{next next}\, in(ball, drain) \Leftarrow \texttt{true}\,.
   \end{array}
   $$

   In the examples that follow assume that time is divided into an infinite sequence of days.

   Example 29. 'It will rain the whole day and there will be a flood in the latter half of the day.'

   $$
   \begin{array}{l}
   \texttt{within}\, rain \Leftarrow\ ,\\
   \texttt{within next}\, flood \Leftarrow\ .
   \end{array}
   $$

   Example 30. *(Loop information)* 'It rains 10 units every Tuesday.'

   $$\texttt{next next loop}\,(7,\ \texttt{within}\, rain(10)) \Leftarrow \texttt{true}\,.$$

   Example 31. *(Loop facts)* 'The ball fell from the table and bounced before rolling into the drain' is represented by the following facts:

   $$
   \begin{array}{l}
   \texttt{within}\, falling(ball, table) \Leftarrow \texttt{true}\,,\\
   \texttt{next within loop}\,(2,\ \texttt{within}\, moving(ball, up)) \Leftarrow \texttt{true}\,,\\
   \texttt{next within loop}\,(2,\ \texttt{next within}\, moving(ball, down)) \Leftarrow \texttt{true}\,,\\
   \texttt{next next within}\, rolling(ball, ground) \Leftarrow \texttt{true}\,,\\
   \texttt{next next next}\, in(ball, drain) \Leftarrow \texttt{true}\,.
   \end{array}
   $$

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

2. A **temporal query** is list of temporal goals, where each temporal goal is a temporal expression.

   Example 32. *(Query)* 'Was the janitor absent on Wednesday?'

   $$\Leftarrow \texttt{next next next within}\, absent(janitor).$$

   Example 33. *(Query with rigid variables)* 'Will there be rain on Monday and flood on Tuesday, and will the rain be twice the amount of the flood?'

   $$\Leftarrow \texttt{next within}\, rain(X),\ Y\ \ \texttt{is}\, X\ \ \texttt{div}\, 2,\ \texttt{next next within}\, flood(Y).$$

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

   - A **universal rule** 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`'. It commutes with `next` and `within`, but does not distribute over $\Leftarrow$. Universal rules have no temporal structure because a tree universal rule leads to degeneracy as it can be applied recursively *within* a subinterval.

     Example 34. *(Universal rule)* 'The bulb is on, whenever the switch is on.'

     $$\texttt{every}\,(on(bulb) \Leftarrow on(switch)).$$

     Example 35. The following universal rule is obviously time independent:

     $$\texttt{every}\,(odd(X) \Leftarrow 1\ \ \texttt{is}\, X \bmod 2)$$

   - An **anchored rule** ordinal tree 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 36. The function f has the value 1 for all $X < 5$ over the initial subinterval:

     $$\texttt{within}\, f(X, 1) \Leftarrow \texttt{within}\, X < 5.$$

     which is equivalent to:

     $$(\,\texttt{within}\, f(X, 1)) \Leftarrow X < 5.$$

   - A **level rule** ordinal tree can be applied on any node only at specific level, where the *level* of a node is defined as the number of w-arcs between the node and the root. Such rules will be prefixed by the sequence '$(\,\texttt{all within}\,)^k\,\texttt{all}$' of operators, where k indicates the level at which the rule is applicable.

     Example 37. *(Level 0 rule, causal rule)* 'Whenever it rains the janitor is absent the next day'
     Since this rule is applicable on the level of days, it is called a *level rule*. To indicate that the rule is applicable on *all* days we prefix it by the new operator '`all`':

     $$\texttt{all}\,(\,\texttt{next within}\, absent(janitor) \Leftarrow \texttt{within}\, rain).$$

     This is also an example of a *causal rule* as there are no bodies to the future of the head. Similarly a rule applicable on all hours of all days would be prefixed by '`all within all`'.

     Example 38. *(Level 1 rule)* 'Whenever it rains more than 20 units for a full hour, the buses are late for the next one hour.'

     $$\texttt{all within all}\,(\,\texttt{next within}\, late(bus) \Leftarrow \texttt{within}\, rain(X), X >= 20).$$

4.6 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 prolog implementation is given appendix A.

4.6.1 Loop Unrolling

We can *unroll* loop events, so that the first few occurrences of the repetitive events are made explicit. This will useful in aligning loops of different sizes during temporal resolution. In general the loop:

$$\texttt{loop}\,(k,\,\texttt{within}\,C\,)$$

can also be written as:

$$\texttt{within}\,C,\,\texttt{next}^{\,k}\,\texttt{loop}\,(k,\,\texttt{within}\,C\,)$$

Example 39. *(loop unrolling)* 'The bouncing ball'

$$\begin{aligned}&\texttt{loop}\,(2,\,\texttt{within}\,\mathit{moving}(\mathit{ball},\mathit{down}))\;\Leftarrow\;\texttt{true}\,,\\&\texttt{next loop}\,(2,\,\texttt{within}\,\mathit{moving}(\mathit{ball},\mathit{up}))\;\Leftarrow\;\texttt{true}\,.\end{aligned}$$

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

$$\begin{aligned}&\texttt{within}\,\mathit{moving}(\mathit{ball},\mathit{down})\;\Leftarrow\;\texttt{true}\,,\\&\texttt{next loop}\,(2,\,\texttt{within}\,\mathit{moving}(\mathit{ball},\mathit{up}))\;\Leftarrow\;\texttt{true}\,,\\&\texttt{next next loop}\,(2,\,\texttt{within}\,\mathit{moving}(\mathit{ball},\mathit{down}))\;\Leftarrow\;\texttt{true}\,.\end{aligned}$$

4.6.2 The Aligning Algorithm

The algorithm *align*, when given a temporal query (*Goal*) and a temporal rule or fact (*Rule*), tries to align the intervals of the *Goal* and the *Rule*. If the *Goal* and the *Rule* 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:

	$\texttt{every}\,R$	$\texttt{all}\,R$	$\texttt{within}\,R$	H	$\texttt{within}\,H$	$\texttt{next}\,H$	$\texttt{loop}\,(j,H)$
Q	$E1$	$L1$	$L5$	$A1$	$A5$	$A9$	$A13$
$\texttt{within}\,Q$	$E2$	$L2$	$L6$	$A2$	$A6$	$A10$	$A14$
$\texttt{next}\,Q$	$E3$	$L3$	$L7$	$A3$	$A7$	$A11$	$A15$
$\texttt{loop}\,(i,Q)$	$E4$	$L4$	$L8$	$A4$	$A8$	$A12$	$A16$

Where R denotes a rule, H a head of a rule, and Q a query.

4.7 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?' Such queries never fail. These queries can be classified into two types: *sometime* queries which ask a existential temporal question at a fixed level, and *any-time* queries which asks about an existential temporal event at any level.

A *sometime* query is prefixed by '`some`', and an *anytime* query by a '`any`' prefix. These two operators do not distribute over their list arguments, distributing them in over a list of goals would be weaking the query. However `some` commutes with `next`, and `any` commutes with both `next` and `within`.

Example 40. *(Sometime query)* 'Did it rain on any day?'

$$\Leftarrow \texttt{some within}\, rain.$$

Example 41. *(Compound sometime query)* 'Was there rain on any day, followed immediately by a flood on the next day?'

$$\Leftarrow \texttt{some}\,(\texttt{within}\, rain,\ \texttt{next within}\, flood).$$

Example 42. *(sometime loop query)* 'Did the ball fall from the table and start bouncing?'

$$\begin{aligned}\Leftarrow\ & \texttt{some}\,(fall(ball, table),\\ & \texttt{loop}\,(2,\ \texttt{within}\, moving(ball, down)),\\ & \texttt{next loop}\,(2,\ \texttt{within}\, moving(ball, up))).\end{aligned}$$

Example 43. *(Nested sometime queries)* 'On some day there was more than 10 units of rain, and on some later day there was a flood of at least 20 units?'

$$\Leftarrow \texttt{some}\,(\texttt{within}\, rain(X), X > 10,\ \texttt{next some within}\, flood(Y), Y >= 20).$$

Example 44. *(Any query)* 'Did it rain at any time?'

$$\Leftarrow \texttt{any}\, rain.$$

Example 45. *(Any query within a loop)* 'Does it rain daily?'

$$\Leftarrow \texttt{loop}\,(1,\ \texttt{within any}\, rain).$$

Currently the program does not handle such queries, as it expands `some` and `any` only at the top level. The aligning algorithm should be able to handle this in the future.

A `some` operator in a sometime query is expanded to a string from the set: $\{\texttt{next}^{\,i} : i \geq 0\}$ Similarly an '`any`' operator in a query is systematically expanded to a string from the set: $\{(\texttt{next} \cup \texttt{within})^i : i \geq 0\}$.

When multiple '`some`' and '`any`' appear in a query, they should be expanded by a fair enumeration of the cross product of their individual expansion strings. Currently the expansion of two or more `some` or `any`, is not fair, it is depth first.

4.8 Clock Trees

Till now, ordinal trees divide each interval into an infinite sub-sequence of intervals. 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 a axiom schema, called *finite level axioms*, for each finite level.

Example 46. *(Axiom schema)* There are 24 hours in *all* days:

$$\texttt{all}\,((\,\texttt{within}\,\texttt{next}^{\,n}\phi) \leftrightarrow (\,\texttt{next}^{\,n}\ \ \texttt{div}^{\,24}\,\texttt{within}^{\,n \bmod 24}\phi)), \qquad n \geq 0$$

Example 47. *(Axiom schema)* There are 60 minutes in *all* hours of *all* days:

$$\texttt{all within all}\,(\,\texttt{within}\,\texttt{next}^{\,n}\phi) \leftrightarrow (\,\texttt{next}^{\,n}\ \ \texttt{div}^{\,60}\,\texttt{within}^{\,n \bmod 24}\phi)), \qquad n \geq 0$$

4.9 Conclusion

We have presented a propositional dense time logic which allows us to reason about nested sequences of events. We showed that the logic is decidable in exponential time by a tableaux based method and presented a complete axiomatization for it.

We next looked at an interesting subset of models called ordinal trees. The logic of ordinal trees was also shown to be decidable in exponential time by extending the tableau based decision procedure used in the first part. Ordinal trees capture the temporal notions of persistence and recurrence and also seem to be a good bridge between point based and interval based temporal logics. We further defined a normalization operation on ordinal trees that allowed a more precise characterization of equivalences between dense time formulas.

Ordinal trees being finite can be directly embedded and used as a data structure in an logic programming language, in particular we extended prolog with ordinal trees for temporal logic programming. We define an *ordinal tree clause* to be an ordinal tree with its nodes labeled in a restricted way with prolog clauses. An ordinal tree clause is used to store temporal-facts, causal-rules and temporal-queries. We describe *tree-resolution* on ordinal trees clauses which allows us to answer temporal queries using temporal facts and causal rules.

Future Work

Future work will include axiomatization of POTL and relating it to other dense time logics. We conjecture that the POTL's equivalence problem is decidable and are working on an algorithm for it.

We would also be implementing *clock trees* in DTLP. which can 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', 'seconds' of a clock.

The *loop* operator of DTLP can be generalized to an *reflect* operator, which would be a PDTL's deterministic version of Wolper's [**?**] grammar based operators. We could further investigate the expressive powers of this logic, and its use in temporal logic programming.

Acknowledgements

We thank Rohit Parikh for pointing out that PDTL can be interpreted in Deterministic Propositional Dynamic Logic [**?**], and Madhavan Mukund for pointing out some corrections and PDTL's relation to *S2S* logic of Rabin

[**?**]. We also thank several anonymous referees for useful suggestions.

Bibliography

[AH 90] Alur, R., Henzinger, T., Real Time Logics: Complexity and Expressiveness, in *LICS* 1990, pp 390–401.

[AM 87] Abadi, M., Manna, Z., Temporal logic programming. In *International Conference on Logic Programming*, San Fransisco, CA, 1987, pp 4–16.

[AV 92] Ahmed Mohsin, Venkatesh, G., A Propositional Dense Time Logic: Based on nested sequences, *Technical Report* **TR-65**, Department of Computer Science and Engineering, Indian Institute of Technology, Bombay, India, 1992.

[AV 92] Ahmed Mohsin, Venkatesh, G., Dense Time Logic Programming, *Technical Report* **TR-64**, Department of Computer Science and Engineering, Indian Institute of Technology, Bombay, India, 1992.

[Bau 89] Baudinet, M., Temporal logic programming is complete and expressive. In *16th ACM Symposium on Principles of Programming Languages*, Austin, Tx, 1989, pp 267–279.

[BHP 82] Ben-Ari, M., Halpern, J. Y., Pnueli, A., Deterministic Propositional Dynamic Logic: Finite Models, Complexity and Completeness, *Journal of Computer and System Sciences* **25**, 1982, pp 402–417.

[BPM 83] Ben-Ari, M., Pnueli, A., Manna, Z., The Temporal Logic of Branching Time, *Acta Informatica* **20**, 1983, pp 207–226.

[Bur 84] Burgess, J. P., Basic Tense Logic, in *Handbook of Philosophical Logic* **II**, ed. D. Gabbay and F. Guenther, pp 89–133, D. Reidel 1984.

[Ben 89] van Benthem, J. F. A. K., Time, Logic and Computation, in *Linear Time, Branching Time and Partial Order in Logics and Models of Concurrency*, eds. J.W.deBakker, W.P.deRoever and G.Rozenberg, **LNCS 354** Springer Verlag 1989.

[Ben 83] van Benthem, J. F. A. K., *The Logic of Time*, (D. Reidel, Dordrecht, Holland, 1983).

[CM 81] Clocksin and Mellish, *Programming in Prolog*, Springer 1981.

[Eme 90] Emerson, E. Allen, Temporal and Modal Logic, in *Handbook of Theoretical Computer Science, Volume B*, Ed. J.V. Leeuwen, North Holland, 1990, pp 996–1072.

[Gab 75] Gabbay, D., Decidability results in non-classical logics - I, *Annals of Mathematical Logic* **8** (1975), 237–295.

[GPSS80] Gabbay, D., Pnueli, A., Shelah, S., Stavi, S., The temporal analysis of fairness, in *7 th ACM Symposium on Principles of Programming Languages*, Las Vegas, NE, 1980, pp 163–173.

[Gab 87] Gabbay, D., Modal and Temporal Logic Programming, in *Programming in Temporal Logic*, ed. A. Galton, Academic Press, 1987, pp 195–273.

[Gab 91] Gabbay, D., Modal and Temporal Logic Programming - II, in *Logic Programming*, ed. T. Dodd, Intellect, Oxford, 1991, pp 82–123.

[Har 84] Harel, D., Propositional Dynamic Logic, in *Handbook of Philosophical Logic* **II**, ed. D. Gabbay and F. Guenther, pp 507–544, D. Reidel 1984.

[Lam 90] Lamport, L., Temporal Logic of Actions, *Digital* **TR-57**, 1990.

[LPZ 85] Lichtenstein, O., Pnueli, A., Zuck, L., The Glory of The Past, in *Proc. Conf. Logics of Programs*, **LNCS 193**, (Springer 1985), pp 196–218.

[Mon 69] Monk, D. J., *Introduction to Set Theory*, McGraw Hill, NY, 1969.

[Mos 86] Moszkowski, B. C., *Executing Temporal Logic Programs*, Cambridge University Press, Cambridge, 1986.

[Rab 69] Rabin, M. O., Decidability of Second Order Theories and Automata on Infinite Trees, in *Transactions of AMS*, **141**, Jul 1969, pp 1–35.

[Tho 90] Thomas, W., Automata on Infinite Trees, in *Handbook of Theoretical Computer Science, Volume B*, Ed. J.V. Leeuwen, (North Holland, Amsterdam, 1990), pp 165–186.

[Sho 88] Shoham, Y., *Reasoning about change*, MIT Press 1989.

[SC 82] Sistla, A. P., Clarke, E. M., The complexity of propositional linear temporal logic, in *J.ACM* **(32)3**, 1985, pp 733-749.

[Wol 83] Wolper, P., Temporal logic can be more expressive, *Information and Control* **56**, 1983, pp 72–93.

Appendix A

DTLP Program

```
/*

=====================================================
   Dense Time Logic Programming, Aligning Algorithm.
   Mohsin Ahmed, mosh@cse.iitb.ernet.in,
   08-Jul-92  12:54  Wed
===================================================== */
:-        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, x ),
          op( 20, fx, w ).
/* ( every x a <= b, c ) == every(x(a)<=(b,c)) */
/* ===================================================== */

/* topcall */
dtlp:-
          retractall( somedata(_) ),
          retractall( anydata(_) ),
          retractall( ruledata(_) ),
          repeat,
          write('?= '), read( Input ),
          write('Input: '), display( Input ), nl,
          dtlp( Input, Output ),
          write('Output: '), display( Output ), nl,
          Input == exit.
/* ===================================================== */
/* dtlp( Rule/Query:i, Output:o:true/false/asserted ) */
dtlp( exit, true ):-!.
dtlp( every(Rule), asserted ):-!,
```

```
          write('Universal rule: '), write( every(Rule)), nl,
          asserta( ruledata( every(Rule) ) ).
dtlp( all(Rule), asserted ):-!,
          write('Level rule: '), write( all(Rule) ), nl,
          asserta( ruledata( all(Rule) ) ).
dtlp( Head<=Body, asserted ):-!,
          write('Anchored rule: '), write( Head<=Body ), nl,
          asserta( ruledata( Head<=Body ) ).
dtlp( Query, Output ):-!,
          normalize( Query, 0, NormQuery ),
          write( 'Temporal Query: ' ), write( NormQuery ), nl,
          provelist( anchor, 0, NormQuery, Output ).
/* ======================================================= */
/* provelist( Type:i:anchor/some/any, Depth:i, NormQueryList:i,
               Output:o:true/false )
*/
provelist( Type, Depth, (QueryA,QueryB), Output ):-!,
          provelist( Type, Depth, QueryA, OutputA ),
          provelist( Type, Depth, QueryB, OutputB ),
          and( OutputA, OutputB, Output ).
provelist( Type, Depth, true, true ):-!.
provelist( Type, Depth, Goal, Output ):-
          write('['), write( Depth ), write( '] { Trying: '),
          write( Type ), write(' '), write( Goal ), nl,
          proveone( Type, Depth, Goal, Output ).
/* ======================================================= */
/* proveone( Type:i:anchor/some/any, Depth:i,
               Goal:i:Single-Normalized-Query/Some-Any-Query,
               Output:o:true/false )
*/
proveone( Type, Depth, true, true ):-!.
proveone( Type, Depth, Goal, true ):-
          prolog( Goal ), !.
proveone( Type, Depth, some(Query), Output ):-!,
          provesome( Depth, Query, Output ).
proveone( Type, Depth, any(Query), Output ):-!,
          proveany( Depth, Query, Output ).
proveone( Type, Depth, Query, Output ):-
          ruledata( Rule ),
          align( AlignType, Query, Rule,
                 ToProveQuery, ProvedQuery ), !,
          write('['), write( Depth ), write( ']   ' ),
          write( AlignType ), write(' aligning |- '),
          write( ProvedQuery ), write( ' } '), nl,
          NewDepth is Depth + 1,
          normalize( ToProveQuery, 0, NormQuery ),
          provelist( Type, NewDepth, NormQuery, Output ).
```

```
/* fail & backtrack          if Type = any/some & Goal unproved */
/* donot-fail, just continue if Type = anchor   & Goal unproved */
proveone( anchor, Depth, Goal, false ):-
          write('['), write( Depth ),
          write( ']   Cannot show: '),
          write( Goal ), write( ' } '), nl.
/* ======================================================= */
/* some/any query, systematically enumerated.
   some/any must be at the top level to expand,
   eg. 'x x some true' will not expand
   eg. 'some any w x true' will expand.
*/
/* some(Query), infinite loop */
provesome( Depth, Query, Output ):-
          normalize( Query, 0, NormQuery ),
          provelist( some, Depth, NormQuery, Output ), !,
          write('['), write( Depth ),
          write( ']   dtlp |- Some '),
          write( NormQuery ), write( ' } '), nl.
provesome( Depth, Query, Output ):-
          proveone( some, Depth, some( x(Query) ), Output ).
/* any(Query), infinite loop */
proveany( Depth, Query, Output ):-
          normalize( Query, 0, NormQuery ),
          provelist( any, Depth, NormQuery, Output ), !,
          write('['), write( Depth ),
          write( ']   dtlp |- Any '),
          write( NormQuery ), write( ' } '), nl.
proveany( Depth, Query, Output ):-
          anyplus( Query, QueryPlus ),
          proveone( any, Depth, any( QueryPlus ), Output ).
/* ======================================================= */
/* anyplus( s A:i, s' A:o ), where s, s' in {w,x}^*,
   and s' is enumerated after s, such that:
   value( w ) = 0, value( x ) = 1, and
   value( s ) = binaryvalue( reverse( s ) )     */
anyplus( w(A), x(A) ) :-!.
/* carry */
anyplus( x(A), w(B) ) :-!, anyplus( A, B ).
anyplus( A, w(A) ).
/* ======================================================= */
/* prolog( GoalPrologCall:i ) */
prolog( w(Goal) ):-!, prolog( Goal ).
prolog( x(Goal) ):-!, prolog( Goal ).
prolog( loop(I,Goal) ):-!, prolog( Goal ).
/* var(Goal), Goal = true */
prolog( true ):- !.
```

```
prolog( Goal ):- integer( Goal ),!, fail.
prolog( Goal ):- real( Goal ),!, fail.
prolog( Goal ):-
          call( Goal ),
          write( '      Prolog |- '),
          write( Goal ), write( ' } '), nl.
/* ======================================================= */
/* and( A:i, B:i, C:o ), all in {true,false}     */
and( false, _, false ):-!.
and( _, false, false ):-!.
and( A, true, A ):-!.
and( true, B, B ):-!.
/* ======================================================= */
/* normalize: normalize/flatten/distribute
              operators before aligning.
   normalize( New:i, Old:i, FlatSeqOfTerms:o ):
              until New =Old.
*/
normalize( A, A, A ):-!.

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

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

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

```
          normalize( loop(I,C), loop(I,A), D).

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

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

normalize( A, _, A ).
/* ========================================================= */
/* ============================ ALIGNING ALGO ========== */
/* 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), (AnchoredQ,MovingQ), Proved ):-
          anchor( G,(H<=B), AnchoredQ, MovingQ, Proved ).
/* === case E1,E2,E3,E4 ================ UNIVERSAL RULES */
/* UNIVERSAL( Goal:i, UnivRule:i, MovingQuery:o, Proved:o ) */
universal( w(G), UnivRule, w(B), w(Proved) ):-
          universal( G, UnivRule, B, Proved).
universal( x(G), UnivRule, x(B), x(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( w(G), all(LevelRule), Mq, Proved ):-!,
          level( w(G), LevelRule, Mq, Proved ).
/* === case L3 */
level( x(G), all(LevelRule), x(Mq), x(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( w(G), w(LevelRule), w(Mq), w(Proved) ):-!,
```

```
          level( G, LevelRule, Mq, Proved ).
/* === cases L7,L8 donot occur */
/* === case L1, L5, Q-catchall */
level( Q, all(LevelRule), loop(1,Mq), loop(1,Proved) ):-!,
          level( w(Q), LevelRule, Mq, Proved ).
/* === case ??, 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( w(Q), (w(H) <= B), Aq, w(Mq), w(Pd) ):-
          anchor( Q, (H <= B), Aq, Mq, Pd ).
/* === case A7, A10 */
anchor( x(_), (w(_) <= _), _, _, _ ):-!, fail.
anchor( w(_), (x(_) <= _), _, _, _ ):-!, fail.
/* === case A11 */
anchor( x(Q), (x(H) <= B), Aq, x(Mq), x(Pd) ):-!,
          anchor( Q, (H <= B), Aq, Mq, Pd ).
/* === case A8 */
anchor( loop(I,Q), (w(H) <= B), Aq, (Mq,UnrollQ), Pd ):-!,
          anchor( Q, (w(H) <= B), Aq, Mq, Pd ),
          unroll( I, loop(I,Q), UnrollQ ).
/* === case A12 */
anchor( loop(I,Q), (x(H) <= B), Aq, (Q,Mq), Pd ):-!,
          unroll( I, loop(I,Q), UnrollQ ),
          anchor( UnrollQ, (x(H) <= B), Aq, Mq, Pd ).
/* === case A14 */
anchor( w(Q), (loop(J,H) <= B), Aq, Mq, Pd ):-!,
          anchor( w(Q), (H <= B), Aq, Mq, Pd ).
/* === case A15 */
anchor( x(Q), (loop(J,H) <= B), Aq, Mq, Pd ):-!,
          unroll( J, loop(J,H), UnrollQ ),
          anchor( x(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( w(Q), (H <= B), Aq, w(Mq), w(Pd) ):-!,
          anchor( Q, (H <= B), Aq, Mq, Pd ).
```

```
/* === case A3,  H catchall */
anchor( x(Q), (H <= B), Aq, x(Mq), x(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, (w(H) <= B), Aq, (x(Q),w(Mq)), w(Pd) ):-!,
          anchor( Q, (H <= B), Aq, Mq, Pd ).
/* === case A9,  Q catchall */
anchor( Q, (x(H) <= B), Aq, (w(Q),x(Mq)), x(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( w(Q), (H <= B), Aq, Mq, Pd ),
          anchorcase13( J, loop(J,w(Q)), S ).
/* === case A1,  Q catchall, H catchall */
anchor( Q, (Q <= Aq), Aq, true, Q ).
/* ======================================================= */
/* unroll( N:i, Loop:i, UnRolledLoop:o == x^N(Loop) ) */
unroll( 0, A, A ).
unroll( N, A, x(B) ):-
          N >0, M is N - 1,
          unroll( M, A, B ).
/* ======================================================= */
anchorcase13( 1, Q, true ):-!.
anchorcase13( 2, Q, x(Q) ):-!.
anchorcase13( J, Q, (x(Q),S) ):-
          J > 2, K is J -1,
          anchorcase13( K, x(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, (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 ).
```

```
/* ========================================================

*/
```