A Tableau Method for Interval Temporal Logic *

Howard Bowman and Simon Thompson

Computing Laboratory,
University of Kent at Canterbury,
Canterbury, Kent, CT2 7NF, United Kingdom
Email Contacts: {H.Bowman,S.J.Thompson}@ukc.ac.uk

1 Introduction

Interval Temporal Logic (ITL) is an important class of temporal logic. Early work on the topic was performed by Moskowski [13] with a number of researchers progressing the topic since then, e.g. Hale [8], Kono [9], Duan [6], Cau et al [5], Bowman et al [3] and Thompson [16].

Standard temporal logics are defined over infinite state models, for example, the models for Manna and Pnueli Linear Time Temporal Logic [11] are infinite state sequences. However, in interval temporal logic the model theory is restricted to finite state sequences, called *intervals*.

There are a number of reasons for being interested in such logics. One reason is that interval temporal logic lends itself to execution. This is apparent from Moskowski's work [13]. In addition, a number of interesting and powerful operators arise naturally from ITL. In fact, it is straightforward to derive operators very like the constructs of imperative programming (e.g. assignment, conditionals, iteration etc). This then yields the possibility that abstract specifications and concrete implementations can be realised in the same notation, with refinement mappings between.

An additional aspect of interval temporal logic is that it provides a very simple real-time model in which one unit of time is past when moving from state to state. Consequently, timings can be obtained by measuring interval lengths. The ITL operator $\mathbf{len}(n)$ is used for this purpose. This operator is satisfied by any interval with $n+1$ states (transitions between states are counted rather than numbers of states).

Two operators which are characteristic of interval temporal logic are the Chop operator **;** and the projection operator **proj** . The former of these implements a form of sequential composition; an interval will satisfy,

$$P \,\mathbf{;}\, Q$$

[0]Travel grants to support the research presented here have been provided by the British Council.

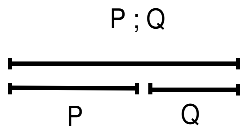

Figure 1: The Chop Operator

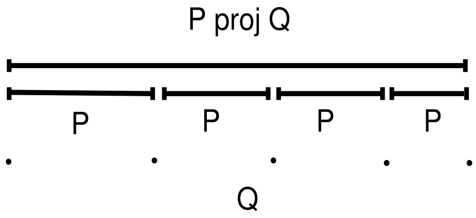

Figure 2: The Projection Operator

if it can be divided into two contiguous sub-intervals such that P holds over the first sub-interval and Q holds over the second. The operator is illustrated in figure 1, where line segments depict intervals.

In contrast, the projection operator yields repetitive behaviour; an interval satisfies,

$$P \ \ \mathbf{proj} \ \ Q$$

if it can be sub-divided into a series of sub-intervals, each of which satisfies P - we call P *the projection formula* - and the new interval formed from the end points of these sub-intervals satisfies Q, which we call *the projected formula*. The operator is illustrated in figure 1, notice the interval that Q holds over is not depicted as a line segment, rather it is the concatenation of the depicted sequence of points (each of which represents a state).

The value of the Chop operator should be self evident, however, the motivation behind projection will perhaps be clarified by some examples. An important use of projection is in deriving iteration constructs. For example, *for* loops and *while* loops can be derived using projection. The *for* loop is defined as:

$$\mathbf{for}\ n\ \mathbf{times\ do}\ P \ \equiv \ P \ \ \mathbf{proj} \ \ \mathbf{len}(n)$$

Notice how $\mathbf{len}(n)$ is used to count the number of iterations, by counting endpoints of sub-intervals. In contrast, the *while* loop is defined as:

$$\mathbf{while}\ P\ \mathbf{do}\ Q \ \equiv \ (P \ \wedge \ Q)^* \ \wedge \ \Box(\mathbf{len}(0) \Rightarrow \neg P)$$

$\Box$ is the ITL operator henceforth, $\Box P$ holds over an interval in which P holds over all suffixes of the interval, and R^* gives arbitrary repetition of R; it is defined as:

$$R^* \equiv R \textbf{ proj True}$$

In the *while* loop P will typically be a *point formula*, satisfaction of which is determined totally from the first state of the satisfying interval. For example, an atomic proposition, p say, when lifted to the interval level, will hold over any interval in which p is true at the first state of the interval. Point formula are called *local formula* in some work [9].

It should also be pointed out that projection has proved a valuable operator in the real-time setting where it can be used to realise temporal abstraction [12] and hence, for example, it can describe speeding up or slowing down real-time presentations [3].

Tableau Methods have been extensively investigated in the standard (infinite) temporal logic setting, e.g. [10]. In addition, there has been some tableau work in the interval temporal logic setting, e.g. Kono [9], but this work is far less mature than that found in the (infinite) temporal logic setting. The reason for this disparity is that ITL operators are in many senses more difficult to deal with. For example, inductive definitions of the until operator of standard temporal logic are straightforward, e.g.

$$P \, \mathcal{U} \, Q \equiv Q \vee (P \wedge \bigcirc(P \, \mathcal{U} \, Q))$$

In contrast, inductive definitions of chop and projection are inherently more complex. In particular, with chop an inductive definition must cope with its first argument evolving when the operator is unfolded. This issue will become clear when we present our normal forms for chop.

Research work that is closest to ours is that by Rosner and Pnueli [15]. The logic considered in their work is a version of standard (infinite) temporal logic which includes the chop operator and they do give a tableau algorithm. However, although this work gives a number of relevant insights it is rather complex and does not handle the projection operator.

In this paper we do present a complete tableau method for interval temporal logic and we include the projection operator. Central to our strategy is the identification of normal forms for all the operators of our logic. In effect, these normal forms give inductive definitions of the ITL operators. Then, in the style of Wolper [17], we define a tableau decision procedure to check satisfiability of our logic. For simplicity of presentation we work in the propositional setting.

Structure of the Paper. Section 2 presents background on interval temporal logic while section 3 presents our normal form. Finally, section 4 describes our tableau algorithm.

2 Interval Temporal Logic

The interval temporal logic that we use is defined in the following subsections. In the context of this paper we will call this logic PITL, for Propositional Interval Temporal Logic.

We begin by defining intervals, which will give the semantic models for our logic.

2.1 Intervals

PITL is defined over finite state sequences. Each sequence is called an interval and $\mathcal{I}$ denotes the set of all possible intervals; $\sigma \in \mathcal{I}$ has the form:

$$\sigma_0, \sigma_1, ..., \sigma_{|\sigma|}$$

where $|\sigma|$ denotes the length of an interval and σ_i denotes the ith state in an interval. By convention the length of an interval is the number of states minus one and all intervals must have at least one state. We use $[\sigma]^i$ to denote the ith prefix of an interval and $(\sigma)^i$ to denote the ith suffix of an interval. Formally,

$$[\sigma]^i = \sigma_0, ..., \sigma_i$$

$$(\sigma)^i = \sigma_i, ..., \sigma_{|\sigma|}$$

Each state is a set containing all the atomic propositions that are true at that state.

2.2 The Logic

The domain of logical propositions is denoted $\mathcal{P}$ and $P \in \mathcal{P}$ is constructed as follows, where $n \in \mathcal{N}$:

$$P \quad ::= \quad p \mid \textbf{False} \mid \neg P \mid P \ \vee \ P \mid \ \bigcirc P \mid \textbf{empty} \mid P \,\textbf{;}\, P \mid P \ \ \textbf{proj} \ \ P$$

Much of this logic will be well known to a reader familiar with interval temporal logic.

- **False**, $\neg$ and $\vee$ are the familiar connectives of classical propositional logic.
- p is chosen from a set of atomic propositions.
- $\bigcirc$ is the (strong) next operator. Thus, $\bigcirc P$ holds if and only if P holds over an interval of length one less than the current interval, resulting from moving one state into the future. In particular, $\bigcirc P$ is **False** on an **empty** interval.
- **empty** holds over any interval of length zero, i.e. which has one state.

- **;** and **proj** are as described in the introduction.

The reader who requires a more detailed discussion of these operators is referred to [13].

Also, we have the following standard derived operators. A wealth of other operators can be defined, see for example [4].

$$
\begin{array}{rcl}
P \Rightarrow Q & \equiv & \neg P \ \vee Q \\
P \ \wedge \ Q & \equiv & \neg(\neg P \ \vee \ \neg Q) \\
\mathbf{len}(0) & \equiv & \mathbf{empty} \\
\mathbf{len}(n+1) & \equiv & \bigcirc \mathbf{len}(n) \\
\circledw P & \equiv & \bigcirc P \ \vee \ \mathbf{empty} \\
\Diamond P & \equiv & \mathbf{True} \ \mathbf{;} \ P \\
\Box P & \equiv & \neg \Diamond \neg P
\end{array}
$$

Thus, in addition, to the standard logical derivations, we have an operator to measure the length of an interval, **len**; a weak next operator, $\circledw$; eventually, $\Diamond$, and henceforth, $\Box$.

2.2.1 Interpreting PITL

Our satisfaction relation, $\models$, interprets PITL formulae over intervals. The notation $\sigma \models P$ denotes that the interval σ satisfies (or models) the formula P.

We interpret PITL propositions in the usual way, by induction over the structure of propositions. We consider the satisfaction rule for each class of construct in turn.

Non-temporal Operators. Atomic propositions are interpreted as follows:

$$\sigma \models p \ \text{ iff } \ p \in \sigma_0$$

i.e. a proposition p holds over any interval in which p holds at the first state of the interval. In this way satisfaction of atomic propositions is lifted to the interval level. In propositional interval temporal logic atomic propositions are the basic source of point formula.

The following rules reflect the standard interpretation of **False** and $\Rightarrow$.

$$\sigma \not\models \mathbf{False}$$

$$\sigma \models \neg P \ \text{ iff } \ \sigma \not\models P$$

$$\sigma \models P_1 \ \vee \ P_2 \ \text{ iff } \ \sigma \models P_1 \text{ or } \sigma \models P_2$$

Interval Temporal Operators. **empty** holds over an interval of length 0.

$$\sigma \models \textbf{empty} \text{ iff } |\sigma| = 0$$

(Strong) next moves the start of the interval under consideration one state into the future.

$$\sigma \models \bigcirc P \text{ iff } (\sigma)^1 \models P$$

Chop subdivides the given interval into two sub-intervals the first of which satisfies P_1 and the second of which satisfies P_2.

$$\sigma \models P_1 ; P_2 \text{ iff } \exists k \in \mathcal{N} \ (k \leq |\sigma| \text{ and } [\sigma]^k \models P_1 \text{ and } (\sigma)^k \models P_2)$$

The two sub-intervals arising from chop have one common state, the kth state in the above definition. Thus, the last state of the sub-interval over which P_1 holds is the first state of the sub-interval over which P_2 holds.

The semantics of projection are somewhat more complex.

$$\begin{array}{ll} \sigma \models P_1 \ \textbf{proj} \ P_2 \text{ iff} & \exists m \in \mathcal{N} \text{ and } \exists \tau_0, \tau_1, ..., \tau_m \in \mathcal{N} \\ & (0 = \tau_0 < \tau_1 < ... < \tau_m = |\sigma| \text{ and} \\ & \forall j < m \ (([\sigma]^{\tau_{j+1}})^{\tau_j} \models P_1 \text{ and} \\ & \sigma_{\tau_0} \sigma_{\tau_1} ... \sigma_{\tau_m} \models P_2) \end{array}$$

This rule states that for an interval to satisfy P_1 **proj** P_2 there must exist a sequence of m increasing points [1] (or states) in the interval, $\tau_0, \tau_1, ..., \tau_m$, such that the first and last points bound the interval. This sequence of points effectively divides the interval into a series of m sub-intervals, each of which comprises the states between τ_j and τ_{j+1}. We require that P_1 holds over each of these sub-intervals and, in addition, that P_2 holds when $\sigma_{\tau_0}, \sigma_{\tau_1}, ..., \sigma_{\tau_m}$ is viewed as an interval.

Satisfaction and Validity. We define that an interval σ *satisfies* a proposition P if and only if $\sigma \models P$. In addition, in the usual way, we state that P is *valid* if and only if for all σ in $\mathcal{I}$, $\sigma \models P$. If P is valid we write $\models P$.

3 Normal Forms in Interval Temporal Logic

3.1 A Normal Form for PITL

The use of normal forms in temporal logic is now relatively common, e.g. [1], [7]. Furthermore, a number of authors have proposed normal forms for interval temporal logic, e.g. [13], [9], [6].

[1] It is common only to constrain this sequence to be non-decreasing, e.g. [13], however, this generates a number of pathological cases that we manage to avoid in our definition.

Our normal form is based on that of previous workers; it has the general format:

$$(\textbf{empty} \ \wedge \ P_e) \ \vee \ \bigvee_i (P_i \ \wedge \ \bigcirc P'_i)$$

where P_e and P_i are point formulas and P'_i is a general PITL formula. The left disjunct characterises under what circumstances a formula can be satisfied over an empty interval, while the second disjunct characterises the possible ways in which a formula can be satisfied over an interval of length greater than zero, i.e. a point property must hold at the initial state and then an arbitrary property must hold over the remainder of the interval.

You should also note that this normal form embodies a recipe for evaluating PITL formula. The first disjunct embodies a base case, i.e. what must hold of a one state interval, while the second disjunct embodies an inductive step, i.e. one of the P_i's must hold now and the associated P'_i must hold from the next state onwards.

We claim that any arbitrary PITL formula can be mapped into this normal form. The next subsection verifies this claim.

3.2 Inductive Definition of the Normal Form

We proceed through PITL highlighting how each construct of the logic can be expressed in our normal form. We there by give an inductive definition of the normal form. So, assume Q is an arbitrary formula of PITL and assume that P_1, P_2 and P are already in normal form, as follows:

$$P_1 \ \equiv \ (\textbf{empty} \ \wedge \ P_{e,1}) \ \vee \ \bigvee_i (P_{i,1} \ \wedge \ \bigcirc P'_{i,1})$$

$$P_2 \ \equiv \ (\textbf{empty} \ \wedge \ P_{e,2}) \ \vee \ \bigvee_j (P_{j,2} \ \wedge \ \bigcirc P'_{j,2})$$

$$P \ \equiv \ (\textbf{empty} \ \wedge \ P_e) \ \vee \ \bigvee_i (P_i \ \wedge \ \bigcirc P'_i)$$

3.2.1 Non-temporal Operators

Atomic Propositions. If Q is an atomic proposition, p, then,

$$Q \ \equiv \ p \ \equiv \ (\textbf{empty} \ \wedge \ p) \ \vee \ (p \ \wedge \ \bigcirc\textbf{True})$$

Thus, an atomic proposition holds over an interval if either the interval has only one state and p holds at this state or the interval is of an arbitrary size greater than zero and p holds at the first state. This clarifies the nature of atomic propositions as point formulae.

False. If Q is **False** then,

$$Q \ \equiv \ \textbf{False} \ \equiv \ (\textbf{empty} \ \wedge \ \textbf{False}) \ \vee \ (\textbf{False} \ \wedge \ \bigcirc\textbf{True})$$

Or. If Q is $P_1 \ \vee \ P_2$ then,

$$Q \equiv P_1 \vee P_2 \equiv (\textbf{empty} \wedge (P_{e,1} \vee P_{e,2})) \vee \bigvee_i (P_{i,1} \wedge \bigcirc P'_{i,1}) \vee \bigvee_j (P_{j,2} \wedge \bigcirc P'_{j,2})$$

And. However, if Q is $P_1 \wedge P_2$ then,

$$Q \equiv P_1 \wedge P_2 \equiv (\textbf{empty} \wedge (P_{e,1} \wedge P_{e,2})) \vee \bigvee_{i,\,j} ((P_{i,1} \wedge P_{j,2}) \wedge \bigcirc (P'_{i,1} \wedge P'_{j,2}))$$

3.2.2 Partitioned normal forms

Before we cover the normal form for negations of propositions, we take a detour into elementary propositional calculus. We say that a collection of formulas $\langle P_i \rangle_i$ *partitions* **True** when

- $\bigvee_i P_1 \equiv \textbf{True}$ – the partition is exhaustive; and
- $\bigvee_{i \neq j} (P_i \wedge P_j) \equiv \textbf{False}$ – the partition is exclusive.

It is an exercise to show that in the case that $\langle P_i \rangle_i$ partitions **True** then,

$$\bigvee_i (P_i \wedge Q_i) \equiv \bigwedge_i (P_i \Rightarrow Q_i)$$

and then it is straightforward to see that,

$$\neg \bigwedge_i (P_i \Rightarrow Q_i) \equiv \bigwedge_i (P_i \Rightarrow \neg Q_i)$$

and so in the partitioned case,

$$\neg \bigvee_i (P_i \wedge Q_i) \equiv \bigvee_i (P_i \wedge \neg Q_i)$$

We call a normal form

$$(\textbf{empty} \wedge P_e) \vee \bigvee_i (P_i \wedge \bigcirc P'_i)$$

partitioned when $\langle P_i \rangle_i$ partitions **True**.

We can turn an arbitrary normal form into partitioned form. For notational simplicity choose the case in which there are two disjuncts,

$$(P_1 \wedge \bigcirc P'_1) \vee (P_2 \wedge \bigcirc P'_2)$$

This generates the partition

$$\langle\, (P_1 \wedge P_2),\ (P_1 \wedge \neg P_2),\ (\neg P_1 \wedge P_2),\ (\neg P_1 \wedge \neg P_2) \,\rangle$$

and the non-empty part of the normal form becomes

$$\begin{aligned}(P_1 \wedge \bigcirc P'_1) \vee (P_2 \wedge \bigcirc P'_2) \quad \equiv \quad & ((P_1 \wedge P_2) \wedge \bigcirc (P'_1 \vee P'_2)) \vee \\ & ((P_1 \wedge \neg P_2) \wedge \bigcirc P'_1) \vee \\ & ((\neg P_1 \wedge P_2) \wedge \bigcirc P'_2) \vee \\ & ((\neg P_1 \wedge \neg P_2) \wedge \bigcirc \textbf{False})\end{aligned}$$

Some logical operations (including conjunction and projection) preserve the property of being partitioned; others, notably disjunction and chop, do not. Using the technique outlined here we can put normal forms into partitioned form should that be necessary or convenient.

3.2.3 Negation

Using the material in the previous section, if P is in partitioned normal form then

$$Q \equiv \neg P \equiv (\mathbf{empty} \wedge \neg P_e) \vee \bigvee_i (P_i \wedge \bigcirc \neg P_i')$$

Any normal form can be transformed into partitioned normal form, and so can be negated by this method.

3.2.4 Temporal Operators

Empty. If Q is **empty** then,

$$Q \equiv \mathbf{empty} \equiv (\mathbf{empty} \wedge \mathbf{True}) \vee (\mathbf{False} \wedge \bigcirc \mathbf{True})$$

Thus, **empty** holds over any arbitrary "one state" interval, but fails to hold over any "larger" interval.

Next. If Q is $\bigcirc P$ then,

$$Q \equiv \bigcirc P \equiv (\mathbf{empty} \wedge \mathbf{False}) \vee \bigvee_i (\mathbf{True} \wedge \bigcirc P)$$

Thus, $\bigcirc P$ cannot hold over an empty interval (this is because it is a strong next), while it puts no constraints on the first state of a non-empty interval and requires that P holds over the remainder of the interval.

Chop. If Q is $P_1 \mathbin{\mathbf{;}} P_2$ then,

$$\begin{aligned} Q \equiv P_1 \mathbin{\mathbf{;}} P_2 \quad \equiv \quad & (\mathbf{empty} \wedge (P_{e,1} \wedge P_{e,2})) \vee \\ & \bigvee_i (P_{i,1} \wedge \bigcirc(P_{i,1}' \mathbin{\mathbf{;}} P_2)) \vee \\ & \bigvee_j ((P_{e,1} \wedge P_{j,2}) \wedge \bigcirc P_{j,2}') \end{aligned}$$

The normal form embodies three cases.

- The first disjunct gives the condition under which the chop formula holds over an empty interval, namely when the two constituent formulas hold over the empty interval;
- the second disjunct is the condition in which P_1 holds over a non-empty interval, and P_2 holds over the remainder; and,
- the third disjunct embodies the case where P_1 holds over an empty interval and P_2 holds over the entire (non-empty) interval.

This explanation forms an informal argument for the correctness of the normal form of the chop; it is formalised in the next section.

The negation of chop is given by transforming the normal form given here into the partitioned normal form, as explained above.

Projection. If Q is $(P_1 \ \mathbf{proj} \ P_2)$ then,

$$\begin{aligned} Q \ \equiv \ P_1 \ \mathbf{proj} \ P_2 \ \equiv \ & (\mathbf{empty} \ \wedge \ P_{e,2}) \ \vee \\ & \bigvee_i \bigvee_j ((P_{i,1} \ \wedge \ P_{j,2}) \ \wedge \ \bigcirc(P'_{i,1} \ ; P_1 \ \mathbf{proj} \ P'_{j,2})) \end{aligned}$$

The normal form embodies two cases.

- The first disjunct gives the condition under which the projection holds over an empty interval, namely when P_2 holds over the interval.
- The other disjunct gives the condition where $(P_1 \ \mathbf{proj} \ P_2)$ holds over a non-empty interval. A choice of initial-state conditions of P_1 and P_2, such as $P_{i,1}$ and $P_{j,2}$, must hold at the first state; over the remaining part of the interval a chop has to hold. First the remainder of the P_1 condition – $P'_{i,1}$ – must hold, then the derivative of the projection – $(P_1 \ \mathbf{proj} \ P'_{j,2})$ – must be valid.

Note that if P_1 and P_2 are in partitioned normal form then this normal form will also be partitioned. The next section formalises this explanation.

3.3 Validity of the Normal Forms

In this section we argue that the normal forms given in the previous section are semantically equivalent to the original formulas; we begin by proving a lemma about the satisfaction of formulas in normal form.

Lemma 1 *If P is the formula*

$$(\mathbf{empty} \ \wedge \ P_e) \ \vee \ \bigvee_i (P_i \ \wedge \ \bigcirc P'_i)$$

then $\sigma \models P$ if and only if either

- $|\sigma| = 0$ *and* $\sigma_0 \models P_e$, *or*
- $|\sigma| > 0$ *and for some* i, $\sigma_0 \models P_i$ *and* $(\sigma)^1 \models P'_i$

Proof
This is a straightforward exercise in checking the semantic definition. □

Theorem 1 *The normal forms derived above are logically equivalent to the original formulas.*

Proof
The proof works by induction over the construction of the formulas; we leave the simpler cases as exercises but in Lemmas 2 and 3 we check the result for chop and projection. □

Lemma 2 *If*

$$P_1 \equiv (\textbf{empty} \wedge P_{e,1}) \vee \bigvee_i (P_{i,1} \wedge \bigcirc P'_{i,1})$$

and

$$P_2 \equiv (\textbf{empty} \wedge P_{e,2}) \vee \bigvee_j (P_{j,2} \wedge \bigcirc P'_{j,2})$$

then for all intervals σ

$$\sigma \models P_1 \,;\, P_2$$

if and only if

$$\begin{aligned} \sigma \quad \models \quad & (\textbf{empty} \wedge (P_{e,1} \wedge P_{e,2})) \vee \\ & \bigvee_i (P_{i,1} \wedge \bigcirc(P'_{i,1} \,;\, P_2)) \vee \\ & \bigvee_j ((P_{e,1} \wedge P_{j,2}) \wedge \bigcirc P'_{j,2}) \qquad\qquad (RHS) \end{aligned}$$

Proof

We do a case analysis over the length of σ. Suppose first that $|\sigma| = 0$, then

$$\begin{aligned} \sigma \models P_1 \,;\, P_2 \quad & \text{iff} \quad \sigma \models P_1 \text{ and } \sigma \models P_2 \\ & \qquad \text{(by the semantics of chop)} \\ & \text{iff} \quad \sigma_0 \models P_{e,1} \text{ and } \sigma_0 \models P_{e,2} \\ & \qquad \text{(by Lemma 1)} \\ & \text{iff} \quad \sigma \models RHS \\ & \qquad \text{(by Lemma 1 again)} \end{aligned}$$

Now consider the case that $|\sigma| > 0$. Then

$$\sigma \models P_1 \,;\, P_2 \quad \text{iff} \quad \exists k \leq |\sigma|,\ [\sigma]^k \models P_1 \text{ and } (\sigma)^k \models P_2$$

If this k is 0 then

$$\begin{aligned} & \text{iff} \quad [\sigma]^0 \models P_1 \text{ and } \sigma \models P_2 \\ & \qquad \text{(since } (\sigma)^0 \text{ is } \sigma) \\ & \text{iff} \quad \sigma_0 \models P_{1,e} \text{ and for some } j,\ \sigma_0 \models P_{j,2} \text{ and } (\sigma)^1 \models P'_{j,2} \\ & \text{iff} \quad \text{for some } j,\ \sigma_0 \models P_{1,e} \wedge P_{j,2} \text{ and } (\sigma)^1 \models P'_{j,2} \\ & \Rightarrow \quad \sigma \models \text{last clause of } (RHS) \\ & \Rightarrow \quad \sigma \models (RHS) \end{aligned}$$

If this $k > 0$ then

$$\begin{aligned} & \text{iff} \quad [\sigma]^k \models P_1 \text{ and } (\sigma)^k \models P_2 \\ & \text{iff} \quad \text{(since this } k \text{ is positive), for some } i, \\ & \qquad \sigma_0 \models P_{1,i},\ ([\sigma]^k)^1 \models P'_{1,i} \\ & \qquad \text{and } (\sigma)^k \models P_2 \\ & \text{iff} \quad \sigma_0 \models P_{1,i} \text{ and } (\sigma)^1 \models (P'_{1,i} \,;\, P_2) \\ & \Rightarrow \quad \sigma \models \text{second clause of } (RHS) \\ & \Rightarrow \quad \sigma \models (RHS) \end{aligned}$$

In the other direction, suppose that $\sigma \models (RHS)$. Still under the supposition that $|\sigma| > 0$ by 1 either

- $\sigma_0 \models P_{1,i}$ and $(\sigma)^1 \models (P'_{1,i} \,\mathbf{;}\, P_2)$ for some i, or
- $\sigma_0 \models P_{1,e} \wedge P_{2,j}$ and $(\sigma)^1 \models P'_{2,j}$.

In the first case, suppose that $[(\sigma)^1]^k \models P'_{1,i}$ and $((\sigma)^1)^k \models P_2$. Then $[\sigma]^{k+1} \models P_1$ and $(\sigma)^{k+1} \models P_2$, so $\sigma \models (P_1 \,\mathbf{;}\, P_2)$.

In the second case, $[\sigma]^0 \models P_1$ and $(\sigma)^0 \models P_2$; again $\sigma \models (P_1 \,\mathbf{;}\, P_2)$. We have therefore the result that $\sigma \models (P_1 \,\mathbf{;}\, P_2)$ in all cases. This concludes the proof in the backwards direction for the $|\sigma| > 0$ case, and indeed the proof itself.

□

Lemma 3 *Under the same hypotheses as Lemma 2, for all intervals σ*

$$\sigma \models P_1 \;\; \mathbf{proj} \;\; P_2$$

if and only if

$$\begin{array}{rcl} \sigma & \models & (\mathbf{empty} \;\wedge\; P_{e,2}) \;\vee \\ & & \bigvee_i \bigvee_j ((P_{i,1} \;\wedge\; P_{j,2}) \;\wedge\; \bigcirc(P'_{i,1} \,\mathbf{;}\, P_1 \;\; \mathbf{proj} \;\; P'_{j,2})) \end{array}$$

Proof

The semantics of projection are given thus

$$\begin{array}{ll} \sigma \models P_1 \;\; \mathbf{proj} \;\; P_2 \;\; \text{iff} & \exists m \in \mathcal{N} \text{ and } \exists \tau_0, \tau_1, ..., \tau_m \in \mathcal{N} \\ & (0 = \tau_0 < \tau_1 < ... < \tau_m = |\sigma| \text{ and} \\ & \forall j < m \; ([\sigma]^{\tau_{j+1}})^{\tau_j} \models P_1 \text{ and} \\ & \langle\, \sigma_{\tau_0} \sigma_{\tau_1} ... \sigma_{\tau_m} \,\rangle \models P_2) \end{array}$$

The case that $m = 0$ is precisely the case that $|\sigma| = 0$, since the requirements

$$0 = \tau_0 < \tau_1 < ... < \tau_m = |\sigma|$$

can only be satisfied by m being 0 and $|\sigma| = 0$. This means that if $|\sigma| = 0$

$$\begin{array}{lll} \sigma \models (P_1 \;\; \mathbf{proj} \;\; P_2) & \text{iff} & \langle\, \sigma_{\tau_0} \sigma_{\tau_1} ... \sigma_{\tau_m} \,\rangle \models P_2 \\ & \text{iff} & \sigma \models P_2 \\ & \text{iff} & \sigma_0 \models P_{e,2} \end{array}$$

This observation shows that the ‘empty’ part of the normal form for $(P_1 \;\; \mathbf{proj} \;\; P_2)$ is correct. Now we consider the case that $m > 0$.

In this case we have

$$([\sigma]^{\tau_1})^{\tau_0} \models P_1 \tag{1}$$

and

$$\langle\, \sigma_{\tau_0}\sigma_{\tau_1}...\sigma_{\tau_m}\,\rangle \models P_2 \tag{2}$$

Since τ_0 is 0 formula (1) simplifies to

$$[\sigma]^{\tau_1} \models P_1 \tag{3}$$

Since $\tau_1 > \tau_0$ and $|\sigma| \neq 0$ both $[\sigma]^{\tau_1}$ and $\langle\, \sigma_{\tau_0}\sigma_{\tau_1}...\sigma_{\tau_m}\,\rangle$ are non-empty. By Lemma 1 and the hypotheses on P_1 and P_2,

$$\begin{array}{lll} [\sigma]^{\tau_1} \models P_1 & \text{iff} & \text{for some } i\ ([\sigma]^{\tau_1})_0 \models P_{i,1} \text{ and} \\ & & ([\sigma]^{\tau_1})^1 \models P'i,1 \end{array}$$

$$\begin{array}{lll} \langle\, \sigma_{\tau_0}\sigma_{\tau_1}...\sigma_{\tau_m}\,\rangle \models P_2 & \text{iff} & \text{for some } j,\ \sigma_0 \models P_{j,2} \text{ and} \\ & & \langle\, \sigma_{\tau_1}...\sigma_{\tau_m}\,\rangle \models P'_{j,2} \end{array}$$

This means that (2) and (3) hold exactly when for some i and j

$$\sigma_0 \models P_{i,1} \wedge P_{j,2} \tag{4}$$

and

$$([\sigma]^{\tau_1})^1 \models P'_{i,1}$$

$$\langle\, \sigma_{\tau_1}...\sigma_{\tau_m}\,\rangle \models P'_{j,2}$$

and also for appropriate $j > 0$,

$$([\sigma]^{\tau_{j+1}})^{\tau_j} \models P_1$$

Putting these together we have (4) and

$$(\sigma)^1 \models P_{i,1}\ \mathbf{;}\,(P_1\ \ \mathbf{proj}\ \ P_{j,2}) \tag{5}$$

which shows that for a non-empty σ, $\sigma \models P_1\ \ \mathbf{proj}\ \ P_2$ if and only if

$$\sigma \models \bigvee_i \bigvee_j ((P_{i,1}\ \wedge\ P_{j,2})\ \wedge\ \bigcirc(P'_{i,1}\ \mathbf{;}\, P_1\ \ \mathbf{proj}\ \ P'_{j,2}))$$

This concludes the part of the proof concerning non-empty intervals, and thus the proof itself. $\square$

3.4 Examples

This subsection illustrates our normal form by working through a number of examples.

3.4.1 Example 1

Assume that the formula P is in normal form, i.e.

$$P \equiv (\textbf{empty} \wedge P_e) \vee \bigvee_i (P_i \wedge \bigcirc P_i')$$

and consider the derived formula eventually P, $\Diamond P$. The standard derivation of eventually is from chop,

$$\Diamond P \quad \equiv \quad \textbf{True ;}\, P$$

Now, using the normal form rule for chop, $\Diamond P$ can be placed in normal form as follows:

$$\begin{array}{lll} \Diamond P & \equiv & (\textbf{empty} \wedge P_e) \\ & & \vee\, (\textbf{True} \wedge \bigcirc(\textbf{True ;}\, P)) \\ & & \vee \bigvee_i (P_i \wedge \bigcirc P_i') \end{array}$$

Notice that using associativity and symmetry of disjunction and **True** as unit of conjunction we can reduce this to:

$$\Diamond P \quad \equiv \quad P \vee \bigcirc \Diamond P$$

which is a more common "inductive" interpretation of $\Diamond$.

3.4.2 Example 2

Assuming again that P is in normal form,

$$P \equiv (\textbf{empty} \wedge P_e) \vee \bigvee_i (P_i \wedge \bigcirc P_i')$$

and now also assume that $\langle P_i \rangle_i$ is a partition.

Consider always P (the dual of eventually P) which is derived in standard fashion as:

$$\Box P \quad \equiv \quad \neg(\textbf{True ;}\, \neg P)$$

Now, let us first consider (**True ;** $\neg P$) in normal form, this takes a similar form to our last example:

$$\begin{array}{lll} \textbf{True ;}\, \neg P & \equiv & (\textbf{empty} \wedge \neg P_e) \\ & & \vee\, (\textbf{True} \wedge \bigcirc(\textbf{True ;}\, \neg P)) \\ & & \vee \bigvee_i (P_i \wedge \bigcirc(\neg P_i')) \end{array}$$

However, before we can negate this normal form and obtain $\Box P$, this must be put into partitioned form, as follows:

$$\begin{array}{rcl} \textbf{True} \,;\, \neg P & \equiv & (\textbf{empty} \;\wedge\; \neg P_e) \\ & & \vee \bigvee_i (P_i \;\wedge\; \bigcirc(\neg P_i' \;\vee\; \textbf{True} \,;\, \neg P))) \end{array}$$

and can now consider $\Box P$:

$$\begin{array}{rcl} \neg(\textbf{True} \,;\, \neg P) & \equiv & (\textbf{empty} \;\wedge\; \neg\neg P_e) \\ & & \vee \bigvee_i (P_i \;\wedge\; \bigcirc\neg(\neg P_i' \;\vee\; \textbf{True} \,;\, \neg P))) \end{array}$$

Assuming that $\textcircled{w}$ is the weak next operator, the following derivation gives a more familiar expansion of $\Box P$.

$$\begin{array}{ll} & \neg(\textbf{True} \,;\, \neg P) \\ \equiv & \{ \text{ manipulation of above normal form } \} \\ & (\textbf{empty} \;\wedge\; P_e) \;\vee\; \bigvee_i (P_i \;\wedge\; \bigcirc(P_i' \;\wedge\; \neg(\textbf{True} \,;\, \neg P)) \\ \equiv & \{ \text{ properties of next } \} \\ & (\textbf{empty} \;\wedge\; P_e) \;\vee\; \bigvee_i ((P_i \;\wedge\; \bigcirc P_i') \;\wedge\; \bigcirc\neg(\textbf{True} \,;\, \neg P)) \\ \equiv & \{ \text{ normal form } \} \\ & (P \;\wedge\; \textbf{empty}) \;\vee\; (P \;\wedge\; \bigcirc\Box P) \\ \equiv & \{ \textbf{ empty } \;\vee\; \bigcirc\Box P \;\equiv\; \textcircled{w}\Box P \} \\ & P \;\wedge\; \textcircled{w}\Box P \end{array}$$

4 The Tableau Method

In this section we present a tableau decision procedure for checking the satisfiability of PITL formulae. The decision procedure is influenced by the tableau procedures developed for standard temporal logic, e.g. [2] [14] and [17]. Our work particularly builds from the tableau method defined by P. Wolper [17].

The tableau decision algorithm provides a systematic search for a model that satisfies a particular formula. It is a graph construction (and then reduction) algorithm. Nodes of the graph are sets of PITL formula; these are said to *label the node*. The interpretation of such a labelling is that for the node to be satisfied, it must be possible for all the formulas in the set to be satisfied together. Edges in the graph represent steps to satisfaction of the formulas of a node.

These steps are made according to a set of *tableau (decomposition) rules* which embody how satisfaction of a particular formula can be decomposed into satisfaction of its constituent formulae. Branches in the tableau reflect disjunctive choices.

An important element of the tableau rules is that they subdivide the requirements imposed by temporal formulas into requirements on the present (i.e. the first state of an interval) and requirements on the remainder of the interval, the latter being embodied in a formula of the form $\bigcirc P$. This is in fact exactly the subdivision in our normal form rules. Accordingly our normal forms will play a central role in these tableau rules.

A set of *graph construction* rules build the graph using the tableau rules. In particular, the graph construction rules generate time passing steps in the tableau that unravel next formulas.

The final stage of the tableau algorithm is a *graph reduction* algorithm which systematically removes unsatisfiable nodes in the graph. A formula can then be deduced to be satisfiable if it appears in the reduced graph.

We will work through the different stages of the tableau algoorithm in turn, but first some notation.

Notation. The following notation is related to that used in [17].

- The formula being evaluated is called the *initial formula.*
- The *labelling of node* n is denoted T_n.
- A formula P is called *elementary* if
  1. it is an atomic proposition or the negation of an atomic proposition;
  2. it is **empty** or $\neg$**empty**; or
  3. it has $\bigcirc$ as the main connective.
- A node containing only elementary formulae is called a *state.*
- A *pre-state* is a node that is either initial or the immediate son of a state.

Tableau Rules. The *tableau rules* (also called *decomposition rules*) drive the reduction of formulae into their components. They map a formula into a set of sets of formulas, i.e. they have the form:

$$P \longrightarrow \{\ S_1, S_2, ..., S_n\ \}$$

where S_i is a set of PITL formulas. The interpretation that such rules embody is that P can be satisfied if there exists a j $(1 \leq j \leq n)$ such that all formulas in S_j can together be satisfied.

Our first set of rules are standard and are unchanged from those in [17]:

$$\begin{array}{rcl}
\neg\neg P & \longrightarrow & \{\{\ P\ \}\} \\
P_1\ \wedge\ P_2 & \longrightarrow & \{\{\ P_1\ ,\ P_2\ \}\} \\
\neg(P_1\ \wedge\ P_2) & \longrightarrow & \{\{\ \neg P_1\ \}, \{\ \neg P_2\ \}\} \\
P_1\ \vee\ P_2 & \longrightarrow & \{\{\ P_1\ \}, \{\ P_2\ \}\} \\
\neg(P_1\ \vee\ P_2) & \longrightarrow & \{\{\ \neg P_1\ ,\ \neg P_2\ \}\}
\end{array}$$

We also need the following rule which enables the next operator to be moved to the top level through negation. Notice that since we are using a strong next operator, $\bigcirc P$ will fail over any **empty** interval.

$$\neg \bigcirc P \quad \longrightarrow \quad \{\{\ \mathbf{empty}\ \}, \{\ \bigcirc \neg P\ \}\}$$

The remaining rules are direct extrapolations from our normal form rules. They all assume that P_1 and P_2 are in the normal forms we highlighted earlier. The next two rules decompose chop and projection:

$$\begin{array}{rcl}
P_1 \ \mathbf{;}\ P_2 & \longrightarrow & \{\{\ (\mathbf{empty}\ \wedge\ (P_{e,1}\ \wedge\ P_{e,2}))\ \vee \\
& & \bigvee_i (P_{i,1}\ \wedge\ \bigcirc(P'_{i,1}\ \mathbf{;}\ P_2))\ \vee \\
& & \bigvee_j ((P_{e,1}\ \wedge\ P_{j,2})\ \wedge\ \bigcirc P'_{j,2})\ \}\} \\
P_1\ \ \mathbf{proj}\ \ P_2 & \longrightarrow & \{\{\ (\mathbf{empty}\ \wedge\ P_{e,2})\ \vee \\
& & \bigvee_i \bigvee_j ((P_{i,1}\ \wedge\ P_{j,2})\ \wedge\ \bigcirc(P'_{i,1}\ \mathbf{;}\ P_1\ \ \mathbf{proj}\ \ P'_{j,2}))\ \}\}
\end{array}$$

$\neg(P_1 \ \mathbf{;}\ P_2)$ and $\neg(P_1\ \ \mathbf{proj}\ \ P_2)$ can be obtained by the negation of normal forms previously highlighted. In fact, negation of all the operators could be obtained from negation of the normal forms, but in the case of the propositional operators and strong next it is more straightforward to handle negation outside the normal form expansion.

The following two rules encapsulate the required normal form negation. Both rules assume that $\langle P_{i,1} \rangle_i$ and $\langle P_{i,2} \rangle_i$ are in partition form.

$$\begin{array}{rcl}
\neg(P_1 \ \mathbf{;}\ P_2) & \longrightarrow & \{\{\ (\mathbf{empty}\ \wedge\ (\neg P_{e,1}\ \vee\ \neg P_{e,2}))\ \vee \\
& & \bigvee_i ((\neg P_{e,1}\ \wedge\ P_{i,1})\ \wedge\ \bigcirc\neg(P'_{i,1}\ \mathbf{;}\ P_2))\ \vee \\
& & \bigvee_i \bigvee_j ((P_{e,1}\ \wedge\ P_{i,1}\ \wedge\ P_{j,2})\ \wedge\ \bigcirc(\neg P'_{j,2}\ \wedge\ \neg(P'_{i,1}\ \mathbf{;}\ P_2)))\ \}\}
\end{array}$$

$$\begin{array}{rcl}
\neg(P_1\ \ \mathbf{proj}\ \ P_2) & \longrightarrow & \{\{\ (\mathbf{empty}\ \wedge\ \neg P_{e,2})\ \vee \\
& & \bigvee_i \bigvee_j ((P_{i,1}\ \wedge\ P_{j,2})\ \wedge\ \bigcirc\neg(P'_{i,1}\ \mathbf{;}\ P_1\ \ \mathbf{proj}\ \ P'_{j,2}))\ \}\}
\end{array}$$

We have the following lemma.

Lemma 4 *All the decomposition rules preserve satisfaction. Thus, for a rule of the form,*

$$P \longrightarrow \{\ S_i\ \}$$

the following holds:

$$\sigma \models P \ \ \textit{iff}\ \ \exists j\ .\ \sigma \models \bigwedge S_j \qquad \textit{where}\ \bigwedge\{P_1, ..., P_n\} = P_1\ \wedge\ ...\ \wedge\ P_n$$

Proof
The propositional rules and the rule for $\neg\bigcirc P$ are straightforward. The rules for **;**, **proj** , $\neg$ **;** and $\neg$ **proj** follow from theorem 1 where validity of the normal forms was verified. $\square$

The Graph Construction. The tableau graph is constructed using the following rules:

1. Where P is the formula to test, label the *initial node* with { P , **True ; empty** } and apply steps (2) and (3) repeatedly until neither can be applied to any node in the graph.

   Note that when we create a son node in rules (2) and (3) with labelling T, we only add a new node if a node labelled T does not already exist, otherwise we add an edge back to the original node.

2. If a node n contains a non-elementary formula P and the tableau rule for P is $P \longrightarrow \{S_i\}$, then,

   For all i create a son of n labelled:
   $(T_n - \{P\}) \cup S_i$

3. If a node n contains only:

   - elementary formulas, and
   - does not contain empty.

   Then create a son of n labelled by the next formulas of T_n with outermost nexts removed.

We call rule 2 *the decomposition rule* and rule 3 is called *the step rule.*

So, we start with the formula under test and we add $\Diamond$**empty** to the initial node. This is needed in order to enforce the requirement that a successful path through the tableau terminates. This termination arises through the final node in the path containing **empty**. Another way of looking at this is that it ensures that a marker for termination of a path is included in every state (note $\Diamond$**empty** is either satisfied immediately and a path is terminated or it is preserved in the next state, i.e. $\Diamond$**empty** arises in all descendent states, lemma 5 encapsulates this fact).

Graph Reduction. The next stage in the tableau algorithm is to eliminate unsatisfiable nodes by repeatedly applying the following rules:

- (E1) Eliminate nodes containing formula P and $\neg P$.
- (E2) If a node has no successors and does not contain **empty**, eliminate it.
- (E3) If a node contains **empty** and $\bigcirc P$, eliminate it.

- (E4) If a pre-state contains an unaccepted formula of the form P_1 **;** P_2, $\neg(P_1$ **;** $P_2)$, P_1 **proj** P_2 or $\neg(P_1$ **proj** $P_2)$ then eliminate the pre-state.

This procedure terminates when all unsatisfiable nodes have been eliminated. The first three of these rules embody straightforward cases of an unsatisfiable node. The fourth rule hinges on the notion of a chop or projection formula failing to be accepted. The following rule defines what it means for a formula to be accepted.

- (F) A formula P is *accepted* in a pre-state if there is a path in the tableau leading from that pre-state to a node containing **empty**.

We claim that the initial formula is satisfiable if we can generate a tableau according to the graph construction rules, reduce it according to rules (E1) - (E4) and the initial formula is not eliminated.

Discussion. The tableau algorithm that we present here is simpler in a number of respects than that defined in [17]. For example, a central concept in Wolper's work is that of a *fulfilling path*. For any temporal formula, other than $\bigcirc$, to be satisfied a path which witnesses the satisfaction of the formula must exist. Our accepting paths play a similar role. However, many of the complexities associated with fulfilling paths do not arise with accepting paths since we are searching for *finite* models/satisfying paths and thus, we can use **empty** as a marker for successful completion of such a satisfying model. In addition, Wolper includes a marking mechanism, whereby formulas are marked when they are being considered for fulfillment. This means that formulas that have been fulfilled are carried through the tableau. In contrast, we simply discard formulas once we unfold them with the decomposition rule. This simpler approach is also justified on the grounds that we are working with finite models.

Examples. We present two example tableau evaluations.

1. The tableau expansion of $\bigcirc\bigcirc p$, where p is an atomic proposition, is shown in figure 3. The tableau uses the normal form for **True ; empty** which after a series of applications of the decomposition rule reduces to:

   $$\textbf{empty} \;\vee\; \bigcirc(\textbf{True ; empty})$$

   In order to simplify the presented tableau, we do not show the intermediate decomposition steps.

   In order to clarify the terminology that we have been using, note that nodes: 1, 4, 7, and 10 are pre-states and nodes: 2, 3, 5, 6, 8, 9, 11 and 12 are states.

   Application of tableau reduction will result in the elimination of nodes: 2 and 5. However, accepting paths for the initial node will remain, so the formula is correctly shown to be satisfiable.

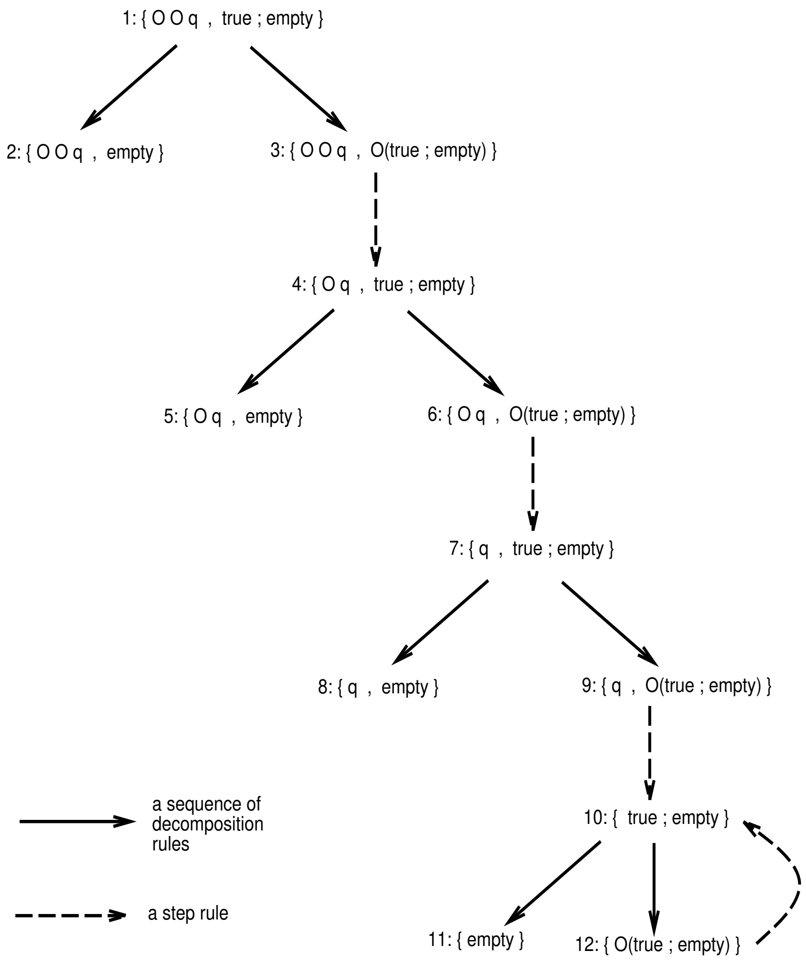

Figure 3: Tableau expansion of $\bigcirc\bigcirc p$

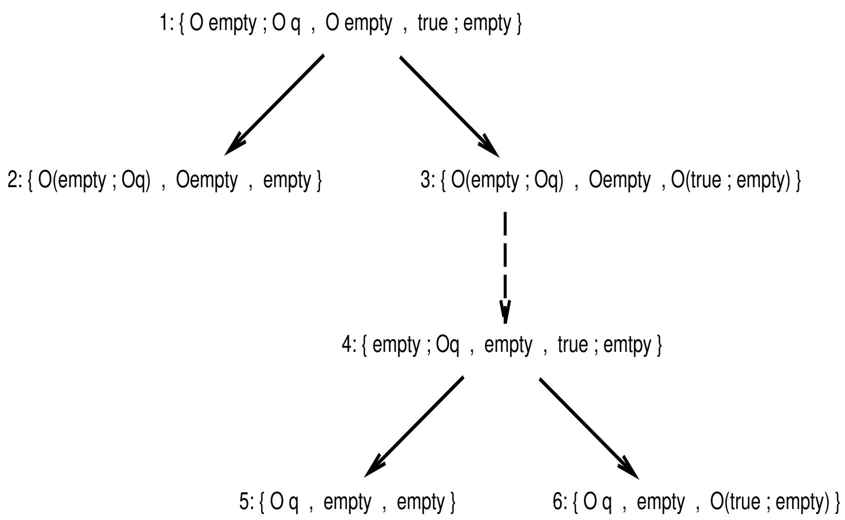

Figure 4: Tableau expansion of $\mathbf{len}(1)$ **;** $\bigcirc q \ \wedge\ \mathbf{len}(1)$

This example indicates the effect of including **True ; empty** in the initial node of the tableau. Specifically, it enables a path to be terminated at any point, by offering the possibility of **empty** at every state. However, any such termination which fails to satisfy the remaining formulae under consideration, will be eliminated during the tableau reduction phase, as happens with nodes 2 and 5. The remaining terminating nodes will yield accepting paths for the formulae under consideration.

2. The tableau expansion for $(\mathbf{len}(1)\ \mathbf{;}\ \bigcirc q)\ \wedge\ \mathbf{len}(1)$ is shown in figure 4. Once again the decomposition arrows shown reflect multiple applications of the decomposition rules. These multiple applications will simplify the normal form expansions, yielding the following relationships:

$$\begin{array}{rcl} \bigcirc\mathbf{empty}\ \mathbf{;}\ \bigcirc q & \equiv & \bigcirc(\mathbf{empty}\ \mathbf{;}\ \bigcirc q) \\ \mathbf{True\ ;\ empty} & \equiv & \mathbf{empty}\ \vee\ \bigcirc(\mathbf{True\ ;\ empty}) \\ \mathbf{empty}\ \mathbf{;}\ \bigcirc q & \equiv & \bigcirc q \end{array}$$

It turns out that all the “terminal” nodes here, i.e. those containing **empty**, will be eliminated because they contain unsatisfied next state formulae. The elimination of all leaf nodes will cause all new leaf nodes (which neither contain **empty** or have any successors) to be eliminated which in turn will have a knock on effect on the new set of leaf nodes and so on, until the initial node has been eliminated. Thus, showing that the formula,

$$(\mathbf{len}(1)\ ;\ \bigcirc q)\ \wedge\ \mathbf{len}(1)$$

is unsatisfiable.

5 Correctness of the Tableau

This section investigates correctness of the tableau algorithm. First we need a small lemma which clarifies how the $\Diamond$**empty** behaves in the initial formula.

Lemma 5 *All states in the tableau contain either* **empty** *or* $\bigcirc$(**True ; empty**).

Proof

In this proof we will refer to the normal form for **True ; empty** which can be determined by substitution from the derivation example 3.4. The normal form is:

$$\begin{array}{rcl}\mathbf{True\ ;\ empty} & \equiv & (\mathbf{empty}\ \wedge\ \mathbf{True})\ \vee \\ & & (\mathbf{True}\ \wedge\ \bigcirc(\mathbf{True\ ;\ empty}))\end{array}$$

Now the proof proceeds by induction over the graph.

Base Case. By rule 1 of graph construction **True ; empty** will be in the initial node. In addition, all initial states (i.e. states reached from the initial node without applying the step rule) will be reached by applying just tableau rules. By inspection of the normal form for **True ; empty** we can see that such an initial state must either contain **empty** (and **True**) or $\bigcirc$(**True ; empty**) (and **True**). Both cases are as required.

Inductive Step. For an arbitrary node n assume that either **empty** or $\bigcirc$(**True ; empty**) is present. We need to show that either **empty** or $\bigcirc$(**True ; empty**) is present in all states accessible from node n.

- *Case (i) -* **empty** *is present.* Since we are at a state we cannot apply the decomposition rule, but in addition, we cannot apply the step rule because **empty** is present. Thus, the node will have no successors and we have what we require vacuously.

- *Case (ii) -* $\bigcirc$(**True ; empty**) *is present.* Since we are at a state we will only be able to apply the step rule. However, all resulting pre-states will contain **True ; empty**. Now in order to reach a new state from such a pre-state, decomposition rules will be applied. In particular, these rule applications will have the effect of reducing **True ; empty** according to its normal form. Now, by inspection of the normal forms, we can see that resulting states must either contain **empty** (and **True**) or $\bigcirc$(**True ; empty**) (and **True**). As was the case with the base condition, both of these are as required.

$\square$

The following theorem shows that our tableau algorithm does indeed check satisfiability.

Theorem 2 *P is satisfiable if and only if the initial node of the graph generated by the tableau method just presented is not eliminated.*

Proof
($\Longrightarrow$) For this we prove the contrapositive:

The initial node is eliminated implies that P is unsatisfiable.

First we prove the general result that a node, n say, labelled $\{P_1, .., P_m\}$ is eliminated implies $P_1 \wedge P_2 \wedge ... \wedge P_m$ is unsatisfiable. We prove this by induction on the structure of the graph. Thus, we assume the result for all successors of node n. Cases (E1) and (E3) are the base cases.

We consider in turn the possible reasons for node n being eliminated.

- (E1) Clearly, if a proposition and its negation are both in T_n then the conjunction of the propositions in T_n is unsatisfiable.
- (E2.1) Assume n is eliminated by (E2) but is not a state. Since n is not a state (and hence decomposition rules can be applied) we know that n must have had successors in the tableau construction. Thus, the sons of n were created using one of the tableau rules, $P \longrightarrow \{S_i\}$ say. From lemma 4 we know that $\Phi \cup \{P\}$ (where Φ is an arbitrary set of formulae) is satisfiable if and only if there exists a j such that $\Phi \cup \{\bigwedge S_j\}$ is satisfiable. However, the fact that no child node containing $\Phi \cup \{S_i\}$ exists implies that (by our inductive hypothesis) for all i, $\Phi \cup \{\bigwedge S_i\}$ is unsatisfiable and hence T_n is unsatisfiable.
- (E2.2) Assume n is eliminated by (E2) and n is a state. Now by lemma 5 we know that n either contains $\bigcirc$(**True ; empty**) or **empty**. However, in this case it must be the former. This implies that node n would have had successor nodes resulting from application of the step rule, which have since been eliminated. This implies that the conjunction of all the P's such that $\bigcirc P$ is in T_n is unsatisfiable (by the inductive hypothesis). By the semantics of $\bigcirc$, it follows that n is unsatisfiable.
- (E3) By the semantics of $\bigcirc$ and **empty** it is clear that for all P, **empty** $\wedge$ $\bigcirc P$ is unsatisfiable.
- (E4) Assume m is a pre-state and is eliminated by (E4). We need to prove the following:

A formula in T_n, P_i say, is not accepted implies that $P_1 \wedge P_2 \wedge \ldots \wedge P_m$ is unsatisfiable.

We prove this in the contrapostive, i.e. we verify:

$$\exists\sigma \,.\, \sigma \models P_1 \wedge \ldots \wedge P_m \implies \forall i(1 \leq i \leq m) \,.\, P_i \textit{ has an accepting path.}$$

Now from our above assumption and properties of conjunction we know that $\forall i(1 \leq i \leq m) \,.\, \sigma \models P_i$. So, take an arbitrary $P \in \{ P_1, P_2, ..., P_m \}$; we need to show that P has an accepting path. We verify this by considering P at the normal form level, i.e

$$P \equiv (P_e \wedge \textbf{empty}) \vee \bigvee_i (P_i \wedge \bigcirc P_i')$$

and by performing induction on the length of σ.

Base Case. Assume $|\sigma| = 0$. Now from lemma 1 we know that $\sigma \models P_e$. In addition, on applying the decomposition rule to n we will be able to reach a state containing P_e and **empty**. The route to this state serves as an accepting path for P, as required.

Inductive Step. Assume $|\sigma| > 0$. Now by lemma 1 we know that $\sigma \models P_i$ and $(\sigma)^1 \models P_i'$ for some i. But, we know by our inductive hypothesis that an accepting path for P_i' exists. Furthermore, our graph construction rules ensure that there is a path from node n to a node containing P_i and P_i' (note: the step rule will be applied once in this path). So, we can concatenate this path with the accepting path for P_i' to get an accepting path for P, as required.

This completes our proof that a node n is eliminated implies that the conjunction of its constituent formulas is unsatisfiable. The required property concerning the initial node is a special case of this result.

($\Longleftarrow$) We need to verify the property:

The initial node of the graph is not eliminated implies P is satisfiable

First note that if the initial node of the graph is not eliminated then an accepting path for P must exist in the (reduced) tableau. Our strategy will be to derive a model for P from this accepting path.

Assume P has an accepting path. Now from that path extract the *subaccepting* path which contains just the states in the accepting path, i.e. all pre-states and intermediate states are removed [2]. We will argue by induction on the size

[2]Note that an accepting path must contain at least one state since the terminating node of the path will be a state. This is because if the terminating node contains a non-elementary formula then the node would be eliminated by rule (E4) since an accepting path for the formula would not exist.

of this subaccepting path that an interval σ exists. In fact, the induction states that σ corresponds exactly to this subaccepting path.

Base Case. Assume the subaccepting path has length 0, i.e. there is just one state (which contains **empty**). However, if this is the case the state cannot contain any temporal formulas since if it did (E3) would have eliminated the node. So, it must have been reached by applying non-temporal decomposition rules. These rules implement a standard propositional logic tableau and will only fail to generate a contradiction if P is satisfiable. The result follows.

Inductive Step. Assume P has a subaccepting path of length $n + 1$. Now if we consider P in normal form:

$$P \equiv (P_e \wedge \mathbf{empty}) \vee \bigvee_i (P_i \wedge \bigcirc P_i')$$

it is clear that the first node in the subaccepting path must "fulfil" P_i and contain $\bigcirc P_i'$ for some i. Furthermore, the tail of the subaccepting path will "fulfil" P_i' and will be of length n. But, now we have a subaccepting path of length n and we can apply our inductive hypothesis to give us an interval, σ say (where $|\sigma| = n$), and $\sigma \models P_i'$. In addition, by *cons*ing the first node of the original subaccepting path on the front of σ we obtain an interval of length $n+1$ that satisfies P.

The result follows. □

References

[1] H. Barringer, M. Fisher, D. Gabbay, G. Gough, and R. Owens. METATEM : A framework for programming in temporal logic. In *Lecture Notes in Artificial Intelligence, vol. 430*. Springer–Verlag, 1989.

[2] M. Ben-Ari, Z. Manna, and A. Pnueli. The temporal logic of branching time. In *8th ACM Symposium on Principles of Programming Languages*, pages 164–176. ACM, 1981.

[3] Howard Bowman, Helen Cameron, Peter King, and Simon Thompson. Specification and prototyping of structured multimedia documents using interval temporal logic. In *ICTL'97, International Conference on Temporal Logic, Manchester*. Kluwer, Applied Logic Series, 1997.

[4] Howard Bowman, Helen Cameron, Peter King, and Simon Thompson. Specification and prototyping of structured multimedia documents using interval temporal logic. Technical Report 3-97 (Kent), Computing Laboratory, University of Kent, 1997.

[5] A. Cau and H. Zedan. Refining interval temporal logic specifications. In *Fourth AMAST Workshop on Real-Time Systems, Concurrent, and Distributed Softward (ARTS'97), LNCS 1231*, May 1997.

[6] Z. H. Duan. *An Extended Interval Temporal Logic and A Framing Technique for Temporal Logic Programming.* PhD thesis, University of Newcastle Upon Tyne, May 1996.

[7] Dov Gabbay. The declarative past and imperative future. In B. Banieqbal, H. Barringer, and A. Pnueli, editors, *Temporal Logic in Specification.* Lecture Notes In Computer Science 389, Springer-Verlag, 1989.

[8] R. Hale. Using temporal logic for prototyping: the design of a lift controller. In *Lecture Notes in Computer Science, vol. 379*, pages 375–408. Springer–Verlag, 1989.

[9] S. Kono. A combination of clausal and non-clausal temporal logic programs. In *Lecture Notes in Artificial Intelligence, vol. 897*, pages 40–57. Springer–Verlag, 1993.

[10] O. Lichtensteim, A. Pnueli, and L. Zuck. The glory of the past. In *Proceedings of Conference on Logics of Programs, LNCS 193.* Springer-Verlag, 1985.

[11] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer-Verlag, 1992.

[12] T. Melham. *Higher Order Logic and Hardware Verification.* Cambridge Tracts in Theoretical Computer Science (31), 1993.

[13] B. Moskowski. *Executing Temporal Logic.* Cambridge University Press, 1986.

[14] N. Rescher and A. Urquart. *Temporal Logic.* Springer-Verlag, 1971.

[15] R. Rosner and A. Pnueli. A choppy logic. In *Proceedings of 1st IEEE Symposium on Logic in Computer Science*, pages 306–314. IEEE, 1986.

[16] Simon Thompson. Constructive interval temporal logic in alf. In *ICTL'97, International Conference on Temporal Logic, Manchester.* Kluwer, Applied Logic Series, 1997.

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