Abstract

Reasoning about events occurring in time is a strong, viable research area in Artificial Intelligence. In spite of numerous successful temporal reasoning theories and systems, few expert system shells exist with temporal reasoning capabilities. Those that do are real-time expert system shells; however, the ontological commitments of these systems limit them to reasoning about past, continuous events. Temporal Production System (TPS) is a production system augmented with an interval-based temporal knowledge representation and maintenance routines that provides a unique tool for temporal domain modeling.

TPS: Incorporating Temporal Reasoning into a Production System

Marcus A. Maloof

March 8, 1995

Acknowledgements

I first want to thank my major professor, Krys Kochut, for his guidance and friendship. Krys' intellect, knowledge, and technical competence continually amaze me and are rivaled only by his patience.

I want to thank my committee members, Donald Nute and Walter D. Potter, for their valuable discussions throughout this project. Their respective backgrounds in logic and expert systems gave them unique perspectives of this work that truly strengthened the final product.

Mr. L. W. Hill of Hill Systems, Inc. also deserves many thanks for providing me the financial assistance necessary to complete this program of study. I will always remember his generosity and vision.

Others requiring thanks include Michael A. Covington for his work on the UGA style sheet, and Leslie D. Roberts for her handy expertise on matters of grammar, mechanics, and readability. Finally, without the support of my family, none of this would have occurred.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Reasoning about events occurring in time and its related formalisms have pervaded several areas of computer science and Artificial Intelligence (AI). The amount of interest in capturing notions of time, formally and programmatically, has blossomed over the past two decades.

The need for temporal reasoning in medical diagnosis, for example, became evident with INTERNIST-1 (Pople 1977), an internal medicine diagnosis system. Miller et al. (1984) cited INTERNIST's lack of temporal reasoning as one of its main design flaws. INTERNIST's inability to reason temporally hindered severity diagnoses which are partially a function of time.

On the other hand, Ventilator Manager (Fagan et al. 1984a), an intensive care monitoring system, used temporal reasoning to augment traditional causal inferences (Fagan et al. 1984b). Instead of having to use rules like:

```
if the patient has been hyperventilating, then ...
```

added temporal reasoning allowed Fagan to construct rules that more precisely represented domain situations. For example, he formulated rules such as:

```
if the patient has been hyperventilating for thirty minutes
then ...
```

While INTERNIST-1 demonstrated the need for temporal reasoning in at least the medical domain, AI researchers largely ignored the temporal characteristics of most domains (McDermott 1982). Prior to 1981, most of the work in temporal reasoning was sparse and varied in formality.

Much of this lack of continuity changed when Drew McDermott (1982) and James Allen (1984) published their respective temporal logics. Since then, several formal temporal reasoning systems have taken one of the two logics as their basis. Most of these, however, are what I call *research vehicles.* These are systems intended for use in controlled, synthesized, somewhat ideal environments and not for use in an application setting, such as a manufacturing plant. Naturally, we as researchers can develop and augment the underlying formalism for applied scenarios, but as these systems stand now, this is not the case.

Not surprisingly, a few researchers have incorporated existing temporal reasoning formalisms into specific-purpose systems and frame-based shells. The ones I would classify as "expert system shells" appear limited since they capture and reason about past, continuous events. These are *real-time expert systems.* Further, these systems require that events associate with a time of occurrence, and consequently, we can think of any temporal analysis as simply trend analysis. These conventions limit these expert systems to domains in which we *can* associate an event with a time.

At the risk of misinterpreting the capabilities of these systems, they have a minimal concept of temporal reasoning. While I do not dispute that these programs reason temporally, they concentrate on one of the safer areas of temporal reasoning. It appears that these systems are essentially intelligent trend analysis programs that do not provide the ability to say anything about the *relationship* between events in time.

Contrary to most of the existing expert system shells with temporal reasoning, I chose to incorporate temporal reasoning into a production system. During the literature survey, I found no production-based systems that were of the scope of *Temporal Production System* (TPS). Shamsudin and Dillon (1991) describe *NetManager*, a production system with added temporal reasoning, but, again, this system concentrates on real-time event analysis.

Kabakçloğlu (1992) also defines a temporal production system. His, however, is a production system whose rule conditions are *relative time logic* formulae. These are "atomic formulae and the temporal order in which they are true relative to a start time ..." (p. 697). While his results are preliminary, there appears to be no way to relate two temporal events, other than sequentially. And many events, depending on the domain, do not have identifiable start times.

TPS differs from these systems because it provides designers with a *temporal blackboard*. We can place events on the blackboard, relate them, and anchor them to a time line. Production rules analyze and update temporal events on the blackboard and behave according to those events and their relationships to other events.

The preceding introduction tacitly outlines the organization of this thesis. In Chapter 2, we survey the formal foundations of temporal reasoning and the areas of computer science and AI that use temporal logics and reasoning. Chapter 3 presents some of the most influential, foundational formalisms and systems developed for temporal reasoning that have appeared in AI literature over the past twenty-five years. Chapter 4 surveys those temporal reasoning systems most closely related to TPS. In Chapter 5, we discuss production systems and OPS5, which is our host system for TPS, which I present in Chapter 6.

Chapter 2

Temporal Logics and Reasoning

This chapter surveys applications of temporal logics in computer science and Artificial Intelligence. We begin with some foundational material, to which we will refer frequently. The remaining sections of this chapter deal with some of the areas where these logics have been applied, such as logic programming and deductive databases. Galton (1987) and Bolour et al. (1982) also present excellent summaries.

2.1 Formal Preliminaries

Almost all sophisticated temporal reasoning systems developed over the last decade have been based on either predicate logic or temporal logic. Since temporal logics appear less frequently than predicate logics, I present a temporal logic traditionally used as a logic for program semantics, and conclude with some features of a temporal ontology.

2.1.1 A Simple Temporal Logic

The following temporal logic is a point-based logic, as opposed to an interval-based logic, because the time primitive is a point. Other researchers (Reichgelt 1987; Shoham and Goyal 1988) refer to this type of logic as a *modal temporal logic* because of its relation to *modal logic* (see Chellas 1980). Others (Resher and Urquhart 1971) refer to these logics as *tense logics* because of the notion of past and future tense. While we only consider a propositional temporal logic (PTL), several others present first-order temporal logics (see Reichgelt 1987; Shoham and Goyal 1988).

Syntax

To formulate PTL syntax, we begin with an alphabet $\mathcal{A}$ such that $p_i \in \mathcal{A}$, for $i \geq 1$; logical connectives $\wedge$ and $\neg$; temporal operators $\bigcirc$ (next time), $\Diamond$ (sometime), $\Box$ (always), and $\mathcal{U}$ (until); and parentheses (and). For formulae, if $p_i \in \mathcal{A}$, then p_i is a formula. Finally, if p_i and p_j are formulae, then so are $\neg p_i$, $p_i \wedge p_j$, $\bigcirc p_i$, $\Diamond p_i$, $\Box p_i$, $p_i \, \mathcal{U} \, p_j$, and (p_i).

Semantics

A *point structure* S is a 4-tuple defined as:

$$S = \langle T, \preceq, \sigma, \pi \rangle$$

where

T is an enumerable set of time points, or times,
$\preceq$ is a total ordering on T that is isomorphic to the integers,
σ is a successor function defined as $\sigma : T \rightarrow T$ such that $t_{i+1} = \sigma(t_i)$, and
π is a function defined as $\pi : (\mathcal{A} \times T) \rightarrow \{\top, \bot\}$ that assigns truth

values to propositions at a given time; $\top$ denotes **true** and
$\perp$ denotes **false**.

Given a point structure S and a time t, such that $t \in T$, we define satisfiability (denoted $\models$) in PTL as follows:

$$
\begin{array}{l}
\langle S,t\rangle \models p_i \text{ if and only if (iff) } \pi(p_i,t) = \top \\
\langle S,t\rangle \models \neg p_i \text{ iff not } \langle S,t\rangle \models p_i \\
\langle S,t\rangle \models p_i \wedge p_j \text{ iff } \langle S,t\rangle \models p_i \text{ and } \langle S,t\rangle \models p_j \\
\langle S,t\rangle \models \bigcirc p_i \text{ iff } \langle S,\sigma(t)\rangle \models p_i \\
\langle S,t\rangle \models \Box p_i \text{ iff } (\forall t')(t \preceq t' \text{ and } \langle S,t'\rangle \models p_i) \\
\langle S,t\rangle \models \Diamond p_i \text{ iff } \langle S,t\rangle \models \neg\Box\neg p_i \\
\langle S,t\rangle \models p_i \, \mathcal{U} \, p_j \text{ iff } \langle S,t\rangle \models \Box p_i \text{ or } \langle S,t\rangle \models p_j \\
\qquad\qquad\qquad \text{or } (\exists t'')(t \preceq t'' \text{ and } \langle S,\sigma(t'')\rangle \models p_j \\
\qquad\qquad\qquad \text{and } (\forall t')(t \preceq t' \preceq t'' \rightarrow \langle S,t'\rangle \models p_i))
\end{array}
$$

A formula ϕ is *satisfiable* if and only if there exists $\langle S,t\rangle$ such that $\langle S,t\rangle \models \phi$. S is a *model* for ϕ if and only if for all t, $\langle S,t\rangle \models \phi$. Finally, ϕ is *valid* if and only if for all S, S is a model for ϕ.

Axiomatization

Following Wolper (1983), the axioms and inference rules for PTL are as follows:

Axiom Schemata:

$$\vdash \Diamond\phi \equiv \neg\Box\neg\phi \tag{A1}$$
$$\vdash \Box(\phi \supset \psi) \supset (\Box\phi \supset \Box\psi) \tag{A2}$$
$$\vdash \bigcirc\neg\phi \equiv \neg\bigcirc\phi \tag{A3}$$
$$\vdash \bigcirc(\phi \supset \psi) \supset (\bigcirc\phi \supset \bigcirc\psi) \tag{A4}$$
$$\vdash \Box\phi \supset \phi \wedge \bigcirc\phi \wedge \bigcirc\Box\phi \tag{A5}$$
$$\vdash \Box(\phi \supset \bigcirc\phi) \supset (\phi \supset \Box\phi) \tag{A6}$$
$$\vdash \Box\phi \supset \phi \, \mathcal{U} \, \psi \tag{A7}$$
$$\vdash \phi \, \mathcal{U} \, \psi \equiv \psi \vee (\phi \wedge \bigcirc(\phi \, \mathcal{U} \, \psi)) \tag{A8}$$

Inference Rules:

If w is a propositional tautology, then $\vdash w$ (I1)
If $\vdash w_1 \supset w_2$ and $\vdash w_1$, then $\vdash w_2$ (I2)
If $\vdash w$, then $\vdash \Box w$ (I3)

Decision Procedure

A decision procedure exists for PTL based on the *semantic tableau method.* This procedure is an adaptation or a reduction of Wolper's (1983) decision method for Extended Temporal Logic.

The decision method involves starting with a temporal formula ϕ and repeatedly applying a set of reduction rules until we decompose the formula into sets of *elementary* formulae. After eliminating unsatisfiable sets of elementary formulae, we are able to determine if ϕ is satisfiable. If this is the case, then we can search for a model.

A formula is *elementary* if and only if it is of the form ϕ or $\bigcirc\phi$, where ϕ is either p_i or $\neg p_i$. If a formula is not elementary, it is *non-elementary.* During the decision procedure, we need to distinguish between formulae as processed or not, thus a formula ϕ can be *marked* (i.e., processed), denoted ϕ^*, or *unmarked* (i.e., unprocessed). The decomposition procedure forms a directed graph. A *state* is a node in the graph containing only elementary or marked formulae. A *pre-state* is the initial node and any child node of a state. Finally, the set of formulae for node n, denoted R_n, is in disjunctive normal form. Hence, the formula

$$(\phi_{11} \wedge \phi_{12}) \vee (\phi_{21} \wedge \phi_{22})$$

translates to the node representation

$$R_n = \{\{\phi_{11}, \phi_{12}\}\{\phi_{21}, \phi_{22}\}\}.$$

Following Wolper (1983), the reduction rules are as follows:

$\phi \wedge \psi$	$\Rightarrow$	$\{\{\phi, \psi\}\}$
$\neg(\phi \vee \psi)$	$\Rightarrow$	$\{\{\neg\phi, \neg\psi\}\}$
$\neg(\phi \supset \psi)$	$\Rightarrow$	$\{\{\phi, \neg\psi\}\}$
$\neg\neg\phi$	$\Rightarrow$	$\{\{\phi\}\}$
$\neg\bigcirc\phi$	$\Rightarrow$	$\{\{\bigcirc\neg\phi\}\}$
$\Box\phi$	$\Rightarrow$	$\{\{\phi, \bigcirc\Box\phi\}\}$
$\neg(\phi\, \mathcal{U}\, \psi)$	$\Rightarrow$	$\{\{\neg\psi, \neg\phi\}\{\neg\psi, \bigcirc\neg(\phi\, \mathcal{U}\, \psi)\}\}$
$\neg\Diamond\phi$	$\Rightarrow$	$\{\{\neg\phi\}\{\bigcirc\neg\Diamond\phi\}\}$
$\phi \vee \psi$	$\Rightarrow$	$\{\{\phi\}\{\psi\}\}$
$\neg(\phi \wedge \psi)$	$\Rightarrow$	$\{\{\neg\phi\}\{\neg\psi\}\}$
$\phi \supset \psi$	$\Rightarrow$	$\{\{\neg\phi\}\{\psi\}\}$
$\Diamond\phi$	$\Rightarrow$	$\{\{\phi\}\{\bigcirc\Diamond\phi\}\}$
$(\phi\, \mathcal{U}\, \psi)$	$\Rightarrow$	$\{\{\psi\}\{\phi, \bigcirc(\phi\, \mathcal{U}\, \psi)\}\}$
$\neg\Box\phi$	$\Rightarrow$	$\{\{\neg\phi\}\{\bigcirc\neg\Box\phi\}\}$

To prove a formula ϕ is satisfiable, we construct a directed graph with the following procedure:

- Rule C1: For some PTL formula ϕ, we name ϕ the *initial formula* and label it as the *initial node*, denoted $\{\phi\}$. Next, we repeatedly apply construction rules C2 and C3. Note that we create a child node only if an identical node is not present in the graph; otherwise, we construct a link to the identical node.

- Rule C2: If a node n contains the formula set R_n, and some formula $\phi \in R_n$ is unmarked and non-elementary, and there exists a reduction rule for ϕ such that $\phi \Rightarrow \{S_i\}, i \geq 1$, then for each S_i, create a child node $n + i$ of node n, such that the child node contains the formula set:

  $$(R_n - \{\phi\}) \cup \{S_i\} \cup \{\phi^*\},$$

  where ϕ^* is ϕ marked.

- Rule C3: If a node n contains the formula set R_n, consisting of only elementary and marked formulae, then create a child of node n consisting of all ϕ such that $\bigcirc\phi \in R_n$.

This procedure expands the initial formula over time until it reaches the formula's atomic terms. Given the rules of decomposition, the nodes in the graph, which are decompositions of the initial formula, contain either subformulae, negated subformulae, or such formulae preceded by $\bigcirc$ operator. Also, for any given node in the graph, the non-elementary, unmarked formulae contained in that node are equivalent to the disjunction of the formulae labeling its children.

To determine satisfiability, we begin eliminating contradictory nodes. If this elimination procedure eliminates the initial node $\{\phi\}$, then ϕ is unsatisfiable. We eliminate graph nodes by repeatedly applying the following rules:

- Rule E1: If a node contains p and $\neg p$, then eliminate the node.

- Rule E2: If all children of a node have been eliminated, then eliminate the node.

- Rule E3: If a pre-state node contains an *eventuality* formula of the form $\neg\Box\phi$ or $\neg(\phi\, \mathcal{U}\, \psi)$ that is not *fulfillable*, then eliminate the node.

A pre-state node n is *fulfilled* if and only if the following criteria hold:

1. the left child of the node n is a state; that is, the formulae in the left child node are elementary or marked. Then, the formula is *immediately fulfilled*.

2. the formulae in some child of node n are fulfilled; that is, the formulae in node n are eventually immediately fulfilled.

We can recursively compute the fulfillment of all eventualities in the graph as follows:

1. Mark all nodes that are immediately fulfilled.
2. Recursively mark all nodes that can be immediately fulfilled, given the current set of marked nodes, until no additional nodes can be marked.

The node elimination procedure halts when it eliminates all unsatisfiable nodes. If the initial node remains after the procedure terminates, the formula ϕ is satisfiable and we can efficiently construct a model; otherwise formula ϕ is unsatisfiable. Theorem 2.1 states that this decision procedure is sound and complete for PTL. The proof that follows is an adaptation of the one given by Wolper (1983).

Theorem 2.1 *A PTL formula ϕ is satisfiable if and only if the initial node of the graph generated by the tableau decision procedure for formula ϕ is not eliminated.*

Proof.
($\Rightarrow$) If a PTL formula ϕ is satisfiable, then the initial node is not eliminated. We prove by induction on the depth of the graph generated by the semantic tableau decision procedure that if this procedure eliminates a node in the graph labeled by $\{\phi_1, \ldots, \phi_k\}$, then $\{\phi_1, \ldots, \phi_k\}$ is unsatisfiable.

Case 1. (Base Case) Rule E1 eliminated the node, thus it contained p and $\neg p$ and is unsatisfiable.

Case 2. Rule E2 eliminated a non-state node n. A reduction rule $\phi \Rightarrow \{S_i\}$ created the children of node n. We can verify that for each reduction rule, ϕ is satisfiable if and only if some S_i is satisfiable. By the induction hypothesis, if the decision procedure eliminates the children of node n, meaning all children are unsatisfiable, then node n is also unsatisfiable.

Case 3. Rule E2 eliminated a state node n. Since a state contains only elementary and marked formulae, the only condition under which children exist is one in which node n contains $\bigcirc$-formulae. By the inductive hypothesis, for the formulae $\bigcirc\phi_i$, if ϕ_i is unsatisfiable, then $\bigcirc\phi_i$ is also unsatisfiable. Hence, so are the formulae in node n.

Case 4. Rule E3 eliminated node n. For the eventualities in node n, there is no path that leads to a child node that fulfills these eventualities, implying that node n's eventualities are unsatisfiable. Therefore, by the inductive hypothesis, the formulae in node n are unsatisfiable.

($\Leftarrow$) If the initial node is not eliminated, then the PTL formula ϕ is satisfiable. To prove this, we must show that if the initial node is not eliminated, there is a model for ϕ. Excluding eventualities, a traversal through the graph generated by the decision procedure defines a model for the initial formula. We must show, however, that the eventualities form a finite state model for the formula ϕ.

Only pre-states in the graph contain unfulfilled eventualities, thus for every pre-state in the graph, *unwind* a path from that pre-state to a state node such that all of these eventualities are fulfilled on the path. We unwind a path by constructing a loopless chain that begins with the pre-state node, ends with the state node, and consists of the nodes along the path between the pre-state node and the state node. Because every reduction rule for eventualities splits nodes into two children, there are at most 2^ℓ eventualities in the graph, where ℓ is the number of disjuncts in the initial formula; therefore, the length of the longest of these paths is also 2^ℓ. Once all paths have been determined, we link them together to construct a chain not longer than $2^\ell \times 2^\ell$, or $2^{2\ell}$, nodes. Since this chain is finite, it forms a model for the eventualities in the graph. Therefore, a model exists if the initial node is not eliminated.

We have proven by induction on the depth of the graph generated by the semantic tableau decision procedure that for all PTL formulae ϕ, ϕ is satisfiable if and only if the decision procedure does not eliminate the initial node containing ϕ. $\square$

Completeness

We now prove completeness for PTL. This completeness proof follows directly from the inductive proof of Theorem 2.1 and is an adaptation of Wolper's (1983) completeness proof.

Theorem 2.2 *PTL, the axiomatic system consisting of axioms (A1)–(A8) and inference rules (I1)–(I3), is complete.*

Proof. For a formula ϕ, if the decision procedure eliminates an initial node $\{\{\neg\phi\}\}$ then ϕ is provable. We must show that for a node labeled by the formula set $R_n = \{\neg\phi_1, \ldots, \neg\phi_k\}$, that $\vdash \phi_1 \vee \ldots \vee \phi_k$. We prove this by induction on the depth of the graph.

There are four cases to prove:

Case 1 (Base Case). Rule E1 eliminated the node.

For node n, labeled with the formula set

$$R_n = \{p, \neg p, \neg\phi_1, \ldots, \neg\phi_k\}$$

we have

$$\vdash \neg p \vee p \vee \phi_1 \vee \ldots \vee \phi_k$$

by inference rule (I1) ($\neg p \vee p \vee \cdots$ is a tautology).

Case 2. Rule E2 eliminated a non-state node.

For node n, labeled by

$$R_n = \{\neg\phi_1, \ldots, \neg\phi_k\}$$

the reduction rule

$$\neg\phi \Rightarrow \{\{\neg\phi_{ij}\}\}$$

for some $\neg\phi \in R_n$, created the children of node n. We can prove within the axiomatic system that, in general, for each of these rules

$$\vdash \neg\phi \equiv \bigvee_i (\bigwedge_j \neg\phi_{ij})$$

which we can write as

$$\vdash \bigwedge_i (\bigvee_j \phi_{ij}) \supset \phi$$

by DeMorgan's Laws and (I1). As we eliminate the children of node n, by the inductive hypothesis,

$$\vdash \bigvee_j \phi_{ij} \tag{2.1}$$

for all i children. Then, by a derived transformation rule,

$$\vdash \bigwedge_i (\bigvee_j \phi_{ij}). \tag{2.2}$$

Thus, by equations 2.1, 2.2, and (I2)

$$\vdash \phi$$

for some $\phi \in R_n$. Then, for all $\phi \in R_n$, by inference rule (I1),

$$\vdash \phi_1 \vee \cdots \vee \phi_k$$

Case 3. E2 eliminated a state node.

The only state nodes that have successors are those states with $\bigcirc$-formulae $\phi'_1, \ldots, \phi'_m$ in $R_n = \{\neg\phi_1, \ldots, \neg\phi_k\}$. We thus have

$$\vdash \phi'_1 \vee \cdots \vee \phi'_m$$

Then,

$$\vdash \neg\phi'_1 \supset \cdots \supset \phi'_m$$

by (I1) and (I2), and

$$\vdash \Box[\neg\phi'_1 \supset \cdots \supset \phi'_m]$$

by (I3). Then by (A5), (I2), and simplification to the second conjunct, we have

$$\vdash \bigcirc[\neg\phi'_1 \supset \cdots \supset \phi'_m]$$

Finally, we use (A4), (A3), and (I1) to write

$$\vdash \bigcirc\phi'_1 \vee \cdots \vee \bigcirc\phi'_m$$

By the inductive hypotheses and (I1), we conclude that

$$\vdash \phi_1 \vee \cdots \vee \phi_k$$

Case 4.1. E3 eliminated the node and

$$R_n = \{\neg\Box\phi_1, \neg\phi_2, \ldots, \neg\phi_k\}$$

labels node n.

For $\neg\Box\phi_1$, we have the reduction rule,

$$\neg\Box\phi \Rightarrow \{\{\neg\phi\}\{\bigcirc\neg\Box\phi\}\}$$

whose left child is eventually fulfilled and therefore inductively consists of elementary formulae. Thus, by the inductive hypothesis, $\vdash \phi$. Then, trivially by inference rule (I3), $\vdash \Box\phi$. Hence, by (I1) and (I2)

$$\vdash \Box\phi_1 \vee \cdots \vee \phi_k.$$

Case 4.2. E3 eliminated the node and

$$R_n = \{\neg(\phi_1 \mathcal{U} \phi_2), \phi_3, \ldots, \neg\phi_k\}$$

labels node n.

For $\neg(\phi_1 \mathcal{U} \phi_2)$, we have the reduction rule,

$$\neg(\phi\ \mathcal{U}\ \psi) \Rightarrow \{\{\neg\psi, \neg\phi\}\{\neg\psi, \bigcirc\neg(\phi\ \mathcal{U}\ \psi)\}\}$$

whose left child is eventually fulfilled and therefore inductively consists of elementary formulae. Thus, by the inductive hypothesis, $\vdash \phi_1 \vee \phi_2$. Then, by proof,[1] $\vdash \phi_1\ \mathcal{U}\ \phi_2$. Thus, by (I1) and (I2),

$$\vdash \phi_1\ \mathcal{U}\ \phi_2 \vee \phi_3 \cdots \vee \phi_k.$$

We have shown by induction on the depth of the graph constructed by the semantic tableau decision procedure that for all PTL formulae, if the procedure eliminates the initial node $\{\{\neg\phi\}\}$ then ϕ is provable. Hence, PTL is complete. □

[1] See Appendix A.

Additional Operators

Depending on the application, we might need temporal operators for the past, such as "always in the past," denoted $\overleftarrow{\Box}$. Future tense operators require similar syntactic sugaring (see Orlowska 1982). Alternate operators for $\Diamond$ and $\Box$ are F (will be the case that), and G (will always be the case that), respectively. Their past tense counterparts are P (has been the case that) and H (has always been the case that). See Resher and Urquhart (1971) for further details.

2.1.2 Ontological Notions

An *ontology* is a theory about the nature of existents. In the context of a computational system, an ontology not only includes those things that exist, but it also includes how the researcher views or models time or space. Crucially, this view can later affect what notions the system can capture and therefore directly impacts the system's viability.

In the following sections, we survey the most salient aspects of an ontology. We begin considering valid temporal primitives and their operators. Structures of time are also important, as is the organization of primitives within this structure.

Individuals

The first ontological question to answer is what individuals exist. The traditional primitive is the *point.* Just as a point in Euclidean space has no spatial dimension, our primitive, the point, has no temporal dimension.

Consequently, there is much criticism of points based on the counterintuitive notion of an instantaneous time moment. Although we would be hard put to find an example of a truly instantaneous event in the physical world, there are occasions, specifically in abstract temporal reasoning, when points are handy (see Galton 1990).

On the other hand, we could choose intervals which are contiguous spans of time. These primitives appear ideal for capturing temporal phenomena in the physical world. Even an event as seemingly instantaneous as a lightning flash has duration. Furthermore, most researchers in natural language semantics and high-level reasoning stress that some type of interval representation as a modeling primitive is necessary (Fleck 1989:251).

As an alternative to using only intervals, we can always begin with points and define intervals in terms of left and right end points. Presumably, we get the advantage of both representations. Several researchers in Artificial Intelligence take this approach (Dean 1986; Kowalski and Sergot 1986; McDermott 1982; Shoham 1988b).

Formally, we can choose either primitive. Van Benthem (1990) demonstrates that we can either start with points and construct intervals, or begin with intervals and derive points. Ultimately, when designing a temporal reasoning system, the domain usually determines the appropriate primitive.

Operators

After selecting temporal primitives, we must define operators. If using points as individuals, the three fundamental operators are `before`, `same-time`, and `after`. Naturally, we can use these basic operators to define more intuitive interval operators as well as additional point operators, like `between`, or `sometime`.

If we choose intervals as our primitive, then the fundamental set of operators is much richer than that for points. We can arrange two intervals into thirteen unique configurations, displayed in Figure 2.1.

Time Structures

If time consists of points, then we can organize time in numerous ways. Recall that a point structure S is a 4-tuple such that $S = \langle T, \preceq, \sigma, \pi \rangle$ and that $\preceq$ is a complete order relation defined on the set of time points T; hence, we have *linear time.* Furthermore, we can define *now* as some time $t_n \in T$. The *past* is therefore all times before t_n and the future is all times after t_n.

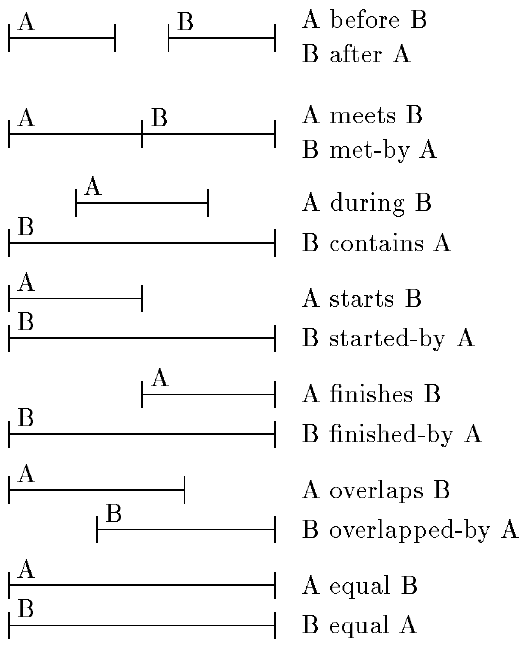


Figure 2.1: Thirteen interval relations.

There is no restriction, however, on the definition of $\preceq$; we can also define it as a partial order on T. Now we can structure time in a tree-like manner called *branching time*. Typically, future time branches while the past is presumably determined.

Branching time gives rise to the notion of *copresence*. Since time branches, there might be one future in which a proposition is true and another future in which the same proposition is false. Further, each of these propositions has a unique time. If the times associated with these propositions are the same temporal distance from a time of reference, say *now*, then the times are *copresent* or the relationship between the two times is *copresence* (Thomason and Gupta 1980).

Formally, no restrictions exist on how we can organize time. *Chronos* (Bruce 1972), discussed in Section 3.4, organizes time into branching future and past. Shoham (1988c) also adopts this structure, but allows *re-entrance*; that is, a time path branches at a time and then meets again at some future time. We might even need *circular time* to model a finite-state process, like a traffic light or an elevator system.

Density and Discreteness

The final ontological notion considered involves properties of time points in a time structure. The first property, called *discreteness*, organizes time in such a way that for every time, there is a next time. In other words, discrete time is isomorphic to the integers.

An alternative to discrete time is *dense time*, which maps to the rational numbers. Formally,

$$(\forall t, t')(\exists t'')(t \preceq t'' \preceq t' \wedge t \neq t' \neq t'')$$

Notice that we cannot define the "next time" operator for dense time.

2.2 Logics of Programs

Researchers in computer science use temporal logics for hardware and software specification and validation. I concentrate mainly on software specification and validation, but these discussions are analogous to hardware.

We can think of programs as temporally ordered sequences of instructions. The ontological conventions adopted for *logics of programs* include discrete time with either linear or branching time structures, depending on if program execution is sequential or concurrent. Operators usually include future tense operators plus the "next time" and "until" operators.

Using temporal logics, researchers can formally specify program properties, such as *eventuality properties* (Manna 1980). These properties are those that if some occurrence arises during execution, then some other occurrence will follow during the same execution. For instance, if the printer begins printing, we expect that it will stop. We can express this notion in temporal logic as follows:

$$p \supset \Diamond \neg p$$

That is, if the printer is printing, then we expect there to be a time in the future when the printer is not printing.

The problem with specifying and validating hardware and software systems with temporal logics, and with any formal means for that matter, is the intractability of completing such a task with large (more than 10,000 lines of code) software projects. Pragmatically it is impossible, and consequently has been relegated to theoretical domains.

2.3 Temporal Logic Theorem Proving

The next area of research we consider involving temporal logics is automated theorem proving. We can use temporal logic theorem provers to prove satisfiability for a temporal logic formula or to verify a software specification expressed in temporal logic. One method of satisfying a temporal logic formula is the semantic tableau method, discussed in Section 2.1.1.

An alternative is *temporal resolution*, which is similar to resolution for predicate calculus (see Genesereth and Nilson 1988). While most resolution methods require formulae to be in *clausal form*, or expressed as conjunctions of disjuncts, Abadi and Manna's *nonclausal* approach is free from this requirement. This enhances legibility of the original formula as well as the resulting proof (Abadi and Manna 1988:2). Proving satisfiability involves applying temporal resolution rules to the original formula until we derive the empty set or reach some stop condition. If this refutation process reaches the empty set, then the original formula is provable or satisfiable.

2.4 Temporal Logic Programming

The next progression is to use temporal logic as a programming language. Just as a subset of predicate logic serves as a basis for Prolog, several researchers have constructed compilers and interpreters for programming languages based on subsets of first-order temporal logic (see Abadi and Manna 1989).

In computer science, we have been discussing software validation problems. Even in theoretical situations, proving the correctness of a program is arduous. Part of this difficulty lies in how we express the program. For example, if we are programming in Pascal and we want to formally verify a program's correctness, we must rely on mathematical notions that represent the Pascal programming constructs.

Alternatively, we have already seen that temporal logics can express programs. If we are using a temporal logic programming language, our program is a temporal logic formula. Thus, program correctness follows directly from the program itself since proving the program's correctness involves proving the formula's satisfiability (see Hale 1987).

2.5 Deductive Databases

The advent of logic programming prompted researchers to construct *deductive databases*, which augment a traditional database with domain axioms. Hence, queries can deduce information by applying the axioms to the information in the database.

When building a deductive database, we inevitably discover a situation in which we have temporally ordered events and our axioms must deduce information about them. Consider an administrative database, for example, that stores information about employees' salaries, promotions, and so forth. Temporal axioms allow deduction of rank or salary during periods of a person's employment. Event Calculus (Kowalski and Sergot 1986), which we discuss in Section 3.7, and the system described by Lee et al. (1985) are systems for temporal deductive databases.

2.6 Natural Language

Temporal reasoning is important in natural language because semantic analyzers must understand and generate a representation for temporal meanings and references in sentences and discourse. Furthermore, as natural language systems process discourse and new, clarifying temporal information becomes known, these systems should automatically and reliably maintain the temporal relationships and knowledge expressed in the discourse.

In Section 4.3, we discuss Ariño's (1989) automatic knowledge acquisition system for a temporal domain. His system stands as a prime example of how temporal reasoning applies to natural language.

Chapter 3

Temporal Reasoning Systems

Temporal reasoning, as a research discipline, divides into four major areas: *prediction*, *explanation*, *planning*, and *learning* (Shoham and Goyal 1988). Prediction involves starting with an initial state of the world and the rules that govern change, and predicting what the future state of the world will be.

Explanation is the opposite of prediction. Rather than finding a future state, we want to find a possible initial state of the world given the current state, or states, of the world and the rules governing change.

Planning is important in several disciplines, such as manufacturing and robotics. In this area, we want to produce a plan for an agent—for instance, a robot. We want to give our planner the initial state of the world, the rules governing change, and a desired target state of the world. The planner should then develop the steps required of our agent to reach the target state.

Learning, the final area, involves determining the rules governing change from world descriptions. This constitutes one of the most challenging disciplines in AI.

Shoham and Goyal (1988) partition temporal reasoning approaches into two classifications: change-based and time-based. Change-based approaches concentrate on those individuals in a domain that change irrespective of time. Situation Calculus (McCarthy and Hayes 1969) is the prototypic change-based formalism.

Time-based approaches, however, grant time a greater role and acknowledge that in some situations, time is the only changing variable. Even though domain objects may remain static with respect to time, we need a formal method to allow reasoning in these situations.

3.1 Representation

Most time-based temporal reasoning systems depend on either predicate or temporal logic as a formal foundation. While temporal logics have syntax and semantics to handle temporal notions, predicate logics do not.

To incorporate time into predicate logic, following Shoham and Goyal (1988), we can *reify* a first-order sentence using either a `true` or a `holds` predicate. For example, in the Blocks World, to express that block `A` was on block `B` over the interval defined by the time points `T1` and `T2`, we would write:

```
holds(T1, T2, on(A, B)).
```

Regardless of how we choose to represent temporal knowledge, there are several considerations necessary for a complete temporal model (Allen 1981). First, the model must make no explicit mention of time. Often events, while occurring *in* time, do not relate *to* time.

Second, the model needs to allow uncertainty. When modeling two events, we might not be able to determine the precise relationship between the events, so our formalism must allow fuzzy temporal notions. This contributes to the framework's expressivity as well as its robustness.

Third, the concept of *persistence* is important. The model must represent events like:

- The car is pink

and understand that the car is pink until told otherwise.

Finally, a temporal model must have a "strong sense" (Allen 1981) of *now*. For a temporal reasoning system, the concept of *now* or *the present* tends to require definition in the context of that system. *Now* appears to be a "computational time of reference" defined in terms of individual temporal reasoning systems rather than a concept that transcends all systems.

Furthermore, the manner in which *now* changes with respect to other events must be efficient and transparent. Transparency here, relates to the modeler's viewpoint; that is, the concept of *now* should not hinder or impede how a modeler captures a domain. On the other hand, if the temporal database requires massive changes every time *now* changes or progresses, this would not be efficient (and from the user's viewpoint, it would no longer be transparent).

3.2 Problems in Temporal Reasoning

Shoham and Goyal (1988) identify three formalism-independent problems. These problems originally occurred within the framework of Situation Calculus, but have since transcended all other approaches to temporal reasoning.

The most famous of these problems is the *frame problem*. This involves a problem solver's ability or inability, as the case may be, to update its knowledge about domain objects that have not changed. For instance, if a person drives her car to Atlanta, the car's location changed, but its color, make, model, and so on, has not.

One method of addressing this problem is to formulate *frame axioms* which specifically state what attributes of a domain object remain static with respect to change in the object. So, if an object having three attributes changes location, we have one axiom for the object's location change and three frame axioms that deduce the obvious—that the object's color, size, and weight have not changed, for example.

The added computation required to process frame axioms significantly hinders the ability of AI researchers to tackle more complex applications. And as I present the other two problems, notice how each requires its own version of frame axioms.

The *ramification problem* involves a problem solver's ability to update knowledge about domain objects as a result of some action. For example, if a person drives her car to Atlanta, the car's engine, wheels, and so on, also go to Atlanta. The similarities between this problem and the frame problem, and the consequent solution and resulting pitfalls should be apparent.

Finally, the *qualification problem* involves a reasoner's ability to implicitly infer that nothing is wrong. When something is wrong, however, we need the reasoner to understand what caused the problem. Let us look at our car example again. Assume our car will not start. The qualification problem arises when the reasoner attempts to determine why the car will not start. This requires vast amounts of information and reasoning. Yet, the reasoner needs this information only when the car will not crank, which is an infrequent occurrence. Ideally, we want the reasoner to determine when to apply the fault knowledge and inference procedures to avoid long, unnecessary chains of reasoning. This idea has prompted many researchers to investigate *nonmonotonic logics*, and we return to these logics in Section 3.10. Two special cases of the qualification problem are the *extended prediction problem* and the *persistence problem*, which we discuss in Section 3.5.

3.3 Situation Calculus

Situation Calculus (McCarthy and Hayes 1969) is a first-order logic for describing situations in the world. In fact, the world is a set of states, or situations, and certain things are true in a given state or situation. Since Situation Calculus is a change-based approach, we progress into another state whenever a change occurs in the current state. There are, however, several problems that arise in Situation Calculus that are common to change-based approaches.

Since all changes occur within a state, these changes and other states must occur before, during, or after another change or state. To represent

- While John was doing his laundry, Mary started doing hers.

Bruce		Traditional Operators
`overlaps`	$\equiv$	(`overlaps` & `meets`)
`during`	$\equiv$	(`during` & `starts` & `finishes`)

Table 3.1: Temporal operator comparison.

we have two options. First, we can put John and Mary's laundering activities in the same state which results in an inability to say anything about the period when only John laundered.

Alternatively, we can use Mary's change from not laundering to move us into a new state in which both John and Mary are washing clothes. But Situation Calculus has no recollection of past events, so the fact that John was laundering without Mary in the previous state is lost. After we make the transition to the new state in which both John and Mary are laundering, we can make no distinction between this scenario and the previous one. In Situation Calculus, the two scenarios would in fact be the same, which is clearly not the case.

Coping with continuous change also presents a problem in Situation Calculus. To express the action of adding heat to water, we must either condense the entire heating process into one situation or use an infinite quantity of situations to represent the continuous heating of the water. The former solution is meaningless and the latter is programmatically impossible.

3.4 Chronos

Chronos (Bruce 1972) is a question and answer program based on a model for temporal references in natural language. Chronos was the first temporal reasoning system developed using a temporal logic.

Bruce's ontology includes discrete time that branches into the past and future. Individuals in Chronos are *events* which are contiguous, linear paths through time. Finally, Chronos allows intuitive interval and point operators like **`before`**, **`during`**, **`same-time`**, **`overlaps`**, **`between`**, and their inverses, as appropriate. Table 3.1 details how Bruce's operators relate to the traditional interval relations, presented in Figure 2.1.

Chronos is interesting because it was the first system to use a temporal logic as a knowledge representation, but temporal reasoning in Chronos consists only of information retrieval and simple relationship comparisons. Furthermore, there are no temporal knowledge maintenance routines (to infer implicit temporal relationships from explicit ones) and no way to express uncertain temporal relations (cf. Allen 1983).

3.5 Histories

Naïve Physics is the area of common sense reasoning concerned with formalizing how human individuals know and reason about phenomena governed by the laws of physics without knowing anything about those governing laws. In a series of papers, Hayes (1979, 1984a, 1984b) lays a formal foundation for naïve physics.

In his first paper[1] (Hayes 1984a), Hayes develops a highly creative and abstract ontology for liquids based on predicate logic. He identifies fifteen possible states of fluids, some of which move in space, but all of which have temporal extent. To capture these states of liquids, Hayes posits *histories*, which are contiguous flows of space-time in which events occur. Histories have temporal extent, but more importantly, have spatial extent and actually mimic the forms of the physical objects they represent. Examples of histories include a car in a driveway and the flow of water from a faucet.

Hayes acknowledges that his theory is preliminary and does not discuss implementing his ideas. While I found no system based on histories, Shoham and McDermott (1988) criticize histories by presenting a prediction task in a histories context.

The *extended prediction problem* is a special case of the quantification problem in which we want a reasoner to make extended predictions about how things will change. Assume we have two billiard balls

[1]Hayes actually wrote this paper in 1978.

rolling toward each other on a table. From this initial state, we want our reasoner to tell us when and where the balls impact and when and where the balls come to rest.

If we use histories, each ball is a history with some temporal extent. As for the nature of this extent, we have three choices:

1. extend the histories indeterminately into the future,
2. extend the histories to the point of impact, or
3. extend short histories and check if they meet.

The first solution is unacceptable because it does not reflect what will happen, assuming the balls collide. The second case is also inadequate since if we could extend the histories to the point of impact, then we could predict where the impact occurs, which is what we are trying to determine.

The third solution is also erroneous since we do not know how long the ball will continue to roll. What if we extend a history and the ball stops rolling before reaching the end of the history? We could extend smaller histories, but this is also subject to the same persistence problem. As the length of the histories converges to zero, the quantity of histories required soars toward infinity. And on this basis, Shoham and McDermott conclude that histories are inadequate for solving the extended prediction problem.

As a result they define yet another problem called the *persistence problem*, which is a special case of the extended prediction problem. This problem involves a reasoner's ability to predict how long a phenomenon will persist. If histories had the ability to determine how long a ball's rolling would persist in a particular direction, then the second and third solutions for the billiard balls prediction task would have worked.

3.6 Time Specialist

Of the systems I consider in this thesis, all except *Time Specialist* (Kahn and Gorry 1977) rely on some type of logic. Time Specialist is much like Chronos (Bruce 1972) in that it answers questions about the temporal characteristics of events. While Chronos relied on a temporal logic for its representation language, Time Specialist uses three separate temporal organizations: *date lines*, *before/after chains*, and *special reference events*.

The concept of a *date line* is simple. Each event maps to some time or date on a chronological line. Depending on the granularity of the date line and its associated events, a time specification consists of either a single time (or date) or two times (or dates) that specify an interval. A problem with date line representations is that many events, while occurring in time, have no explicit reference to time. Thus, we cannot represent statements like:

- I went to the grocery store.

in a date line organization.

Another method of organizing temporal events is as a linear sequence. These *before/after chains* easily capture situations like:

- You're born, you live, you die.

Not all temporal organizations are this simplistic, however. There are occasions when two events occur simultaneously and with a before/after representation, we would have to make an unacceptable decision on how to coerce the two events into a before/after ordering.

Furthermore, without additional information from the date line, we cannot determine the exact relation between two events. Referring to our example, does one stop living upon one's death or is there a period of decline between life and death? We again have to rely on the date line, but these relationships should be expressible without explicit date and time labelings.

Finally, the *Specialist*'s third temporal representation is *special reference events*, which is a special case of before/after chains. Instead of organizing arbitrary sequences of events, the user identifies special events, such as one's birth or today, and relates them to other pertinent events. Presumably, since these events are salient, the system will need to reference them often, thereby helping the reasoner avoid searching long before/after chains.

3.7 Event Calculus

Event Calculus (Kowalski and Sergot 1986) is a point-based temporal reasoning approach implemented within a logic programming framework for deductive database assertions and queries in a temporal domain. Kowalski and Sergot implement Event Calculus in Prolog and employ default reasoning schemes to handle indeterminacy and incompleteness of knowledge.

Programmatically, a problem with Event Calculus is that points, or events, represent the ending of an interval and the beginning of another. For example, if we know that a college fired Pat, an event, then the event ends the interval when Pat was working and begins the interval when Pat is not working. Thus, to establish the truth value of a proposition at an interval, we must look at one of the interval's bounding points and infer that the proposition holds for that interval.

For incomplete information, however, if we ask about the truth of a proposition of an interval preceding (or following) some event and that event does not explicitly state what proposition ended (or began), we must look back (or forward) in time to the event that established (or terminated) the proposition. Now if we cannot find the fact, we either have to fail or look further back (or forward) in time. Regardless, we potentially have to look in two or even more places and infer the truth of a proposition at an interval.[2]

This "jumping back and forth between events" gives rise to endless looping, which Kowalski and Sergot (1986:77) acknowledge. We can solve the problem by employing more sophisticated reasoning techniques, but this is only treating a symptom. The problem is the representation. If we use a more explicit interval representation, we have at least eliminated the problem of determining what is true at an interval. Summarily, we end up talking about intervals in the first place, why define them in terms of points?

3.8 McDermott's Temporal Logic

McDermott's logic for process and plans (1982) has become one of the most influential theories in temporal reasoning. His formalism serves as the basis for Dean's (1986) work in planning and problem solving, and Shoham's (1988b) work in nonmonotonic temporal reasoning. In this section we briefly discuss the salient aspects of his logic.

To begin, McDermott constructs a point-based interval logic. Time is dense and branches into the future while the past is linear. Every time has a *date*, which is a number on the real number line. Finally, McDermott posits *facts* and *events* as object types.

Facts are true either at a point or over an interval defined by two points. Instead of a fact being a predicate, McDermott stresses that his facts are functions which return the set of states (or times) at which the fact is true. For example, the fact

```
on(blockA, blockB)
```
[3]

denotes the set of times when `blockA` was on `blockB`. Thus, we can say that a fact is true at time `t1` by writing

```
T(t1, on(blockA, blockB)),
```

and true over an interval defined by time points `t1` and `t2` by writing

```
TT(t1, t2, on(blockA, blockB)).
```

Events are things that are happening. For instance, "the building is falling" is an event. One of the major distinctions of events is that they take time. A restriction on events is that they occupy the least amount of time. So given an interval over which an event occurs, there are no subintervals when the event is not occurring. We represent events or occurrences as

```
occ(t1, t2, event),
```

[2]See the SOURCE and DESTINATION axioms in Kowalski and Sergot (1986).

[3]I will use traditional logic syntax rather than McDermott's Lisp-style notation.

that is, `event` occurred over the interval defined by the time points `t1` and `t2`.

Much of why we need event-type constructs is because of the problems discovered in change-based formalisms. In such systems, we cannot represent the duration of events since we concentrate only on things that change. Assume we move a block from one location to another. In a change-based system we can only express that the block is at one location, the block is moving, and the block is at another location. With facts and events, we can represent the same information, but with the added expressivity of time information.

Another important aspect of McDermott's logic is his allowance of *persistence*; that is, some fact is true until it is not true. Yet, he allows a fact to persist for a specified *lifetime*. This notion seems counterintuitive to the concept of persistence as defined by others, notably Allen (1981) and Shoham and Goyal (1988). It seems that the point of allowing persistence is to capture those situations in which we do not know how long something lasts, so how can we affix a lifetime to such a situation?

McDermott also presents axioms for reasoning about causality, continuous change, actions, and problem solving. Furthermore, McDermott allows prevention inferences in which we can formulate rules to prevent another event's occurrence. For example, we can construct rules that express the fact that motorcycle helmets prevent head injuries. Another prevention example would be rules to prevent a water tank from overflowing.

3.9 Allen's Interval Logic

Allen (1984) takes a different approach to capturing notions of time than McDermott. These differences probably relate to their respective backgrounds. Allen is primarily a natural language researcher. Consequently, his theory is more intuitive and provides more natural tools with which to model temporal events. McDermott, on the other hand, states explicitly that his logic has no relation to natural language.

The other difference is in their primitives. While both allow intervals, McDermott constructs intervals from points, whereas Allen abandons points for a strict interval logic.[4] As a result, the normal ontological questions for a point-based logic become irrelevant or take on a different flavor. For example, time for Allen is linear, but it is linear in such a way that intervals flow forward in time or they have the property of *directedness* (van Benthem 1990).

Relating intervals, there are thirteen interval operators, presented in Figure 2.1. Additionally, Allen defines temporal transitivity axioms, such as:

```
before(i1, i2) ∧ before(i2, i3) ⊃ before(i1, i3)
```

to compute temporal relationships not explicitly expressed.

Allen's main predicate is

```
holds(p, i)
```

which is true if and only if *property* `p` holds during interval `i`. Properties are *homogeneous*; that is, if a property holds over an interval, then it holds over all subintervals.

Like McDermott, Allen posts several object types. Specifically he defines two classes called *processes* and *events*. Processes are happenings without an identifiable culmination, like "I am fishing." Events, on the other hand, have an identifiable goal. An example is "I am driving to Atlanta." Allen suggests that events are countable, whereas processes are not. For instance, we can enumerate how many times we have driven to Atlanta, but we cannot count the process "I am fishing." We can, however, count how many times "we went fishing." This is an event, however, not a process, because the happening culminates.

The characteristics we attribute to event intervals are somewhat different from those for property intervals. Recall that property intervals are homogeneous. Obviously, this is an oversimplification and does not encompass all temporal happenings. Therefore, event intervals by definition have no subintervals and occupy the smallest possible span of time. (This is similar to McDermott's definition of events.) For instance, if walking to school, then we cannot express any part of the event of walking to school without expressing the entire event.

[4] Galton (1990) critiques Allen's logic on this basis.

Processes are like properties in that a process may occur over subintervals, but different from them since a subinterval may exist when that process is not occurring. For example, if I am writing over an interval of time, there might be subintervals when I am resting or thinking.

We account for event causation by the `ecause` predicate, defined as

```
ecause(event1, interval1, event2, interval2),
```

which is true only if `event1`, occurring over `interval1`, causes `event2` which occurs over `interval2`. Allen restricts events from causing past events, intuitively.

An *action*, for Allen, is an agent performing or involved in some occurrence, characterized by the `acause` function, defined as

```
acause(agent, occurrence).
```

Actions that cause an event are *performances* whereas actions causing processes are *activities*. In this regard, an agent threatening to kill me, thus causing me to run would be an activity. If I ran to the police station, then the agent's threats would be a performance.

We return to Allen's logic in Chapter 6 when we discuss the implementation of TPS.

3.10 Non-Monotonic Temporal Logics

In Section 3.2, we discussed the formalism-independent problems in temporal reasoning. To solve the qualification problem, if the reasoner could decide that there is a problem or "jump to the conclusion" that there is a fault, then it could enter a diagnosis routine to determine what the problem is. This "jumping to conclusions," called *nonmonotonic reasoning*, has become a rich area of research, especially in temporal domains.

A problem known as the *Yale shooting problem*, which is a specific instance of an extended prediction problem, has become the litmus test of nonmonotonic temporal reasoning formalisms. The scenario involves Joe and a person with a gun. The reasoner knows the gun is loaded. After an indeterminate period of time, the person fires the gun at Joe. Obviously, Joe should die, but according to Hanks and McDermott (1985), of the nonmonotonic reasoning strategies they surveyed (McCarthy's (1980) Circumscription, Reiter's (1980) Default Logic, and McDermott and Doyle's (1980) Nonmonotonic Logic), none was able to conclude that Joe is dead.

Chronological Ignorance (Shoham 1988b) is a nonmonotonic temporal logic developed to solve some of the formalism-independent problems of temporal reasoning. Specifically, Shoham presents solutions to the qualification problem and the extended prediction problem. His solution to the qualification problem comes directly from his formulation of Chronological Ignorance. The extended prediction problem, however, required additional theoretic constructs, namely *potential histories*, which are strikingly similar to histories in the Hayesian sense. Shoham's potential histories, however, have a defeasible flavor, since they are only potential. Thus, several potential histories can describe a situation and any one could be true, depending on the sequence of events. The parts of potential histories that become reality are then manifested (Shoham 1988a). Other than Shoham's publications on Chronological Ignorance, I found no other discussions of his work.

Chapter 4

Related Systems

In the previous chapter, we surveyed several temporal reasoning systems. While I would place these systems in the same classification as TPS from the standpoint of temporal reasoning systems, I do not consider them as "related systems." They are foundational systems.

The distinction lies in the purpose of the systems. I classify these systems as research vehicles and do not consider them appropriate for placement in a working environment. TPS is an *expert system shell* with extensions to allow temporal domain modeling. Few expert system shells exist that fit this definition. Most that do are *real-time expert systems shells.* These are expert systems shells designed to model continuously changing temporal events, like the water flow rate through a pipe. Because of their ontological conventions for capturing continuous change, these systems are limited to those domains.

An expert system shell is an AI tool that allows domain modeling and problem solving at "expert" levels of performance. Further, a *real-time expert system* is a traditional expert system in terms of performance expectations, but targets domains in which information comes to the system continually as time progresses. Monitoring the temperature of a nuclear reactor core is an ideal domain a for real-time expert system. The term *expert*, however, is extremely subjective; nevertheless, an expert system should at least contain expertise and knowledge greater than that of its intended user.

In the following sections, we discuss the expert system shells with temporal reasoning that most closely relate to TPS.

4.1 Lockheed Expert System Shell

Perkins and Austin (1990) added temporal reasoning to the Lockheed Expert System Shell (LES), a frame-based expert system shell. Their characterization of time and consequent implementation directs their system toward trend analysis of past continuous events. They describe an example in which they develop a program to diagnose problems with the Hubble telescope's "reaction wheels" from telemetry data. The system allows four temporal operators (`before`, `during`, `after`, and `at`), relational qualifiers (e.g., `anytime`, `alltimes`), and data manipulation functions for incomplete information (e.g., interpolation functions). Unfortunately, the system requires us to give every event a time label, thereby limiting what we can model since we cannot always associate events with a time or date.

4.2 G2

The G2 Real-Time Expert System (Moore et al. 1989) is strikingly similar to LES from an ontological point of view. Both concentrate on analyzing and reasoning about past, continuous events.

The main temporal constructs in G2 are *history expressions*, which are temporally sequenced attribute values. An example is a sequence of water flow rate measurements sampled at one minute intervals for ten minutes. Queries on histories consist of three elements: a function, a variable, and a time period. Intuitively, a variable is the symbol for the history expression we are investigating, such as the `water flow rate`.

G2 provides nine temporal functions whose domain is history expressions. These functions perform various interpolation, integration, and analysis functions on histories. Rather than present all nine functions and their explanations, I will present some illustrative examples following a discussion on G2's time periods.

A *time period* in G2 is a subset of a history expression whose length is a time interval. Three temporal operators, **during**, **between**, and **as-of**, designate what time interval to select from a history expression. We can express these intervals in any combination of weeks, days, hours, minutes, and seconds. Finally, arithmetic expressions allow the calculation of time intervals.

Using our water flow rate history expression, we can construct several queries to determine information about this ongoing process. Recall that we sampled the flow rate every minute, and consider the query:

```
the interpolated value of the water flow rate
as of 3 minutes and 30 seconds ago
```

Because of our sampling frequency, no value exists at this time. Thus G2 performs a straight line interpolation to determine the value. Finally,

```
the rate of change per minute of the
water flow rate between 7 minutes
and 3 minutes ago
```

returns how quickly the water flow rate changed for the period indicated.

4.3 Ariño's System

Ariño (1989) presents a knowledge-acquisition system that automatically extracts knowledge from newspaper articles and incorporates it into a knowledge-base and a temporal system. Specifically, the domain is the information in natural language texts from the Economics section of the Spanish newspaper 'El País'. Ariño's paper focuses on the relation between time and language and the temporal system required to represent these relations.

To model the temporal characteristics of the domain, the system uses three temporal constructs: points, intervals, and chains. Points relate to days, the smallest granule of time (e.g., 18 January 1987). A series of points constitutes an interval (e.g., 15 May to 20 May), and a non-consecutive, ordered series of intervals compose a chain (e.g., to represent a history of economic events).

To associate time and language, Ariño uses Hornstein's (1977) theory of linguistic expression of time. According to Ariño, Hornstein associates three identifiable temporal elements with a sentence: the speech moment (denoted S), which relates to the publication date of the newspaper article; the reference moment (or R), or the period during which an event happens; and the event moment (E), which is the point at which some event occurs. In Ariño's framework, temporal events can be either points or intervals. For example, a newspaper article published on 5 May 1991 could have stated that an economic conference might be scheduled after 1 June 1991. In this example, the speech moment (S), a point, is 5 May 1991; the reference moment (R), an interval, begins on 1 June 1991; and the event moment (E), also a point, occurs sometime during the reference interval. Figure 4.1 represents this situation.

Ariño adds a further distinction between temporal occurrences. Closed temporal events have a direct reference to a date. Since the event moment (E) lacks a direct reference to a date, it is open. Conversely, events (S) and (R) are closed; they have associated dates.

After identifying the temporal events, the system derives relations from these happenings. The system determines temporal relationships by examining linguistic elements of the text, such as verb tense (past, present, or future), temporal adverbs (yesterday, today, tomorrow), and explicit dates (5 May 1991). Referring to our example, the system would generate the following relations:

```
S before R
S before E
E during R
```

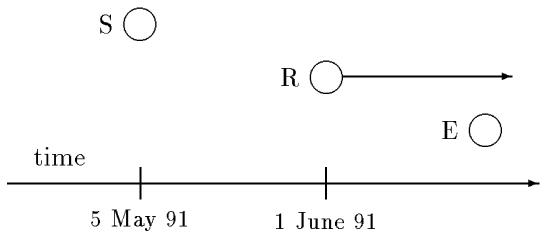


Figure 4.1: Temporal events over time

Thirteen total temporal operators relate events (see Figure 2.1). Now the system can enter the acquired and processed information into the knowledge-base and temporal constructs.

Ariño uses three knowledge representation structures to capture factual and temporal information: a frame-based knowledge representation, a temporal network, and a calendar. Frames store factual information derived from the newspaper articles. In our example, a frame encodes all pertinent atemporal information about an economics conference.

The temporal network stores the relations between events derived by the system from the article. The nodes of the network represent events (i.e., S, E, or R), and the links denote the relations between events (see Figure 4.2).

The third knowledge representation structure in this system is the calendar, realized as a tree of dates. Links exist between the closed instances in the temporal network and the date nodes of the calendar. For example, nodes S and R in Figure 4.2 would be linked to the calendar nodes 5 May 1991 and 1 June 1991, respectively.

Whenever the system receives new information, besides the usual incorporation of factual knowledge, special inference procedures maintain and update the temporal network. Regarding our example, assume another article later appeared in the newspaper about the same economic summit that makes a specific reference to the date of event. Now the inference procedure must update the open reference for the original story dated 5 May 1991. In other words, the special inference procedure propagates temporal relations through the network in an attempt to close any open events. Closing an event involves deducing and constructing a direct reference from an open event to a calendar node. Such updating of the temporal network guarantees knowledge-base closure during queries. Now if a user makes a query about the 5 May 1991 article, because of updating, the system would infer the actual date of the event even though at acquisition time that fact did not exist.

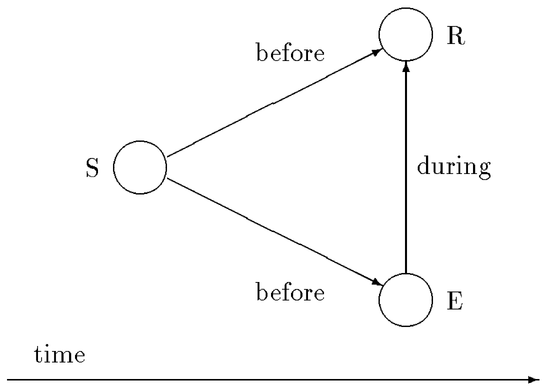


Figure 4.2: A temporal network.

Chapter 5

Production Systems

Production systems have received immense popularity in Artificial Intelligence as a knowledge representation and a computational tool. These systems relate directly to automata theory and also have applications in mathematics, natural language, and cognitive psychology. This chapter begins with a brief historical account of the development of production systems, and concludes with a detailed discussion of OPS5 and the Rete Match Algorithm, influential contributions to production system computation as well as Artificial Intelligence.

5.1 Formal Beginnings

Production systems date back to 1943 when Post (1943) formulated *canonical systems.* Chomsky (1959) adopted a similar formalism called *type 0 grammars* to represent and generate natural language. Since type 0 grammars are equivalent to Turing Machines, they have become a cornerstone in automata theory. Hopcroft and Ullman (1979) define a type 0 grammar, or an *unrestricted grammar*, G as the 4-tuple $G = (V, T, S, P)$ where

V is a finite set of variables,
T is a finite set of terminals,
S is the start symbol $S \in V$, and
P is a finite set of productions of the form $\alpha \rightarrow \beta$,
where $\alpha, \beta \in (V \cup T)^*$.

We can define a *step* as $\gamma\alpha\delta \Rightarrow \gamma\beta\delta$ such that $\gamma, \delta \in (V \cup T)^* \cup \{\epsilon\}$, where ϵ is the empty string, if and only if $\alpha \rightarrow \beta \in P$. A *derivation* is therefore the reflexive and transitive closure of $\Rightarrow$, denoted $\overset{*}{\Rightarrow}$. If $S \overset{*}{\Rightarrow} \gamma$ for some $\gamma \in T^*$, then we can say that γ is in the language of the grammar G. Moreover, Newell and Simon (1973) adopted production systems as models of human cognition, and in AI, Feigenbaum first used production rules to encode domain-specific knowledge in DENDRAL (Buchanan and Shortliffe 1984).

5.2 Production System Architecture

Production systems consist of three components: *long-term memory*, *short-term memory*, and an *executive*, or *rule interpreter.* Long-term memory is the space or area where rarely-changing knowledge resides. Productions or *if-then rules* consist of a left-hand, conditional side (LHS) and a right-hand, action side (RHS),and represent long-term knowledge. Short-term memory or *working memory* is the area or space where the executive stores intermediate results. Finally, the executive manages the computational process. In terms of grammars, it carries out the derivation.

Production system computation consists of three steps: *match*, *resolve*, and *act.* In the match step, the executive algorithm matches patterns in working memory with the LHS patterns of the rules in long-term memory. Whenever all LHS patterns of a rule match patterns in working memory, the executive adds the rule to a *conflict set*. Invariably, more than one rule in long-term memory matches, resulting in several possible rule firings. Thus some scheme has to determine which rule actually fires. In this sense, production system computation is nondeterministic.

The scheme, called *conflict resolution*, selects one rule from the conflict set based on some predetermined strategy. Examples of conflict resolution strategies are:

- fire the first rule selected,
- fire the rule that has the oldest short-term memory patterns,
- fire the rule that fired last.

Once the conflict resolution scheme selects a rule, the executive acts upon the rule's RHS. Typically, these actions include adding, deleting, or modifying working memory patterns. Alternatively, a rule could ask the human user for input or notify the user of some result.

Production system computation described thus far has been completely forward chaining, or data-driven; that is, given some initial patterns in working memory, the executive applies the rules until no productions are true. Then, the result is whatever patterns are present in working memory. Conversely, a production system can also use goal-driven, or backward chaining, inference; that is, given a potential answer, the executive applies the match algorithm to the RHS of rules until the system reaches a valid starting configuration. MYCIN (Buchanan and Shortliffe 1984), for example, uses backward chaining inference.

5.3 General-Purpose Shells

As specific-purpose production systems evolved and proliferated, more generic *shells* appeared, such as EMYCIN (van Melle et al. 1984) and the Official Production System (OPS) series (Forgy and McDermott 1977). These shells provided system designers with the inference mechanisms and conflict resolution strategies required for intelligent system construction in a software package that optimized development time. EMYCIN stands for "Essential MYCIN" or "Empty MYCIN," which is MYCIN minus its domain knowledge. Furthermore, instead of simple RHS actions, shell designers added more sophisticated ones, like displaying a graphic or reading a data acquisition device.

A drawback of production systems became evident early. As applications grew in complexity, the size of the rule-base increased and the number of working memory elements increased, resulting in unacceptable inefficiency in the match-resolve-act cycle. This prompted Charles Forgy to investigate more efficient methods of pattern matching.

5.4 The Rete Match Algorithm

Forgy (1982) noticed and subsequently exploited two properties of production system computation that led to a more efficient pattern matching algorithm. First he recognized that in most applications the rule-base rarely changes and there is great *structural similarity* between rules. In other words, as we consider each condition element of every rule, we can partition the rules based on the similarity between condition elements.

The second property Forgy exploited was *temporal redundancy*. Typically, working memory changes slowly with respect to time. Time here has a different definition than in the temporal reasoning sense. If we assume one match-resolve-act cycle is a time unit, then few working memory patterns change from cycle to cycle, or over time.

Existing production systems re-computed the conflict set every cycle from scratch. Not only was this practice inefficient, but it also disregarded the fact that many rules in the conflict set, depending on working memory updates, would have the same status within the conflict set from one cycle to the next. Forgy thought that we should compute how the conflict set *changes* as computation updates working memory.

Forgy's implementation of this idea is the Rete Match Algorithm. The Rete Algorithm "compiles" a rule-base into a tree structure, or network, whose interior nodes are simple comparisons. The leaves of this tree are the rules. All paths between the root and a given leaf node represent the comparisons that require satisfaction for a rule to fire. Because of the structural similarity of the rules and the tree structure of the network, we can reuse comparison nodes and construct a minimal representation of the rule-base.

Additionally, notice that deletion and addition are the only operations necessary for maintaining working memory—modification is simply a deletion followed by an addition. As computation asserts and retracts patterns to and from working memory, Rete propagates these patterns through the comparison nodes of the network. If this activity leads to a leaf node (i.e., a rule node), then the algorithm inserts the rule into the conflict set according to how many working memory patterns the rule matched.

5.5 OPS5

OPS5 (Forgy 1981) is available on a wide range of computing platforms which demonstrates its viability as a expert system shell. While OPS5 was the first AI tool with the Rete Match Algorithm, several other production systems also use it. Examples include OPS83 (Sherman and Martin 1990:162); Automated Reasoning Tool (Firebaugh 1988:406), or ART; and, most recently, NASA's C Language Integrated Production System (Lopez et al. 1987), or CLIPS.

The following treatise on OPS5 language syntax, while not exhaustive, should provide enough of a basis to understand the remaining discussions. Sources that present the OPS5 language in detail include Brownston et al. (1985), Cooper and Wogrin (1988) and Sherman and Martin (1990).

For efficiency, OPS5 establishes a *canonical ordering* for working memory patterns using the `literalize` command. For example, persons in our application domain might correspond to the literalization:

```
(literalize person name age)
```

where `person` is the *object class* and `name` and `age` are *attributes* of the class.

To enter a pattern into working memory, we use the `make` command. Assuming our person's name is Pat and Pat is twenty-five years of age, we have the following command:

```
(make person ↑name pat ↑age 25).
```

The ↑, or tab, character identifies an attribute symbol. Because of the canonical ordering imposed on working memory patterns, we are free to omit unspecified attributes. We therefore could have just as easily written:

```
(make person ↑name pat)
```

or

```
(make person ↑age 25).
```

Unspecified attribute values are `nil`.

OPS5 production rules are of the form:

```
(p <symbol>
     <condition-list>
     →
     <action-list>)
```

where `<symbol>` is the name of the rule, `<condition-list>` is a list of patterns, and `<action-list>` is a list of valid OPS5 actions.

We can understand how to construct rules best with an example. Assume working memory looks as follows:

```
(person ↑name pat ↑age 25)
```

Then, a trivial rule is:

```
(p trivial-rule
   (person ↑name <who> ↑age <howold>)
   →
   (write (crlf) <who> | is |<howold>| years old (crlf)))
```

An OPS5 variable is any symbol bracketed with < and >. So, <`who`> is a variable. The LHS of this rule matches the pattern in working memory since both the working memory pattern and the condition element have the symbol `person` in the first position. Consequently, `pat` binds to the variable <`who`> and `25` binds to <`howold`>. Since the pattern in working memory satisfies the rule's condition, the executive places `trivial-rule` into the conflict set. To fire the rule, the executive acts upon the RHS action of `trivial-rule`, which writes

```
pat is 25 years old
```

on the current output device.

Because of how the Rete Algorithm works, we must organize OPS5 programs as follows:

```
literalizations
⋮
rules
⋮
make commands
```

5.6 Rete: The Details

We have already discussed the general characteristics of the Rete Algorithm and since we understand the basics of OPS5 language syntax, we will concentrate on Rete's internal workings. Again, this is not intended to be a detailed presentation of Rete. Sherman and Martin (1990) and Cooper and Wogrin (1988) present excellent in-depth descriptions of how Rete works. Recall our trivial example:

```
(literalize person name age)

(p trivial-rule
   (person ↑name <who> ↑age <howold>)
   →
   (write (crlf) <who> | is | <howold> | years old (crlf)))

(make person ↑name pat ↑age 25)
```

Looking at the literalization, we define a person class. The canonical ordering begins with the class type, thus `person` is in position one with `name` and `age` following in positions two and three.

Rete "compiles" a rule-base into a network of simple comparisons and propagates added and deleted working memory patterns through the network. The leaf nodes of the network are rules, so if the propagation of patterns reaches a rule node, we can add the rule to the conflict set.

Referring to the first condition of `trivial-rule`, there is one test that requires satisfaction for the rule to fire: the class name of the pattern must be equal to `person`. If a working memory pattern meets this criteria, then we bind its attribute values for `name` and `age` to the variables <`who`> and <`howold`>.

We denote tests using the following mnemonic:

`t` <*test-type*> <*argument-type*>

where <*test-type*> is `eq` for equals, `gt` for greater than, `lt` for less than, and so on; <*argument-type*> is `n` for numeric, `a` for atomic, and so on. Thus, we can express the required test as

```
teqa 1 person
```

which succeeds if a working memory pattern's class is equal to `person`. More precisely, this test succeeds if a working memory pattern's first atomic element is equal to `person`.

The network representation of `trivial-rule` appears in Figure 5.1. The `&bus` node serves as the root node of the network. A `&mem` node retrieves a value from an internal register and binds it to a variable. For example, the node

$$\begin{tikzcd}
\texttt{\&bus} \arrow[d] \\
\texttt{teqa 1 person} \arrow[d] \\
\texttt{\&mem 2 *c1*} \arrow[d] \\
\texttt{\&mem 3 *c2*} \arrow[d] \\
\texttt{\&p trivial-rule}
\end{tikzcd}$$

Figure 5.1: Network representation for **`trivial-rule`**.

```
&mem 3 *c2*
```

takes the value from the `*c2*` register and binds it to the variable indexed by location `3` of the current condition element. **`&p`** nodes are the rule nodes.

Now that we have a compiled rule-base, we issue the **`make`** command:

```
(make person ↑name pat ↑age 25).
```

Rete loads **`person`** into the first register, **`pat`** into the second, and **`25`** into the third, and then traverses the rule network attempting to satisfy the simple conditions at each node. When the procedure reaches the **`&p`** node, as in this case, it adds **`trivial-rule`** to the conflict set.

To execute an OPS5 program, we simply take the first rule from the conflict set and evaluate its RHS actions. Any additions or deletions to working memory during execution will change the conflict set, but the first rule in the set is always the next rule to fire.

Chapter 6

Temporal Production System

The preceding chapters began by giving us a broad perspective on how and in what areas of computer science and AI temporal logics and reasoning apply. These chapters have gradually prepared us for this chapter. I begin by presenting an illustrative example that justifies the approach taken to implement TPS. From there, we investigate how TPS represents and maintains temporal knowledge, followed by discussions on the syntactic interface to the temporal reasoner and the modifications to the Rete Match Algorithm. The chapter concludes with applications of TPS.

6.1 Raison d'être

On the surface, it appears that much of what TPS does could be done within a conventional rule-base system. For some problems, this approach would work. As the complexities of these problems increase, however, so do the required solutions. To illustrate this point, we will step through what would be required to duplicate the function of TPS with OPS5.

6.1.1 The Specific Case

The first approach we consider is the case in which we explicitly specify all temporal aspects of a domain. For simple applications, this approach would work. The flaws of this approach become apparent when we need to update temporal relations based on new clarifying information. For variety, consider a football example. Assume our production system knows that the first half of a football game precedes the second half. It does not know, however, whether some interval occurs between these halves. So, both potential situations must be represented, as follows:

```
firsthalf — (before meets) → secondhalf
```

In OPS5, we represent this situation with the following literalization:

```
(literalize relation operator event1 event2)
```

Next, we make the following working memory patterns:

```
(relation ↑operator before ↑event1 firsthalf ↑event2 secondhalf)
(relation ↑operator meets ↑event1 firsthalf ↑event2 secondhalf)
```

Now we are free to formulate our production rules as required. Yet, consider what happens if some rule later asserts the following working memory patterns:

```
(relation ↑operator meets ↑event1 firsthalf ↑event2 halftime)
(relation ↑operator meets ↑event1 halftime ↑event2 secondhalf)
```

That is, **halftime** occurs between **firsthalf** and **secondhalf**. Now we know the relationship between first and second halves is **before**. We therefore need some mechanism to update temporal relationships. Again, we can explicitly develop rules to update only this situation. Not only is this poor development practice, but it also confounds our goal of developing a general system to model temporal events.

The obvious solution is to use the production system's rules to develop this general strategy. We analyze this solution in the next section.

6.1.2 The General Case

To automatically maintain and update temporal relationships, we must first have mechanisms that compute the *temporal transitivity*. For example, if the **firsthalf** meets **halftime** and **halftime** meets the **secondhalf**, then the **firsthalf** is before the **secondhalf**. Given the literalization,

```
(literalize relation operator event1 event2)
```

assume we have three events related as follows:

```
(relation ↑operator meets ↑event1 firsthalf ↑event2 halftime)
(relation ↑operator meets ↑event1 halftime ↑event2 secondhalf)
```

To compute the relationship between **firsthalf** and **secondhalf**, we need the general rule:

```
(p meets-transitivity
    (relation ↑operator meets ↑event1 <e1> ↑event2 <e2>)
    (relation ↑operator meets ↑event1 <e2> ↑event2 <e3>)
  - (relation ↑operator before ↑event1 <e1> ↑event2 <e3>)
    →
    (make relation ↑operator before ↑event1 <e1> ↑event2 <e3>))
```

Furthermore, we would also need rules to compute the inverse temporal relations, as well as the inverse transitivities:

```
(p meets-met-by
    (relation ↑operator meets ↑event1 <e1> ↑event2 <e2>)
  - (relation ↑operator met-by ↑event1 <e1> ↑event2 <e2>)
    →
    (make relation ↑operator met-by ↑event1 <e1> ↑event2 <e2>))

(p met-by-meets
    (relation ↑operator met-by ↑event1 <e1> ↑event2 <e2>)
  - (relation ↑operator meets ↑event1 <e1> ↑event2 <e2>)
    →
    (make relation ↑operator meets ↑event1 <e1> ↑event2 <e2>))

(p met-by-transitivity
    (relation ↑operator met-by ↑event1 <e1> ↑event2 <e2>)
    (relation ↑operator met-by ↑event1 <e2> ↑event2 <e3>)
  - (relation ↑operator met-by ↑event1 <e1> ↑event2 <e3>)
    →
    (make relation ↑operator met-by ↑event1 <e1> ↑event2 <e3>))
```

Since there are thirteen temporal relations, we need thirteen rules to calculate the inverse relations and another 169 rules to calculate all possible temporal transitivities.

The next problem arises when we attempt to capture uncertain temporal relationships. Often we need to express a relationship between two events as a vector of two (or more) relations. We saw this with our football example before we knew that halftime occurred between first and second halves. We can use OPS5's vector-attribute data type to represent these relationships. Expressing that the **firsthalf** either is before or meets the **secondhalf** requires the following definitions:

```
(vector-attribute operator)
(literalize relation operator event1 event2)
⋮
(make relation ↑operator before meets
               ↑event1 firsthalf ↑event2 secondhalf)
```

Naturally, the transitivity rules require modification to take into account the vector-attribute. This alteration does not affect the quantity of temporal transitivity and inverse rules; however, OPS5 does not provide functions to retrieve vector-attribute values, other than the first. So, we would have to develop additional rules to search and inspect these fields, all at the added cost of computation.

In fact, the computational overhead associated with this approach is tremendously high. Burdening the production system with an application will only magnify the inefficiencies. The better approach is to develop these temporal knowledge maintenance routines outside of OPS5 in Lisp and develop an interface, accessible from OPS5, to the temporal knowledge system. This is exactly the approach taken in developing TPS, which provides an efficient, general, transparent method of maintaining and updating temporal knowledge.

6.2 Ontological Considerations

While I do not attempt to give the full ontology for TPS, I do present its salient features, especially those relevant to implementation. After discussing what primitive TPS allows, we turn toward the relations that exist between primitives, and conclude with a brief analysis of the nature of time.

6.2.1 Temporal Primitives

Without delving too deeply into historical literature, several issues require consideration before we develop an ontology for a temporal reasoning system. First we should identify the target domains. I intend TPS for use in observable domains; that is, domains in which objects and events have temporal extent.

Next we must ask what temporal primitives these domains require. For any serious temporal reasoning system in an observable domain, we must use some type of interval representation. In fact, we do not need points whatsoever. Furthermore, point-based representations have computational detractions. We explored some of these with Event Calculus in Section 3.7, but the main argument against points comes from deriving intuitive temporal operators.

Since we intend system designers to use TPS, we need to provide a rich set of *intuitive* operators. These operators should be intuitive in the sense that a designer would use an operator naturally when describing a situation. We want to avoid requiring knowledge engineers to learn a cryptic language.

To establish intuitive temporal relations (e.g., **`before`**, **`during`**, and **`after`**) in a point-based interval logic, we must make end-point computations. Each interval i has a lower point (denoted $i-$) and an upper point (denoted $i+$). For two intervals i and j, defining i **`before`** j is straightforward: $i+ < j-$. We can define **`after`** similarly. Defining i **`during`** j requires:

$$((i- > j-) \wedge (i+ < j+)).$$

Making these repeated computations inevitably degrades system performance. Allen's (1983, 1984) formalism provides us with a pleasant alternative. We can avoid end-point computations altogether by directly dealing with intervals. The resulting representation is programmatically cleaner than point-based representations since no computation is necessary to derive operators.

Although we have abandoned points, there are occasions when an assumed *granularity of time* exists. Language provides an illustrative example. For instance, when discussing the evolution of the universe, an assumed granularity would be on the order of millions of years, not of seconds. In TPS, one can define an appropriate granularity of time for a given domain. This allows an atomic notion of time without committing to counterintuitive instantaneous time moments.

It is important for a temporal reasoning system to have some concept of *persistence* (Allen 1981); that is, some assertion holds from now on. To capture this notion, we can attach attributes, such as **`start`**, **`end`**, and **`extent`**, to intervals. We can therefore assign an infinitely large time value to the **`end`** attribute to specify persistence. The system must then observe constraints requiring that no temporal interval bound the persistent interval in the future.

Likewise, a type of persistence from the past should also exist; that is, some proposition has been true forever until now. To accomplish this, we assign an infinitely small time value to the **`start`** attribute. Again, the reasoner must observe certain relationship constraints to preserve persistence.

These attributes also anchor intervals to a time/date line. Although the system does not explicitly use these attribute values for reasoning, they are available for arithmetic comparisons and computation. Recall that the system described by Perkins and Austin (1990) requires that all events relate to a time line. Since we cannot always anchor events to a time line, a temporal reasoning system should perform without explicit mention of time (Allen 1981). We must model the relationship between events in time, not the time itself.

6.2.2 Temporal Relations

What we have so far is a collection of intervals to which we can ascribe certain attributes. Now we need to describe the relations that can exist between intervals. While the traditional complement of interval relations is thirteen (Allen 1983; van Benthem 1990), other researchers have reduced or increased the number of relations to meet computational or ontological requirements (Vilain 1982; Zhu et al. 1987). TPS, however, uses the original set of interval relations, presented in Figure 2.1.

6.2.3 The Nature of Time

The most common time structures used in temporal reasoning systems are linear time and variations of branching time. With the former case, there exists a complete ordering of time. On the other hand, with branching time structures, this ordering is partial. Typically, we consider the past as determined and therefore linear.

So, what is the nature of time for TPS? To begin, Allen (1984:131) adopts a linear time structure for his formalism, which underlies TPS. Yet, the computing mechanism of the production system must affect the nature of time.

The rules of the production system form a decision tree. During execution, the program follows a branch of this tree to a solution depending on starting conditions and what facts become available during execution. We can view the branching structure of the decision tree as a kind of *possible-worlds model* (Chellas 1980). Each decision point is a world in which certain things are true. From any world, there are several possible worlds into which we could move, depending on what production rules fire.

In order to posit branching time, we would have to identify different times with each possible world. Since the production system computation forms a branching structure that is atemporal, the temporal model must be responsible for associating a different time to each possible-world. The model possesses no ability to do this. In TPS, there is a strong separation of the temporal reasoning system and the production system. Since I attribute the branching structure to the production system, which is loosely coupled to the temporal system, I find difficulty in positing a branching time structure and therefore adopt a linear time model.

Given this view of linear time coupled with the possible-worlds model, we get a time structure called *pseudo-branching time* (Nute 1991). In this model of time, every possible world, or decision point, that is temporally equidistant from a time of reference associates to one time from the totally ordered set of all times.

Finally, we need to define the concept of *now* for TPS. A *time map* (McDermott 1982) is to a temporal database of past or developing events. The TPS time map focuses on the temporal organization of events through the past, present, and future rather than on their relationship to *the present*. So, defining *now* involves finding some temporal point of reference that divides the time map into past and future. I assert that the current rule-set does this. Depending on how one models a domain, that is by looking at past events, future events, or a combination of both, we could situate *now* at almost any time relative to the time map.

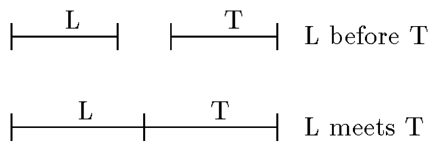


Figure 6.1: What is the proper relation?

6.3 Time Map Representation

Following Allen (1983), I used a directed graph whose vertices are intervals and whose arcs are vectors of temporal relations. Atemporal components of assertions reside in working memory with links to the time map. For example, to represent that Bill worked during the period that Mary worked, working memory contents would be:

```
(person ↑name bill ↑activity worked ↑at interval1)
(person ↑name mary ↑activity worked ↑at interval2)
```

and the time map would be:

```
interval1 — (during) → interval2
interval2 — (contains) → interval1.
```

Note that as the system asserts new relations into the time map, it also adds the corresponding inverse relations. Association lists represent the time map. For example, the relation

```
interval1 — (during) → interval2,
```

has a Lisp representation as follows:

```
(setf (get 'interval1 'relations)
          '(interval2 (during)))
```

6.4 Temporal Knowledge Maintenance

As TPS asserts new temporal relations into the time map, there is the potential for other, already present, relations to change in light of the newly added information. Recall that Allen (1981) recommends that a model of time handle uncertain temporal relations since we are not always sure of the true relationship that exists. For instance, suppose that, during a storm, thunder always follows lightning. We will say that we have an interval when thunder occurs and an interval when lightning occurs. Yet, what relation exists between these two intervals? (See Figure 6.1.) If we say that the intervals meet, then can we be sure that some amount of time does not separate the periods. And if we use a before/after relation, what if there is no time intervening? The solution is to use both relations, so the time map representation is as follows:

```
lightning — (before meets) → thunder
thunder — (after met-by) → lightning.
```

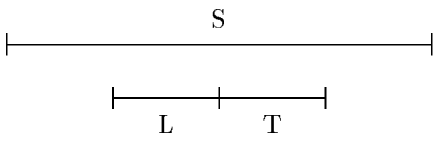
As computation proceeds, assume additional information clarifies the relation between these periods and we can infer that the periods do indeed meet each other. In this situation, we must have some mechanism to update the time map as the system presents additional information.

We maintain these relations using a general technique called *constraint propagation* (see Kumar 1992). Consider our thunder and lightning scenario again; except now, we add the information that these events occur during a storm. For the sake of illustration, assume that thunder immediately follows lightning. If we symbolically represent this situation, we have:

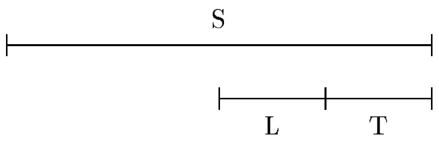
`S ← (during) — L — (meets) → T.`

Now what can we say about the relation between the storm and thunder? To determine this relationship, we must consider all the relationships that could exist between the storm and thunder given the constraints that lightning occurs during the storm, and thunder immediately follows lightning. We have three situations:

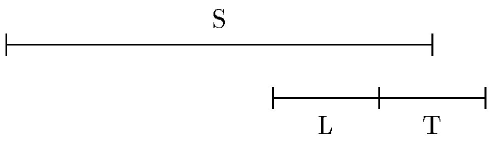
1. the lightning and thunder both occur during the storm,



2. the thunder *finishes*[1] the storm, and



3. the storm overlaps the thunder.



Thus, the vector between S and T is:

`S — (contains finished-by overlaps) → T`

and its inverse is:

`S — (during finishes overlapped-by) → T`

The precise algorithm for propagating these temporal constraints is as follows. First, we must be able to calculate the transitivity between any two relations, denoted $T(r1, r2)$, where $r1$ and $r2$ are temporal relations. Using Allen's (1983) transitivity table, I constructed a simple look-up function using association lists to implement $T(r1, r2)$.[2]

Next, we must calculate the overall temporal constraint vector using the Constraints algorithm, presented in Figure 6.2. Finally, using the Add algorithm, presented in Figure 6.3, we add the intervals and their relations to the time map by propagating the temporal constraints between all 3-connected neighbors in the time map. The Add algorithm attempts to keep all time map relations consistent. Hence, determining whether a relationship holds between two intervals involves retrieving the relation vector that exists between the intervals and checking if the relationship is present in the vector.

As Vilain and Kautz (1986) demonstrate this constraint propagation algorithm is sound, but it is not complete. Furthermore, it operates in $O(n^3)$ time and $O(n^2)$ space, where n is the number of temporal intervals in the time map. Unfortunately, an algorithm that is complete is $\mathcal{NP}$-hard. Therefore, one can either accept the incompleteness of the polynomial time algorithm, as I do, or make some ontological adjustments, as Vilain and Kautz do. Valdés-Pérez (1987), however, presents a sound and complete propagation algorithm that still grows exponentially, but uses backtracking and search tree pruning to gain efficiency.

[1] This is *finishes* in the temporal relation sense.

[2] I also investigated implementing $T(r1, r2)$ with hash tables, but for ten thousand worst-case retrievals, found a one and one-half seconds difference in favor of hashing, but decided to remain with association lists.

```
Constraints ($R1, R2$)
  $C \leftarrow \emptyset$
  for each $r1 \in R1$ do
    for each $r2 \in R2$ do
      $C \leftarrow C \cup T(r1, r2)$;
  return $C$;
end; { Constraints }
```

where

$R1$ and $R2$ are vectors of temporal relations,
$\emptyset$ is the empty set, and
$\cup$ is the set union operation.

Figure 6.2: Constraints Algorithm (Allen 1983).

```
Add ($R(i,j)$)
  $queue \leftarrow R(i,j)$;
  while not empty($queue$) do
    $R(i,j) \leftarrow queue$;
    $N(i,j) \leftarrow R(i,j)$;
    $Intervals \leftarrow Intervals \cup \{i,j\}$;
    for each $k \in Intervals$ do
      if $N(k,j) = \emptyset$ then
        $N(k,j) \leftarrow$ all-relations;
      $R(k,j) \leftarrow N(k,j) \cap constraints(N(k,i), N(i,j))$;
      if $R(k,j) \subset N(k,j)$ and $R(k,j) \neq \emptyset$ then
        $queue \leftarrow R(k,j)$;
      if $N(i,k) = \emptyset$ then
        $N(i,k) \leftarrow$ all-relations;
      $R(i,k) \leftarrow N(i,k) \cap constraints(N(i,j), N(j,k))$;
      if $R(i,k) \subset N(i,k)$ and $R(i,k) \neq \emptyset$ then
        $queue \leftarrow R(i,k)$
    end do
  end while
end; { Add }
```

where

$R(i,j)$ is the temporal relation vector between intervals i and j,
$N(i,j)$ is the temporal relation vector between intervals i and j
in the time map,
all-relations is the complete set of temporal relations
(`before`, `after`, ..., `equal`),
Intervals is the set of all interval labels in the time map,
$\subset$ is the proper subset function,
$\cap$ is the set intersection operation,
$\cup$ is the set union operation, and
$\emptyset$ is the empty set.

Figure 6.3: Add Algorithm

6.5 Syntactic Interface

TPS required few syntactic modifications to OPS5. Most were assertion and retrieval functions for the temporal reasoner.

Although temporal reasoners should function without explicit mention of time, it is true that they need some concept of time. The first command that allows us to attach time and date characteristics to intervals is the **granularity** command, which sets a lower bound on the length of intervals. It should appear in the literalizations section of a TPS program[3] and has the following syntax:

```
(granularity [year] [month] [day] [hours] [minutes] [seconds])
```

For example,

```
(granularity minutes seconds)
```

specifies that the smallest interval duration is one second. Additionally, the granularity command establishes a format for the specification of interval values, like **start** and **end**. In this case, we express all interval attribute values in a minutes/seconds format.

The **literalize** command required modification to recognize pattern literalizations with links to the time map. If the keyword **at** appears in a literalization, then the working memory pattern will have a link to the time map. Therefore, the value of the **at** attribute is an interval label. For instance,

```
(literalize person name at)
```

establishes that the person class, while having a name, will also have an association to the time map.

Turning our attention to rule conditions, we need mechanisms to match patterns in the time map. TPS features two such functions. The first, which we will call a *temporal query*, consists of a temporal relation followed by two interval labels. It succeeds if the relation exists between the intervals in the time map. For example,

```
(during interval1 interval2)
```

succeeds if, according to the time map, **interval1** is during **interval2**.

The second query function, called a *temporal vector query*, succeeds if any temporal relationship in a specified vector exists between two intervals. Although this construct is easy to simulate with multiple rules and temporal queries, it is less concise and inefficient. An example of a temporal vector query is:

```
([before during after] interval1 interval2)
```

which succeeds if, according to the time map, **interval1** is before, during, or after **interval2**.

The **make** command also required modification to properly handle intervals and attributes. We specify intervals the same as any other attribute value. For example,

```
(make person ↑name pat ↑at lifetime)
```

establishes that **pat** is a **person** over the interval **lifetime**.

Interval attributes allow us to anchor intervals to a time/date line and give them duration. An *interval attribute specification* is a vector containing one or more attributes and their values. The **extent**, **start**, and **end** attributes specify an interval's duration and when it started and ended, respectively. We express the values in units of the granularity declaration. For example,

```
(start 10 5)
```

specifies that some interval started at ten minutes and five seconds, if we use the granularity declaration presented earlier. Intervals can also have *open pasts* and *open futures*. Assume that our friend Pat was born at ten minutes and five seconds. Additionally, we do not know how long Pat will live, so this requires an interval with an open future. The complete make command for this situation is:

```
(make person ↑name pat ↑at lifetime ((start 10 5) (end open)))
```

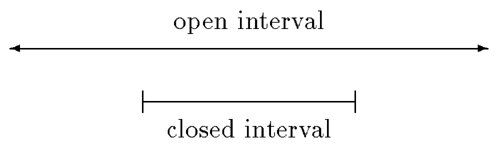


Figure 6.4: Open and closed intervals

TPS observes constraints when rules establish new relations with open intervals. For open past intervals, for example, no intervals can occur before them, since this would imply that we can determine the beginning of the open interval. TPS observes similar constraints for open future intervals. For example, for the situation pictured in Figure 6.4, the only valid relationship between the open interval and the closed interval is `contains`.

The `start-of`, `end-of`, and `extent-of` functions retrieve interval attributes. We can use these functions to bind values to variables. Examples are:

```
(bind <variable> (start-of interval))
(bind <var1> (start-of <interval-variable>))
```

Like the OPS5 `bind` command, these functions can appear only in the RHS of rules.

Finally, we need an action command to establish temporal relationships between intervals in the time map, which is the `relation` command's purpose. For example,

```
(relation during interval1 interval2)
```

establishes that `interval1` is `during interval2`. The `relation` command can appear in a rule's RHS or with the `make` commands in a TPS program.

6.5.1 Other TPS Commands

The following are additional commands that display the contents of various memories, reset TPS, execute programs, and remove patterns from working memory.

- `(tm [`<*interval1*> `[`<*interval2*>`]])`

  Prints the current time map. With no arguments, `tm` prints the entire map. With <*interval1*> as its argument, `tm` prints all relationships between <*interval1*> and all other intervals in the time map. With both arguments, `tm` prints only the relationships between <*interval1*> and <*interval2*>.

- `(pm *` | <*rule-name*>$^+$ `)`

  Prints production memory. With `*` as its argument, `pm` prints all productions; otherwise, it prints each <*rule-name*>.

- `(wm)`

  Displays working memory contents.

- `(remove` <*n*>`)`

  Removes the working memory pattern referenced by <*n*>.

- `(reset)`

  Erases working memory, production memory, and the time map.

- `(run)`

  Executes the TPS program in memory.

[3]TPS programs follow the same organization as OPS5 programs.

$$\begin{tikzcd}
\texttt{\&bus} \arrow[d] \\
\texttt{teqa 1 during} \arrow[d] \\
\texttt{teqa 2 event1} \arrow[d] \\
\texttt{teqa 3 event2} \arrow[d] \\
\texttt{tqtm during event1 event2} \arrow[d] \\
\texttt{\&p temporal-rule1}
\end{tikzcd}$$

Figure 6.5: Network representation for `temporal-rule1`.

6.6 Modifications to Rete

Rete compiles the LHS of rules, so modifications were necessary to incorporate the two time map query functions into the rule network. Since each node in the network is a test, each time map query required a node.

For the temporal query, I defined the node:

`tqtm` <*relation*> <*interval1*> <*interval2*>

which succeeds if <*relation*> exists between <*interval1*> and <*interval2*>. (`tqtm` stands for "test: query time map".) Given the rule `temporal-rule1` with a temporal query:

```
(p temporal-rule1
    ⋮
    (during event1 event2)
    ⋮
    →
    ··· )
```

we have the network representation, presented in Figure 6.5.

The second required node was "test: query temporal vector," or `tqtv`:

`tqtv` <*vector*> <*interval1*> <*interval2*>

which succeeds if any temporal relation present in <*vector*> exists in the time map between <*interval1*> and <*interval2*>. Therefore for the following rule with a temporal vector query:

```
(p temporal-rule1
    ⋮
    ([before during after] event1 event2)
    ⋮
    →
    ··· )
```

we have the network representation, presented in Figure 6.6, where `&any` succeeds if the value in the indicated register, in our case register 1, is a member of the vector.

The Rete Match Algorithm required several modifications to account for the continual updating of knowledge in the time map. Recall that the only working memory operations are addition and deletion. Whichever

$$\begin{tikzcd}
\texttt{\&bus} \arrow[d] \\
\texttt{\&any 1 (before during after)} \arrow[d] \\
\texttt{teqa 2 event1} \arrow[d] \\
\texttt{teqa 3 event2} \arrow[d] \\
\texttt{tqtv (before during after) event1 event2} \arrow[d] \\
\texttt{\&p temporal-rule2}
\end{tikzcd}$$

Figure 6.6: Network representation for `temporal-rule2`.

operation occurs, the match procedure takes the pattern, propagates it through the network, and if this activity reaches a rule node, updates the conflict set.

In the context of the time map patterns, the same must occur. When we add a temporal relation to the time map, a flurry of activity results. Because of the way in which the temporal knowledge maintenance routines work, one added temporal relationship can spawn several additions and deletions as the new relationship propagates throughout the time map. We must in turn propagate each relationship change through the rule network to ensure a consistent conflict set.

Since the constraint propagation routine runs in $O(n^2)$ space, the number of potential temporal patterns that require matching is high. Furthermore, the number of these patterns will be highest for domains that require large quantities of uncertain temporal relations. Indeed, representing and maintaining temporal knowledge is costly whatever the form or purpose. We should be thankful for the polynomial space requirement.

6.6.1 Consistency Maintenance

Consistency maintenance or *truth maintenance* (Doyle 1979) is the module of a reasoner that ensures knowledge integrity and prevents contradictory information from making its way into the knowledge-base. Ideally, when a reasoner's beliefs become invalid because of new knowledge, then the reasoner must be capable of revising any supporting evidence for those beliefs. Thus, we need some accounting method to keep track of what evidence supported which belief.

For TPS, consistency maintenance involves validating time map relations. Because of the way in which the constraint propagation algorithm computes temporal relationships, after it adds a new relation, if any arc between two intervals is empty, then an inconsistent relationship exists in the time map. If TPS's consistency maintenance procedure discovers such a situation, it issues an error message and indicates what relationship caused the inconsistency.

6.7 Examples

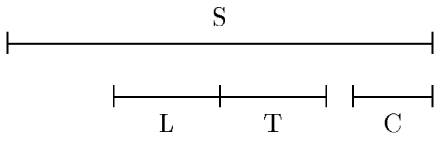
In this section, we consider two examples. For the first, which is primarily illustrative, we further develop the thunder and lightning example. The second example, which relates to an application domain, demonstrates how TPS handles a more complex problem. Before choosing a domain problem, I investigated elevator scheduling, manufacturing, and a temporal variation of the "monkey and banana." After much deliberation, I chose the domain of international terrorism.

6.7.1 Thunder and Lightning

The following example demonstrates how TPS refines and maintains temporal knowledge as new information becomes available. To demonstrate this, we return to our familiar thunder and lightning example. Assume we have the following time map:

```
S ← (during) — L — (meets) → T
S — (contains finished-by overlaps) → T
```

At some point during computation, suppose we discover that a period of calm, denoted `C`, always ends a storm and no thunder or lightning occurs during this period of calm. Given this new information, TPS refines the relationships in the time map, specifically the relationships between the storm and the thunder. After we assert that some period of calm finishes the storm and this period follows thunder, the only valid relations that could exist between the storm and thunder is the **`contains`** relation, pictured below:



Therefore, the time map representation is:

```
S ← (during) — L — (meets) → T
S — (contains) → T
```

The TPS program and sample runs for this example appear in Appendix C.

6.7.2 Terrorism

International terrorism is a phenomenon few individuals experience, yet it is often on the minds of travelers, government officials, and corporate executives. Although traditional domains of expert systems include manufacturing, medicine, and finance, there have been attempts to apply AI techniques to the ill-defined domain of terrorism (Waterman and Jenkins 1986).

In capturing this domain, Waterman and Jenkins (1986:98) make three assumptions. The first is that we can formally analyze the terrorism domain and that it follows logical rules discernable by experts. Secondly, we can focus on small, tractable portions of the domain and still have a useful system. Thirdly, a rule-based system is adequate for capturing this domain.

To build an expert system for terrorism, Waterman and Jenkins (1986:106) concentrate on three informational focal points: *events* (what happened and when), *groups* (what terrorist group is responsible), and *context* (what political, social, or economic events led to the incident). Although events are a main focal point of the system, as described, no temporal reasoning exists. The system can associate dates and times to events and make simple before/after analyses (e.g., did this event occur before six o'clock?), but it appears that no mechanisms are available to reason about the temporal organization of events.

Without such abilities, it is difficult to establish modes of operation for a group or to identify which group is potentially responsible for a given pattern of events. Evidently, these abilities are important. In one example, Waterman and Jenkins (1986:114) formulate the following rule:

```
IF:
    ⋮
    the frequency OF the campaign IS ``escalating''
    OR the time-between-attacks OF the campaign IS ``short''
    ⋮
THEN:
    ⋮
```

Presumably, the system queries the user who supplies terms like *escalating* and *short*. Yet, these terms have no semantic meaning relevant to the knowledge-base. They could in fact contradict the knowledge in the knowledge-base. Therefore, we need some mechanism to attach meaning to these terms. Furthermore, we need to be able to relate events in an efficient manner that prevents us from specifying every possible temporal relationship. TPS solves both problems.

Consider the following scenario: we require our expert terrorism system to identify potential terrorist groups, based not only on events, but also on the temporal organization of these events. Also, following the lead of Waterman and Jenkins, we develop rules to determine the amount of time between terrorist attacks. To demonstrate how we can use TPS to accomplish these functions, I devised a simple example presented in Appendix D.

The terrorism example identifies which group is responsible for a pattern of events (the mode-of-operation task) and determines if the time between terrorist events is short, moderate, or long (the time-spans task).

For the mode-of-operation task, TPS analyzes six hypothetical terrorist events and using specified interval attributes, determines how the average time between these incidents compares to elicited expert knowledge. The system responds that the time period between these events is short.

With temporal reasoning, we are able to analyze the knowledge in working memory and the time map to determine that the average time between terrorist attacks is short. Conversely, without temporal reasoning, the system must ask the user how long the period between events is. The user's response has no relevance to the knowledge in the knowledge-base and could be contradictory.

The mode-of-operation task uses TPS's temporal knowledge maintenance routines to analyze temporal patterns, or the temporal organization of events, to make a determination about which terrorist group might be responsible for an attack, given certain related events and political contexts.

Chapter 7

Conclusions

Diagnosing problems in a temporal domain can depend on the existence of one relationship between two events. Without temporal knowledge maintenance routines, a user attempting to diagnose a problem has to explicitly state pertinent temporal relations in order for the system to provide a diagnosis. Identification systems behave in much the same manner. Production rules fire based on temporal patterns that the user expresses during a consultation. We, as researchers and developers, cannot expect system users to enumerate all domain temporal relations for a problem. Nor should we expect them to specify only the key relationships, especially if such relationships are obscure.

Intuitively, we need mechanisms to infer temporal relations between events from explicitly stated ones. Furthermore, these mechanisms should function transparently within whatever host system we choose. TPS is a tool that satisfies these requirements and provides temporal reasoning functions which are not available in any existing expert system shell. Other shells provide temporal reasoning abilities, but they concentrate on reasoning temporally about past, continuous events.

TPS focuses on another aspect of temporal reasoning which is analyzing how events relate in time. Temporal sequences are important, but not all temporal phenomena reduce to chains of data points. So, the organization of events in time is equally pertinent and this aspect of temporal reasoning has been largely ignored by expert system shell designers.

As much as I want to claim TPS's superiority as a temporal reasoning expert system shell, I cannot. Researchers in atemporal reasoning admit that more than one knowledge representation is necessary to capture most complex domains, and temporal reasoning is no different. Kahn and Gorry (1979) had the right idea in providing several types of different temporal knowledge representations in Time Specialist. As demonstrated, TPS maintains and reasons about events in time with ease and efficiency. Nevertheless, temporal sequences are difficult to model because of the interval representation since there is no immediate method of determining the "next interval." Again, we have come to the give and take of representation versus computation. Certainly in a rule-based system, we can determine the next interval by constructing more complex rules. We can avoid complex rules by adding a different representation. This representation can be either separate from or integrated with the time map. If it is separate, then before/after chains would work well. On the other hand, we could use links within the time map to connect strictly successive intervals. While this addition would immediately improve TPS, there are numerous global problems in temporal reasoning that require solutions.

The tractability of computing transitive closure with the interval algebra requires further investigation. As it stands, this problem is $\mathcal{NP}$-hard, which means any complete *algorithm* runs in time that exponentially increases with the number of intervals in the time map. We can avoid facing this intractability by doing one of the following:

1. limiting ourselves to domains requiring small amounts of temporal knowledge,
2. using an incomplete, polynomial time algorithm,
3. making representational adjustments, or

4. compromising operator expressivity.

While none of these options is palatable, at present we have no alternatives. Looking at other possible methods for temporal knowledge maintenance would be a fruitful endeavor.

Bibliography

Abadi, M., and Manna, Z. (1985) Nonclausal temporal deduction. In Parikh, R., ed., *Logic of programs* (Lecture Notes in Computer Science, 193), 1–15. New York, NY: Springer.

Abadi, M., and Manna, Z. (1989) Temporal logic programming. *Journal of Symbolic Computation* 8:277–295.

Allen, J. F. (1981) An interval-based representation of temporal knowledge. *IJCAI-81*, 221–226.

Allen, J. F. (1983) Maintaining knowledge about temporal intervals. *Communications of the ACM* 26.11 (November) 832–843.

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

Ariño, N. C. (1989) Integrating temporal reasoning in a frame-based formalism. In Campbell, J., and Cuena, J., eds., *Perspectives in artificial intelligence, volume II: machine translation, NLP, databases, and computer-aided instruction*, 86–96. New York, NY: John Wiley & Sons.

Bolour, A.; Anderson, T. L.; Dekeyser, L. J.; and Wong, H. K. T. (1982) The role of time in information processing: a survey. *ACM SIGART Newsletter* 80 (April) 28–48.

Brownston, L.; Farrell, R.; Kant, E.; and Martin, N. (1985) *Programming expert systems in OPS5: an introduction to rule-based programming.* Reading, MA: Addision-Wesley.

Bruce, B. C. (1972) A model for temporal references and its application in a question answering program. *Artificial Intelligence* 3:1–25.

Buchanan, B. G., and Shortliffe, E. H., eds. (1984) *Rule-based expert systems: the MYCIN experiments of the Stanford Heuristic Programming Project.* Reading, MA: Addison-Wesley.

Chellas, B. (1980) *Modal logic: an introduction.* New York, NY: Cambridge University Press.

Chomsky, N. (1959) On certain formal properties of grammars. *Information and Control* 2:137–167.

Cooper, T. A., and Wogrin, N. (1988) *Rule-based programming with OPS5.* San Mateo, CA: Morgan Kaufmann.

Dean, T. (1986) *Temporal imagery: an approach to reasoning about time for planning and problem solving.* Ph.D. dissertation, Yale University, New Haven.

Doyle, J. (1979) A truth maintenance system. *Artificial Intelligence* 12:231–272.

Fagan, L. M.; Kunz, J. C.; Feigenbaum, E. A.; and Osborn, J. J. (1984a) Extensions to the rule-based formalism for a monitoring task. In Buchanan, B. G., and Shortliffe, E. H., eds., *Rule-based expert systems: the MYCIN experiments of the Stanford Heuristic Programming Project*, 302–313. Reading, MA: Addison-Wesley.

Fagan, L.; Shortliffe, E.; and Buchanan, B. (1984b) Computer-based medical decision making: from MYCIN to VM. In Clancey, W., and Shortliffe, E., eds., *Readings in medical artificial intelligence: the first decade*, 241–255. Reading, MA: Addison-Wesley.

Firebaugh, M. W. (1988) *Artificial intelligence: a knowledge-based approach.* Boston, MA: PWS-Kent Publishing.

Fleck, M. M. (1989) *Boundaries and topological algorithms* (MIT AI Lab Technical Report 1065). Ph.D. dissertation, Massachusetts Institute of Technology, Cambridge.

Forgy, C. L. (1981) *VPS2 — interpreter for OPS5.* Pittsburg, PA.

Forgy, C. L. (1982) Rete: a fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence* 19:17–37.

Forgy, C. L., and McDermott, J. (1977) OPS, a domain-independant production system language. *IJCAI-77*, 933–939.

Galton, A. (1987) Temporal logic and computer science: an overview. In Galton, A., ed., *Temporal logics and their applications*, 1–52. San Diego, CA: Academic Press.

Galton, A. (1990) A critical examination of Allen's theory of action and time. *Artificial Intelligence* 42:159–188.

Genesereth, M. and Nilson, N. (1988) *Logical foundations of artificial intelligence.* Palo Alto, CA: Morgan Kaufmann.

Hale, R. (1987) Temporal logic programming. In Galton, A., ed., *Temporal logics and their applications*, 91–119. San Diego, CA: Academic Press.

Hanks, S., and McDermott, D. (1985) *Temporal reasoning and default logics* (Technical Report 430). Department of Computer Science, Yale University.

Hayes, P. J. (1979) The naïve physics manifesto. In Michie, D., ed., *Expert systems in the micro-electronic age*, 242–270. Edinburg: Edinburg University Press. Also in Boden, M. A., ed., *The philosophy of artificial intelligence*, 171–205. Oxford: Oxford University Press, 1990.

Hayes, P. J. (1984a) Naïve physics I: ontology for liquids. In Hobbs, J. R., and Moore, R. C., eds., *Formal theories of the commonsense world*, 71–107. Norwood, NJ: Ablex Publishing.

Hayes, P. J. (1984b) The second naïve physics manifesto. In Hobbs, J. R., and Moore, R. C., eds., *Formal theories of the commonsense world*, 1–36. Norwood, NJ: Ablex Publishing.

Hopcroft, J. E., and Ullman, J. D. (1979) *Introduction to automata theory, languages, and computation.* Reading, MA: Addison-Wesley.

Hornstein, N. (1977) Towards a theory of tense. *Linguistic Inquiry* 8.3:521–557.

Kabakçloğlu, A. M. (1992) Temporal production systems. *IEEE Southeastcon '92*, 697–698.

Kahn, K., and Gorry, G. A. (1977) Mechanizing temporal knowledge. *Artificial Intelligence* 9:87–108.

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

Kumar, V. (1992) Algorithms for constraint satisfaction problems: a survey. *AI Magazine* 13.1:32–44.

Lee, R. M.; Coelho, H.; and Cotta, J. C. (1985) Temporal inferencing on administrative databases. *Information Systems* 10.2:197–206.

Lopez, F.; Savely, R. T.; Culbert, C.; and Riley, G. (1987) CLIPS: a NASA developed expert system tool. *NASA Tech Briefs* 11.10 (November/December) 12–14.

Manna, Z. (1980) Logics of programs. In Lavington, S., ed., *Information Processing 80*, 41–51. New York, NY: North-Holland.

McCarthy, J. (1980) Circumscription — a form of non-monotonic reasoning. *Artificial Intelligence* 13:27–39.

McCarthy, J., and Hayes, P. J. (1969) Some philosophical problems from the standpoint of artificial intelligence. In Meltzer, B.; Michie, D.; and Swann, M., eds., *Machine intelligence 4*, 463–502. New York, NY: American Elsevier. Also in *Readings in artificial intelligence.* 466–472. Palo Alto, CA: Tioga, 1981.

McDermott, D. (1982) A temporal logic for reasoning about processes and plans. *Cognitive Science* 6:101–155.

McDermott, D., and Doyle, J. (1980) Non-monotonic logic I. *Artificial Intelligence* 13:41–72.

Miller, R.; Pople, H.; and Myers, J. (1984) INTERNIST-1, an experimental computer-based diagnostic consultant for general internal medicine. In Clancey, W., and Shortliffe, E., eds., *Readings in medical artificial intelligence: the first decade*, 190–209. Reading, MA: Addison-Wesley.

Moore, R. L.; Rosenhof, H.; and Stanley, G. (1989) Process control using the G2 real-time expert system. *Conference record of the IEEE Industry Application Society Annual Meeting.* 1452–1456.

Newell, A., and Simon, H. (1972) *Human problem solving.* Englewood Cliffs, NJ: Prentice-Hall.

Nute, D. (1991) Historical necessity and conditionals. *Noûs* 25:161–175.

Orlowska, E. (1982) Representation of temporal information. *International Journal of Computer and Information Sciences* 11.6:397–408.

Perkins, W. A., and Austin A. (1990) Adding temporal reasoning to expert-system-building environments. *IEEE Expert* 5.1 (February) 23–30.

Pople, H. E. (1977) The formation of composite hypotheses in diganostic problem solving: an excercise in synthetic reasoning. *IJCAI-77*, 1030–1037.

Post, E. L. (1943) Formal reductions of the general combinatorial decision problem. *American Journal of Mathematics* 65:197–268.

Reichgelt, H. (1987) Semantics for reified temporal logic. In Hallam, J., and Mellish C., eds., *Advances in artificial intelligence*, 49–61. New York, NY: John Wiley & Sons.

Reiter, R. (1980) A logic for default reasoning. *Artificial Intelligence* 13:81–132.

Rescher, N., and Urquhart, A. (1971) *Temporal logic.* New York, NY: Springer.

Shamsudin, A. Z., and Dillon, T. S. (1991) *NetManager: a real-time expert system for network traffic management* (Technical Report 15/91). Department of Computer Science and Computer Engineering, La Trobe University, Melbourne, Australia.

Sherman, P., and Martin, J. (1990) *An OPS5 primer: introduction to rule-based expert systems.* Englewood Cliffs, NJ: Prentice-Hall.

Shoham, Y. (1988a) Chronological ignorance: experiments in nonmonotonic temporal reasoning. *Artificial Intelligence* 36:279–331.

Shoham, Y. (1988b) *Reasoning about change.* Cambridge, MA: The MIT Press.

Shoham, Y. (1988c) *Time for action: on the relation between time, knowledge, and action* (Report No. STAN-CS-88-1236). Department of Computer Science, Stanford University.

Shoham, Y., and Goyal, N. (1988) Temporal reasoning in artificial intelligence. In Shrobe, H., ed., *Exploring artificial intelligence: survey talks from the national conferences on artificial intelligence*, 419–438. San Mateo, CA: Morgan Kaufmann Publishers.

Shoham, Y., and McDermott, D. (1988) Problems in formal temporal reasoning. *Artificial Intelligence* 36:49–61.

Thomason, R. H., and Gupta, A. (1980) A theory of conditionals in the context of branching time. In Harper, W. L.; Stalnaker, R.; and Pearce, G., eds., *Ifs: conditionals, belief, decision, chance, and time*, 299–322. Boston, MA: D. Reidel.

Valdéz-Pérez, R. E. (1987) The satisfiability of temporal constraint networks. *AAAI-87*, 256–260.

van Benthem, J. F. A. K. (1990) *The logic of time: a model-theoretic investigation into the varieties of temporal ontology and temporal discourse* (Synthese Library, 156). 2nd ed. Boston, MA: D. Reidel.

van Melle, W.; Shortliffe, E. H.; and Buchanan, B. G. (1984) EMYCIN: a knowledge-engineer's tool for constructing rule-based expert systems. In Buchanan, B. G., and Shortliffe, E. H., eds., *Rule-based expert systems: the MYCIN experiments of the Stanford Heuristic Programming Project*, 397–423. Reading, MA: Addison-Wesley.

Vilain, M. (1982) A system for reasoning about time. *AAAI-82*, 197–201.

Vilain, M., and Kautz, H. (1986) Constraint propagation algorithms for temporal reasoning. *AAAI-86*, 377–382.

Waterman, D., and Jenkins, B. M. (1986) Developing expert systems to combat international terrorism. In Klahr, P., and Waterman, D., eds., *Expert systems: techniques, tools, and applications,* 95–134. Reading, MA: Addison-Wesley.

Wolper, P. (1983) Temporal logic can be more expressive. *Information and Control* 56:72–99. Also in *22nd Annual IEEE Symposium on Foundations of Computer Science*, 340–348, 1981.

Zhu, M.; Loh, N. K.; and Siy, P. (1987) Towards the minimum set of primitive relations in temporal logic. *Information Processing Letters* 26:121–126.

Appendix A

Additional Details for Completeness

Case 4.2 of the completeness proof for PTL, presented in Section 2.1.1, requires that if $\vdash \phi \vee \psi$ then $\vdash \phi \,\mathcal{U}\, \psi$. The proof presented in Figure A.1 demonstrates this supposition is correct.

(1)	$\vdash \phi \vee \psi$	P
(2)	$\vdash \Box(\phi \vee \psi)$	1 I3
(3)	$\vdash \Box(\neg\phi \supset \psi)$	2 I1
(4)	$\vdash \Box(\neg\phi \supset \psi) \supset (\Box\neg\phi \supset \Box\psi)$	A2
(5)	$\vdash \Box\neg\phi \supset \Box\psi$	3,4 I2
(6)	$\vdash \Box(\Box\neg\phi \supset \Box\psi)$	5 I3
(7)	$\vdash \Box(\Box\neg\phi \supset \Box\psi) \supset [(\Box\neg\phi \supset \Box\psi)$ $\wedge \bigcirc(\Box\neg\phi \supset \Box\psi) \wedge \bigcirc\Box(\Box\neg\phi \supset \Box\psi)]$	A5
(8)	$\vdash (\Box\neg\phi \supset \Box\psi) \wedge \bigcirc(\Box\neg\phi \supset \Box\psi) \wedge \bigcirc\Box(\Box\neg\phi \supset \Box\psi)$	6,7 I2
(9)	$\vdash \bigcirc(\Box\neg\phi \supset \Box\psi)$	8 I1,I2
(10)	$\vdash \bigcirc(\Box\neg\phi \supset \Box\psi) \supset (\bigcirc\Box\neg\phi \supset \bigcirc\Box\psi)$	A4
(11)	$\vdash \bigcirc\Box\neg\phi \supset \bigcirc\Box\psi$	9,10 I2
(12)	$\vdash \bigcirc\Box\phi \vee \bigcirc\Box\psi$	11 I1
(13)	$\vdash (\phi \vee \psi) \wedge (\bigcirc\Box\phi \vee \bigcirc\Box\psi)$	1,12 I1[†]
(14)	$\vdash [(\phi \vee \bigcirc\Box\phi) \wedge (\phi \vee \bigcirc\Box\psi)] \wedge [(\psi \vee \bigcirc\Box\phi) \wedge (\psi \vee \bigcirc\Box\psi)]$	13 I1
(15)	$\vdash \psi \vee \bigcirc\Box\phi$	14 I1,I2
(16)	$\vdash \psi \vee \phi$	1 I1
(17)	$\vdash \Box\phi \supset \phi \,\mathcal{U}\, \psi$	A7
(18)	$\vdash \Box(\Box\phi \supset \phi \,\mathcal{U}\, \psi)$	17 I3
(19)	$\vdash \Box(\Box\phi \supset \phi \,\mathcal{U}\, \psi) \supset [(\Box\phi \supset \phi \,\mathcal{U}\, \psi)$ $\wedge \bigcirc(\Box\phi \supset \phi \,\mathcal{U}\, \psi) \wedge \bigcirc\Box(\Box\phi \supset \phi \,\mathcal{U}\, \psi)]$	A5
(20)	$\vdash (\Box\phi \supset \phi \,\mathcal{U}\, \psi) \wedge \bigcirc(\Box\phi \supset \phi \,\mathcal{U}\, \psi) \wedge \bigcirc\Box(\Box\phi \supset \phi \,\mathcal{U}\, \psi)$	18,19 I2
(21)	$\vdash \bigcirc(\Box\phi \supset \phi \,\mathcal{U}\, \psi)$	20 I1
(22)	$\vdash \bigcirc(\Box\phi \supset \phi \,\mathcal{U}\, \psi) \supset \bigcirc\Box\phi \supset \bigcirc(\phi \,\mathcal{U}\, \psi)$	A4
(23)	$\vdash \bigcirc\Box\phi \supset \bigcirc(\phi \,\mathcal{U}\, \psi)$	21,22 I2
(24)	$\vdash \neg\psi \supset \bigcirc\Box\phi$	15 I1[†]
(25)	$\vdash \neg\psi \supset \bigcirc(\phi \,\mathcal{U}\, \psi)$	24,23 I1,I2
(26)	$\vdash \psi \vee \bigcirc(\phi \,\mathcal{U}\, \psi)$	25 I1
(27)	$\vdash (\psi \vee \phi) \wedge (\psi \vee \bigcirc(\phi \,\mathcal{U}\, \psi))$	16,26 I1[†]
(28)	$\vdash \psi \vee (\phi \wedge \bigcirc(\phi \,\mathcal{U}\, \psi))$	27 I1
(29)	$\vdash \psi \vee (\phi \wedge \bigcirc(\phi \,\mathcal{U}\, \psi)) \supset \phi \,\mathcal{U}\, \psi$	A7
(30)	$\vdash \phi \,\mathcal{U}\, \psi$	28,29 I2

[†]For the sake of brevity, I am using a derived transformation rule.

Figure A.1: Proof required for completeness.

Appendix B

Running TPS

During the development of TPS, I freely moved between three Lisp environments. The following instructions should be enough to allow anyone to execute a simple TPS program. I strongly recommend the use of the OPS5 code found in Appendix F. During development, I discovered two different versions of OPS5 code. The two versions may have been equivalent in function, but they were not equivalent in form. The OPS5 code presented in this thesis does have some modernizations. For instance, I replaced several hand-written routines with native Lisp functions, intersection was one of them. My version is definitely different in form, and I dare not comment on its function.

B.0.3 Macintosh Common Lisp

The main development environment for TPS was Macintosh Common Lisp, version 2.0b1. To run a TPS program, follow these instructions:

1. Double-click on the MCL 2.0b1 icon.
2. Open the files `ops5.cl` and `tps.cl`.
3. Evaluate the buffer containing `ops5.cl`.
4. Evaluate the buffer containing `tps.cl`.
5. From the Eval menu, select the Load command and select the desired TPS source file.
6. From the Lisp Listener window, type `(run)`.

B.0.4 Austin Kyoto Common Lisp

These instructions are for Austin Kyoto Common Lisp version 1.492.

1. Make the directory containing all source files the present working directory.
2. At the Unix prompt, type `kcl`.
3. Once in KCL, type `(load "tps.cl")`. This loads both OPS5 and TPS.
4. After both files load, type `(setf *package* (find-package "TPS"))`. The Lisp prompt should change to: `TPS>`.
5. Now type `(load "`*<filename>*`")`, where *<filename>* is the desired TPS source file.
6. Type `(run)`.

B.0.5 Allegro Common Lisp

These instructions are for Allegro Common Lisp version 4.0.1.

1. Make the directory containing all source files the present working directory.
2. At the Unix prompt, type `cl`.
3. Once in Allegro CL, type `(load "tps.cl")`. This loads both OPS5 and TPS.
4. After both files load, type `(setf *package* (find-package "TPS"))`.
5. Now type `(load "`<*filename*>`")`, where <*filename*> is the desired TPS source file.
6. Type `(run)`.

Appendix C

Thunder and Lightning Example

C.1 TPS Program

```
;;;;
;;;; Thunder and Lightning
;;;;
;;;; A simple TPS program to demonstrate how TPS
;;;; updates temporal relationships as new ones
;;;; become known.
;;;;

(literalize event at)

(p during-relationship
   (during thunder storm)
   -->
   (write (crlf) |the thunder is during the storm| (crlf)))

(p finishes-relationship
   (finishes thunder storm)
   -->
   (write (crlf) |the thunder finishes the storm| (crlf)))

(p overlaps-relationship
   (overlaps storm thunder)
   -->
   (write (crlf) |the storm overlaps the thunder| (crlf)))
```

```
(make event ^at storm)
(make event ^at lightning)
(make event ^at thunder)
(make event ^at calm)

(relation meets lightning thunder)
(relation during lightning storm)
```

C.2 Sample Run

```
;;;
;;; Evaluate the buffer
;;;
? ***
nil
? (wm)                  ;; look at working memory

1:  (event ^at storm)
2:  (event ^at lightning)
3:  (event ^at thunder)
4:  (event ^at calm)
nil
? (tm)                  ;; look at time map
5 : lightning -- (meets) --> thunder
6 : lightning -- (during) --> storm
5 : thunder -- (met-by) --> lightning
6 : thunder -- (finishes overlapped-by during) --> storm
6 : storm -- (contains) --> lightning
6 : storm -- (finished-by overlaps contains) --> thunder
nil
;;;
;;; Look at conflict set; Notice
;;; which rules match.
;;;
? (cs)

during-relationship
overlaps-relationship
```

```
finishes-relationship
(finishes-relationship dominates)
;;;
;;; Assert that the calm finishes the storm.
;;;
? (relation finishes calm storm)
nil
;;;
;;; Assert that the calm always follows thunder.
;;;
? (relation after calm thunder)
nil
;;;
;;; Now when we look at the time map, notice how
;;; the relations have been constrained based on
;;; the new information, especially the
;;; relationship between the storm and thunder.
;;;
? (tm)
8 : lightning -- (before) --> calm
6 : lightning -- (during) --> storm
5 : lightning -- (meets) --> thunder
8 : calm -- (after) --> lightning
7 : calm -- (finishes) --> storm
8 : calm -- (after) --> thunder
6 : storm -- (contains) --> lightning
7 : storm -- (finished-by) --> calm
8 : storm -- (contains) --> thunder
5 : thunder -- (met-by) --> lightning
8 : thunder -- (before) --> calm
8 : thunder -- (during) --> storm
nil
;;;
;;; The conflict set has now been updated to
;;; reflect the changes in the time map.
;;; Compare with the conflict set from
;;; above.
;;;
? (cs)
```

```
during-relationship
(during-relationship dominates)
;;;
;;; Execute the production
;;;
? (run)

1. during-relationship 8
the thunder is during the storm

end -- no production true
  3 productions (15 // 15 nodes)
  1 firings (8 rhs actions)
  4 mean working memory size (4 maximum)
  4 mean time map size (4 maximum)
  1 mean conflict set size (1 maximum)
  0 mean token memory size (0 maximum)
nil
?
```

Appendix D

Terrorism Example

D.1 TPS Program

```
;;;;
;;;; Terrorism Example
;;;;


;;;
;;; Initializations
;;;

(granularity days)

(literalize event type target flag at)
(literalize context type what by at)
(literalize time-between-attacks time-span)
(literalize distance value)
(literalize count value)
(literalize average value)


;;;
;;; Determining time span productions
;;;

(p start-time-spans
   { (start) <s> }
   (task time-spans)
   -->
   (remove <s>)
   (make distance 0)
   (make count 0))

(p compute-distances
   (task time-spans)
   { (event ^flag n ^at <int1>) <n> }
   (event ^at <int2>)
   ([before meets] <int1> <int2>)
   { (distance ^value <dist>) <d> }
   { (count ^value <cnt>) <c> }
   -->
   (modify <n> ^flag t)
   (bind <first> (end-of <int1>))
   (bind <second> (start-of <int2>))
   (bind <distance> (compute <second> - <first>))
   (bind <newdist> (compute <distance> + <dist>))
   (bind <newcnt> (compute <cnt> + 1))
   (modify <c> ^value <newcnt>)
   (modify <d> ^value <newdist>))

(p start-average-computation
   (task time-spans)
   - (event ^flag nil)
   -->
   (make compute-average))

(p calculate-average
   (task time-spans)
   { (compute-average) <ca> }
   { (distance ^value <dist>) <d> }
   { (count ^value <cnt>) <c> }
   -->
   (remove <ca>)
   (bind <avg> (compute <dist> // <cnt>))
   (make average ^value <avg>))

(p determine-time-span1
   (task time-spans)
   (average ^value < 5)
   -->
   (make time-between-attacks ^time-span short)
   (write (crlf) |The time between attacks is short| (crlf)))

(p determine-time-span2
   (task time-spans)
   (average ^value > 5)
   (average ^value < 10)
   -->
   (make time-between-attacks ^time-span moderate)
   (write (crlf) |The time between attacks is of moderate length| (crlf)))

(p determine-time-span3
   (task time-spans)
   (average ^value > 15)
   -->
   (make time-between-attacks ^time-span long)
   (write (crlf) |The time between attacks is long| (crlf)))

;;;
;;; Mode of operation productions
;;;

(p start-mode
   { (start) <s> }
   (task mode-of-operation)
   -->
   (remove <s>)
   (make event ^type hijack-claim ^flag t ^at hijack-claim)
   (relation starts hijack-claim hijacking)
   (relation contains all-times hijack-claim))

;;;
;;; Mode of operation domain rules
;;;

(p abunidal
   (task mode-of-operation)
   (context ^by israel ^at <interval1>)
   (context ^type political ^what support-isreal)
   (event ^type <event> ^at <interval2>)
   (contains <interval1> <interval2>)
   -->
   (write (crlf) |Abu Nidal could be responsible for| <event> (crlf)))

(p ira
   (task mode-of-operation)
   (context ^what retaliation ^by britain ^at <interval1>)
   (event ^type bombing ^at <interval2>)
   ([after overlaps finishes met-by] <interval1> <interval2>)
   -->
   (write (crlf) |IRA could be responsible for bombing| (crlf)))

;;;
;;; Make and Relation commands
;;;

(make event ^type bombing ^target train-station
      ^flag n ^at bombing ((start 5) (end 5)))
(make event ^type kidnapping ^target ambassador
      ^flag n ^at kidnapping ((start 7) (end 10)))
(make event ^type hijacking ^target airliner
      ^flag n ^at hijacking ((start 12) (end 15)))
(make event ^type warning ^flag n ^at warning ((start 16) (end 16)))
(make event ^type bomb-claim ^flag n
      ^at bomb-claim ((start 16) (end 16)))
(make event ^type shooting ^target executive
      ^flag n ^at shooting ((start 20) (end 20)))

(make context ^type political ^what no-prisoner-relase
      ^by israel ^at all-times ((start open) (end open)))
(make context ^type political ^what retaliation
      ^by britain ^at retaliation)

(relation before bombing kidnapping)
(relation meets kidnapping hijacking)
```

```
(relation before hijacking warning)
(relation before warning bomb-claim)
(relation meets bomb-claim shooting)
(relation overlaps retaliation bombing)

(make start)
```

D.2 Sample Runs

```
;;;
;;; Evaluate buffer
;;;
? **********
nil
? (make task time-spans)       ;; set up the correct task
nil
? (run)                        ;; execute the example

1. start-time-spans 15 16
2. compute-distances 16 5 6 13 18 19
3. compute-distances 16 3 21 13 25 23
4. compute-distances 16 2 27 10 31 29
5. compute-distances 16 1 33 9 37 35
6. compute-distances 16 4 21 12 43 41
7. start-average-computation 16
8. calculate-average 16 50 49 47
9. determine-time-span1 16 52
;;;
;;; Computation results
;;;
the time between attacks is short

end -- no production true
 10 productions (93 // 133 nodes)
  9 firings (53 rhs actions)
 11 mean working memory size (13 maximum)
  6 mean time map size (6 maximum)
  7 mean conflict set size (17 maximum)
100 mean token memory size (141 maximum)
```

```
nil
? (wm)                        ;; check working memory contents

6:  (event ^type shooting ^target executive ^flag n ^at shooting)
21:  (event ^type bomb-claim ^flag t ^at bomb-claim)
27:  (event ^type hijacking ^target airliner ^flag t ^at hijacking)
33:  (event ^type kidnapping ^target ambassador ^flag t ^at kidnapping)
39:  (event ^type bombing ^target train-station ^flag t ^at bombing)
45:  (event ^type warning ^flag t ^at warning)
7:  (context ^type political ^what no-prisoner-relase
                             ^by israel ^at all-times)
8:  (context ^type political ^what retaliation
                             ^by britain ^at retaliation)
16:  (task time-spans)
49:  (distance ^value 9)
47:  (count ^value 5)
52:  (average ^value 1)
53:  (time-between-attacks ^time-span short)
nil
;;;
;;; Set up for next task
;;;
? (remove 16)
nil
? (make task mode-of-operation)
nil
? (make start)
nil
? (run)                        ;; execute next task

10. start-mode 56 55
11. ira 55 8 39 14
;;;
;;; Computation results
;;;
ira could be responsible for bombing

end -- no production true
 10 productions (93 // 133 nodes)
 11 firings (60 rhs actions)
 12 mean working memory size (14 maximum)
  6 mean time map size (8 maximum)
  6 mean conflict set size (17 maximum)
 96 mean token memory size (141 maximum)
nil
? (wm)                          ;; check working memory contents

6:  (event ^type shooting ^target executive ^flag n ^at shooting)
21:  (event ^type bomb-claim ^flag t ^at bomb-claim)
27:  (event ^type hijacking ^target airliner ^flag t ^at hijacking)
33:  (event ^type kidnapping ^target ambassador ^flag t ^at kidnapping)
39:  (event ^type bombing ^target train-station ^flag t ^at bombing)
45:  (event ^type warning ^flag t ^at warning)
58:  (event ^type hijack-claim ^flag t ^at hijack-claim)
7:  (context ^type political ^what no-prisoner-relase
                             ^by israel ^at all-times)
8:  (context ^type political ^what retaliation
                             ^by britain ^at retaliation)
55:  (task mode-of-operation)
49:  (distance ^value 9)
47:  (count ^value 5)
52:  (average ^value 1)
53:  (time-between-attacks ^time-span short)
nil
;;;
;;; Check time map contents.  Notice
;;; how no interval bounds the
;;; all-times interval in the future
;;; or past, which is a result of
;;; the consistency and truth
;;; maintenance routines.
;;;
? (tm)
13 : bomb-claim -- (meets) --> shooting
14 : bomb-claim -- (after) --> retaliation
12 : bomb-claim -- (after) --> bombing
59 : bomb-claim -- (after) --> hijack-claim
12 : bomb-claim -- (after) --> warning
```

```
13 : bomb-claim -- (after overlapped-by met-by) --> hijacking
60 : bomb-claim -- (during) --> all-times
12 : bomb-claim -- (after) --> kidnapping
13 : shooting -- (met-by) --> bomb-claim
14 : shooting -- (after) --> retaliation
13 : shooting -- (after) --> bombing
59 : shooting -- (after) --> hijack-claim
13 : shooting -- (after) --> warning
13 : shooting -- (after) --> hijacking
60 : shooting -- (during) --> all-times
13 : shooting -- (after) --> kidnapping
14 : retaliation -- (before) --> bomb-claim
14 : retaliation -- (before) --> shooting
14 : retaliation -- (overlaps) --> bombing
59 : retaliation -- (before) --> hijack-claim
14 : retaliation -- (before) --> warning
14 : retaliation -- (before) --> hijacking
60 : retaliation -- (during) --> all-times
59 : retaliation -- (before overlaps meets) --> kidnapping
12 : bombing -- (before) --> bomb-claim
13 : bombing -- (before) --> shooting
14 : bombing -- (overlapped-by) --> retaliation
59 : bombing -- (before) --> hijack-claim
11 : bombing -- (before) --> warning
10 : bombing -- (before) --> hijacking
60 : bombing -- (during) --> all-times
9 : bombing -- (before) --> kidnapping
59 : hijack-claim -- (before) --> bomb-claim
59 : hijack-claim -- (before) --> shooting
59 : hijack-claim -- (after) --> retaliation
59 : hijack-claim -- (after) --> bombing
59 : hijack-claim -- (contains) --> warning
59 : hijack-claim -- (starts) --> hijacking
60 : hijack-claim -- (during) --> all-times
59 : hijack-claim -- (after overlapped-by met-by) --> kidnapping
12 : warning -- (before) --> bomb-claim
13 : warning -- (before) --> shooting
14 : warning -- (after) --> retaliation
11 : warning -- (after) --> bombing
```

```
59 : warning -- (during) --> hijack-claim
11 : warning -- (after) --> hijacking
60 : warning -- (during) --> all-times
11 : warning -- (after) --> kidnapping
13 : hijacking -- (before overlaps meets) --> bomb-claim
13 : hijacking -- (before) --> shooting
14 : hijacking -- (after) --> retaliation
10 : hijacking -- (after) --> bombing
59 : hijacking -- (started-by) --> hijack-claim
11 : hijacking -- (before) --> warning
60 : hijacking -- (during) --> all-times
10 : hijacking -- (met-by) --> kidnapping
60 : all-times -- (contains) --> bomb-claim
60 : all-times -- (contains) --> shooting
60 : all-times -- (contains) --> retaliation
60 : all-times -- (contains) --> bombing
60 : all-times -- (contains) --> hijack-claim
60 : all-times -- (contains) --> warning
60 : all-times -- (contains) --> hijacking
60 : all-times -- (contains) --> kidnapping
12 : kidnapping -- (before) --> bomb-claim
13 : kidnapping -- (before) --> shooting
59 : kidnapping -- (after overlapped-by met-by) --> retaliation
9 : kidnapping -- (after) --> bombing
59 : kidnapping -- (before overlaps meets) --> hijack-claim
11 : kidnapping -- (before) --> warning
10 : kidnapping -- (meets) --> hijacking
60 : kidnapping -- (during) --> all-times
nil
?
```

Appendix E

TPS Code

```
;;;;
;;;; Temporal Production System (TPS)
;;;;
;;;; Marcus A. Maloof
;;;; Artificial Intelligence Programs
;;;; University of Georgia
;;;; Athens, Georgia 30602
;;;; Spring 1992
;;;;
;;;; TPS completed as partial requirements
;;;; for Master of Science degree.
;;;;
;;;; TPS is built on OPS5 and provides an
;;;; interval-based temporal semantics
;;;; within a production system
;;;; environment.
;;;;
;;;; Definitions TPS functions and
;;;; Redefinitions of OPS5 functions
;;;;
```

Appendix F

OPS5 Code

```
;;;;
;;;; VPS2 – Interpreter for OPS5
;;;;
;;;; Copyright (C) 1979, 1980, 1981
;;;; Charles L. Forgy,  Pittsburgh, Pennsylvania
;;;;
;;;;
;;;; Users of this interpreter are requested to contact
;;;;
;;;; Charles Forgy
;;;; Computer Science Department
;;;; Carnegie-Mellon University
;;;; Pittsburgh, PA  15213
;;;; Forgy@CMUA
;;;;
;;;; so that they can be added to the mailing list for OPS5.  The mailing list
;;;; is needed when new versions of the interpreter or manual are released.
;;;;
;;;;  REPORT BUGS IN THIS VERSION TO:
;;;;   George Wood
;;;;   Computer Science Department
;;;;   Carnegie-Mellon University
;;;;   Pittsburgh, PA  15213
;;;;    arpanet: George.Wood@CMU-CS-A
;;;;
;;;; Major cleanup and modification made by:
;;;;    Marcus A. Maloof
;;;;    Artificial Intelligence Programs
;;;;    111 Graduate Studies Research Center
;;;;    The University of Georgia
;;;;    Athens, GA  30602
;;;;    mmaloof@aisun2.ai.uga.edu
;;;;    Fall 91 - Spring 92
;;;;
;;;; This version intended for use with Temporal Production System (TPS).
;;;;
```