University of Dublin

TRINITY COLLEGE

Finite-State Methods for Event Structures and Intervals

Louise Doran

B.A. (Mod.) Computer Science, Linguistics and a Language

Final Year Project, April 2011

Supervisor: Dr. Tim Fernando

School of Computer Science and Statistics

O’Reilly Institute, Trinity College, Dublin 2, Ireland
Declaration

I hereby declare that this thesis is entirely my own work and that it has not been submitted as an exercise for a degree at any other university.

_______________________________ April 11, 2011
Louise Doran
Permission to Lend

I agree that the Library and other agents of the College may lend or copy this thesis upon request.

__________________________  April 11, 2011
Louise Doran
Acknowledgements

I would like to thank my supervisor Dr. Tim Fernando for his help, advice and time during the meetings throughout the year. Without these, this project would not have been possible.

I am grateful to Dr. Carl Vogel for his advice, inspiration and guidance over the last four years.

I would like to thank my parents, brother and Stephen for their continuous support and love during my undergraduate studies.

Finally I thank my classmates for being such great and supportive friends throughout this course.
Contents

1 Introduction .......................... 1
  1.1 Background .......................... 2
  1.2 Motivation and Objective ........ 3

2 Finite State Temporality .......... 4
  2.1 Snapshots .......................... 4
  2.2 Finite State Automata .......... 8
  2.3 Event Structures .............. 10
  2.4 Allen interval relations .... 13

3 Implementation ..................... 14
  3.1 Superposition .................... 15
  3.2 Subsumption and Constraints ... 18
  3.3 Constrain ......................... 20
  3.4 Restriction ....................... 21
  3.5 Composition ..................... 22
  3.6 Block Compression ............ 24
  3.7 Unpad .................. 26
  3.8 Pi and further superposition .. 28
     3.8.1 Allen relations revisited 31
  3.9 Circumscribe .................... 32
  3.10 Output to xfst and dot .... 33
  3.11 Problems encountered .. 34

4 Analysis ............................ 37
  4.1 Future Work and improvements .. 37
  4.2 Conclusion ....................... 38
Appendices

A Code Listings

A.1 Block Compression ................................................. 41
A.2 Restriction .......................................................... 41
A.3 Superposition ...................................................... 42
A.4 Subsume ............................................................. 42
A.5 Constrain ........................................................... 43
A.6 Unpad ............................................................... 44
A.7 Pisuperpose ......................................................... 44
A.8 Dead Transitions and Remove Epsilon ......................... 45
A.9 Prolog to xfst ..................................................... 46
A.10 Prolog to dot ...................................................... 46
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Example of a simple FSM</td>
<td>9</td>
</tr>
<tr>
<td>2.2</td>
<td>Thirteen Allen Relations</td>
<td>12</td>
</tr>
<tr>
<td>3.1</td>
<td>Example of superposition between FSMs</td>
<td>18</td>
</tr>
<tr>
<td>3.2</td>
<td>Example of composition of FSTs</td>
<td>23</td>
</tr>
<tr>
<td>3.3</td>
<td>An FST for block compression</td>
<td>26</td>
</tr>
<tr>
<td>3.4</td>
<td>An FST for unpadding</td>
<td>27</td>
</tr>
<tr>
<td>3.5</td>
<td>Example of $\pi_1$ on an FSM</td>
<td>29</td>
</tr>
<tr>
<td>3.6</td>
<td>Example of a composed FSM without removal of useless transitions</td>
<td>35</td>
</tr>
</tbody>
</table>
Abstract

To allow the analysis of natural language semantics by machines, it is necessary to form a system for interpretation. This project focuses on events and temporality as they exist in natural language and describes the finite-state approach to the interpretation of interval temporal propositions and the method of implementation of a system where this approach is applied.

Here, we look to reduce time to something more basic, so we analyse events in terms of regular languages, each string is regarded as a temporal sequence of observations in chronological order. As each string is formed from a regular language, it may be recognised by a finite-state automaton. The observations are formulated as “snapshots”, which are then used as symbols in the alphabet for an FSA.

By implementing regular relations on the strings of event symbols we can reason about time and events. An implementation of these relations was created in the Prolog programming language and an output is created for the dot visualisation software and the Xerox finite-state toolkit.
Chapter 1

Introduction

Understanding the semantics of natural language is a difficult task faced in the field of artificial intelligence, it is especially difficult when we aim to understand discourse concerning time, change and actions and their effects. The investigation into the formation of a model capable of reasoning is a task that spans several disciplines. Across philosophy and mathematics to neuroscience to psychology, there are a large number of theories describing how humans mentally analyse discourse and how best to simulate our mental abilities.

As we look to find the best approach to create a working system for reasoning about events, it is necessary to be able to analyse events and their temporal relations with ease, using a system that is expressive and flexible. Thus we are led to the analysis of events as regular languages, using finite-state automata to investigate them.

In order to create a system capable of reasoning about change we require a notion of time to order events. We use the Russell-Wiener-Kamp event structure as well as finite-state temporality based on interval temporal logics. The finite-state approach consists of using strings as a representation for event structures and applying finite-state automata on them in order to extract temporal information.

The aim of this project is to present the theory given for finite-state temporality in [Fernando, 2003] [Fernando, 2007] [Fernando, 2011] and more, and discuss how certain operations on event structures were implemented in Pro-
log as runs of finite-state automata. The working implementation shows the tractability of the operations for this representation of events.

A finite-state approach to reasoning about events allows greater flexibility and efficiency in that it can encode infinite structures with ease. In addition, this approach yields high expressiveness and low computational complexity. The system for representing time and the temporal relations between events is tractable.

1.1 Background

Several systems for temporal reasoning and information extraction from events in natural language have been proposed including TimeML [Pratt-Hartmann, 2007], which looks at representing events and their temporal references that appear in natural languages text in an markup language format, Cooper’s notion of frames [Cooper, 2010] based on those of FrameNet (which uses Fillmore’s frame semantics).

Logical formalisms such as situation calculus (proposed by McCarthy in 1963) and event calculus have also been proposed as methods for reasoning about change and actions. As well as the STRIPS (Stanford Research Institute Problem Solver) system used for automated planning.

This project is based on the work in [Fernando, 2003, Fernando, 2004, Fernando and Nairn, 2005, Fernando, 2006] and especially [Fernando, 2011] where the system used here for reasoning about events and the methods on temporal intervals is formalised. The event structures analysed are based on those of Russell and Wiener and formalised in [Kamp and Reyle, 1993]. We take these several steps further, as in Fernando’s work, and re-evaluate events as strings in a regular language. We use the “snapshot” notion, where each snapshot represents events occurring at a particular time and are of the form

\[
\text{[overcast, Dublin]}
\]

These will be discussed further in the next chapter.
Many schemes have used Allen’s interval algebra [Allen, 1983] as a basis, using it to reason about the chronological order of events. This will also be discussed in the next chapter.

1.2 Motivation and Objective

Time is an important element of discourse, particularly when used in relation to events, actions and their effects. Moreover reasoning about events and their temporal relations is a difficult problem faced in the field of artificial intelligence. The ability to analyse events requires a method for reasoning if one is to understand many expressions in English.

Thus we require a notion of time with which to order events. In this project we will examine finite-state temporality introduced in Fernando’s work. This discussion is supplemented with an implementation of operations for extraction of information and temporal structure.

Barring finite-state temporality, most of the systems above have limitations to their implementation. As pointed out in [Pratt-Hartmann, 2007], gaining expressivity at the cost of increasing computational complexity is a common problem.

Finite-state temporality takes advantage of the flexibility and efficiency of finite-state methods. Infinite structures can be represented simply. So we take events as snapshots, treat them like strings and perform analysis on them in order to reveal more about their temporal structure and the information they contain. We formulate snapshots as runs of finite-state automata that constitute an order of causality and effect with which we can reason about temporality in natural language.

The main reason for this project is proving the tractability of the finite-state approach applied to event structures. Showing that the representation employed in this system is both expressive and that computers are capable using it for natural language analysis in a reasonable amount of time.
Chapter 2

Finite State Temporality

The primary idea behind finite-state temporality is the characterisation of events in natural language semantics as regular languages. The following sections will examine the background theory and build up a notion of an event based on these ideas.

2.1 Snapshots

In this project, as in previous work by Fernando, the notion used for time and events is the “snapshot”. Instances of events are contained in boxes as seen below:

\[
\text{overcast, Dublin}
\]

stating that in this particular instance it was overcast in Dublin.

We can imagine these representations as photographs, which similarly show an event occurring at the time the photo was taken. By viewing photographs we can extract information from them, “Was it sunny in Dublin at this time?” “Had it just rained?” etc. we receive boolean responses to these questions. In the theory presented in [Fernando and Nairn, 2005], we aim to derive facts from snapshots as one would from photos. The above example contains the facts 'overcast' and 'Dublin' from which we can derive that at the time the snapshot represents it is overcast and the location is Dublin.
When we wish to represent an event interval we place several snapshots together, to show multiple instances of events.

\[
\text{raining, Dublin} \quad \text{raining, London}
\]

As snapshots appear in chronological order, in this case we can say that it rained in London after it rained in Dublin.

To describe how snapshots are formulated we use temporal propositions, fluents and regular languages from formal language theory. We wish to represent the facts about events contained in natural language discourse in a way which makes it simple for a machine to reason about them. Thus it will be shown how they can be represented as strings and finite-state methods can be used to extract information about their temporal structure.

Propositions in natural language semantics are types of situations, defined as the set of possible worlds in which the proposition is true. For example, the proof of the proposition “John ate the scone” is an event where John eats a scone. Thus the sentence is true if there is an event that makes it true. We use temporal logic to reason about propositions conditional on time and the truth values of these statements can vary in time. We say a snapshot contains a list of temporal propositions that hold in the state it represents.

We can equally say that a snapshot is a set of fluents. A fluent is a condition which may change over time [McCarthy and Hayes, 1969], and may be represented as a temporal formula. It is effectively another way of speaking about temporal propositions. A propositional fluent has the range \( \{\text{true, false}\} \) and they can be used to give partial information about a situation.

As an example take a fluent as the value of the function

\[
raining(x)
\]

\(x\) could be true at some times at false at others, but it will never be both true and false at the same time. It could be raining at the current moment in time thus proving the statement true. However later on in time this statement could be false.
Thus we describe a snapshot as a set of fluents or temporal formulae that are true at the moment the snapshot represents.

Events in this project will be represented as a series of snapshots, if we view a snapshot

\[\text{raining, Dublin}\]

like a photograph or even better, a frame in a film, we can portray an event as a filmstrip where each frame is an instance of the event. So each event is a list of sets of temporal formulae as follows:

\[\text{raining, Dublin, Dublin, Dublin, Dublin}\]

where the snapshots or frames in the filmstrip are in chronological order. If we call the snapshots symbols, we can call the filmstrip above a string.

We can turn the filmstrip of events shown above into a regular expression with Kleene plus:

\[\text{raining, Dublin}^+\]

This new string signifies one or more occurrences of it raining in Dublin, a larger interval of events. Since a regular expression describes a regular language, we can view a filmstrip as a string in a regular language consisting of a temporal sequence of observations. This is a language over the alphabet of snapshot symbols (usually denoted as \(\Sigma\), this report follows Fernando’s example of using \(\Phi\)) consisting of strings in \(\Phi^+\). We use Kleene star (\(\ast\)) and Kleene plus (\(+\)) notation on symbols in the language to form strings:

\[
\phi^+ = \epsilon \cup \phi_1 \cup \phi_2 \cup \ldots \cup \phi_n \\
\phi^* = \epsilon \cup \phi_1 \cup \phi_2 \cup \ldots \cup \phi_n
\]

for \(\phi_i, 1 \leq i \leq n\) and where \(\epsilon\) denotes the empty string. An example of regular expressions applied to snapshots is:

\[p^+ \square^+ = p \mid pp \mid pp \mid ppp \mid pp \mid ppp \mid \ldots\]
Since any regular language can be recognised by some finite-state automaton, we use the alphabet of symbols Φ as an input alphabet for an FSA recognising strings of the regular language describes rules for the occurrence of events.

From [Fernando, 2007] we have:

- A temporal proposition: φ
- A symbol (snapshot): \( a_i = [\phi, \ldots] \)
- A string (film strip) = \( a_1 a_2 \ldots a_n \)

\( a_i \) is seen as a set of propositions rather than a single temporal proposition. So we construe a string \( a_1, a_2, \ldots, a_n \in \text{Pow}(\Phi) \) as an event of \( n \) successive moments with every fluent \( a_i \) asserted to hold at the \( i_{th} \) moment. Similar to set theory, the order of propositions as well as the number of occurrences of a proposition in a single snapshot is unimportant, formulae that appear in the same snapshot are true at the same time.

\[
\psi, \phi = \phi, \psi = \psi, \phi, \psi
\]

These snapshots represent simultaneous events φ and ψ. It is also not necessary for \( a_i \) to include every temporal proposition that is true in the world at a certain time, the absence of a formula does not mean it doesn’t hold at that time. Where we have an empty snapshot □, it simply means that no formula can be derived from it, that it’s contents are not significant. It is not the same as the empty set. □ is an element of the set containing only □, the empty set contains no strings.

The idea of symbols as snapshots is reinforced by placing φ in a box, which is described as a set whose elements are written inside of it.

Since we call a string a filmstrip, let us then call a finite-state automaton a film projector. We run through a string in an FSA almost as a filmstrip would run through a projector, where a string would have certain properties, a large to infinite number of occurrences for a symbol, an FSA would recognise this with a simple loop on a state. An FSA is capable of more complex operations, such as giving no output for a certain input (epsilon transitions).
In order to analyze events in natural language semantics, we require a notion of time or temporal intervals. In this project events have been represented as ‘snapshots’, we treat each snapshot as a symbol consisting of propositions that hold at the point in time the snapshot represents. We view these snapshots as symbols of the input alphabet given to a finite-state automaton, so they can be described using a regular language.

2.2 Finite State Automata

Any regular expression can be implemented as a finite-state automaton and similarly any finite state automaton can be described using a regular expression. Furthermore both of these can be used to describe a regular language, a specialised type of formal language. Using a finite state machine we can decide whether or not a string is part of a particular language, based on the set of strings a certain FSM accepts.

Formally, a deterministic finite state automaton is a 5-tuple \((Q, \Sigma, \delta, q_0, F)\) consisting of:

- a finite set of \(n\) states: \(Q = \{q_0, q_1, q_2, \ldots, q_{n-1}\}\)
- a finite input alphabet of symbols \(\Sigma\)
- a start state \(q_0 \in Q\)
- a set of accepting or final states \(F \subseteq Q\)
- a transition function \(\delta\), a relation from \(Q \times \Sigma\) to \(Q\). Given state \(q \in Q\) and input symbol \(i \in \Sigma\), \(\delta(i, q)\) returns new state \(q' \in Q\).
Figure 2.1: Example of a simple FSM accepting the language $L = \{\text{ha!}, \text{haha!}, \text{hahaha!}, \ldots\}$ with start state $q_0$, accepting state $q_3$ and four transitions.

Non-deterministic finite state automata can be similarly described but differ in that the transition function is $\delta : Q \times (\Sigma \cup \{\epsilon\}) \to \text{Pow}(Q)$, where $\text{Pow}(Q)$ is the power set of $Q$, and $\epsilon$ is the empty string.

A deterministic finite-state automaton has no choice points, for a given input and state there is at most one transition. Non-deterministic automata make decisions based on the input. An NFSA will accept a string if there is a path which begins at the start state and ends on some final state after reading the whole string. Non-deterministic FSAs also permit epsilon-transitions ($\epsilon$-transitions), which allows the automaton to move states without looking at the input. It also allows strings of symbols to be read in one movement.

A finite state transducer acts in a similar fashion as the finite state automaton, indeed, it is an extension of the automaton, a transducer maps between one representation and another. For every symbol input it outputs another symbol. When comparing a finite-state automaton to a Turing machine, we say there is a tape which can be used to read input. On a finite-state transducer there are two tapes, input and output. The formal definition of a finite-state transducer is again similar to the formal description of a finite-state automaton. It has a finite set of $n$ states,
$Q$, a finite set $\Sigma$ of input symbols, a start state $q_0$ and a set of final states $F \subseteq Q$. However it differs as follows:

- It has a finite set $\Delta$, the output alphabet.
- The transition function $\delta(q, w)$. Given a state $q$ and a string $w \in \Sigma^*$, the function returns a set of states as the given input may be ambiguous as to which state it maps to. This is thus a function from $Q \times \Sigma^*$ to $2^Q$.
- The output function $\sigma(q, w)$ gives a set of possible output strings $o \in \Delta$ for each state $q \in Q$ and input string $w \in \Sigma^*$. It is thus a function from $Q \times \Sigma^*$ to $2^\Delta$.

Operations performed on FSTs include intersection, projection and composition. The application developed in this project makes much use of composition.

From [Jurafsky and Martin, 2009] composition of two FSTs is defined as:

**Composition:** If $T_1$ is a transducer from $I_1$ to $O_1$ and $T_2$ is a transducer from $O_1$ to $O_2$, then $T_1 \circ T_2$ maps from $I_1$ to $O_2$.

This operation allows us to replace two transducers previously running in series with a single more complex transducer. Applying $T_1 \circ T_2$ to an input sequence $S$ yields the same result as applying $T_1$ to $S$ and then applying $T_2$ to its output. Thus $T_1 \circ T_2 = T_2(T_1(S))$. [Jurafsky and Martin, 2009] Although the normal way of viewing $T_1 \circ T_2$ is $T_1$ after $T_2$, however in this project it will be seen as $T_2$ after $T_1$. The implementation of this in Prolog will be covered in the next chapter.

### 2.3 Event Structures

The notion of events formalised by Russell and Wiener and modified by Kamp is a 3-tuple $\langle E, \circ, \prec \rangle$, where $E$ is a nonempty set of events and $\circ$ and $\prec$ are binary operations on $E$ of temporal overlap and complete precedence respectively. Thus this structure permits the expression of temporal relationships between events. Take for example two events $e$ and $e' \in E$. The statement $e \circ e'$ denotes that the events overlap temporally, thus taking place mostly or
totally at the same time. $e \prec e'$ states that $e$ starts and completely finishes by the time $e'$ starts.

This event structure thus satisfies the following axioms:

1. $e_1 \prec e_2 \rightarrow \neg e_2 \prec e_1$
2. $e_1 \prec e_2 \& e_2 \prec e_3 \rightarrow e_1 \prec e_3$
3. $e \bigcirc e$
4. $e_1 \bigcirc e_2 \rightarrow e_2 \bigcirc e_1$
5. $e_1 \prec e_2 \rightarrow \neg e_2 \bigcirc e_1$
6. $e_1 \prec e_2 \& e_2 \bigcirc e_3 \& e_3 \prec e_4 \rightarrow e_1 \prec e_4$
7. $e_1 \prec e_2 \lor e_1 \bigcirc e_2 \lor e_2 \prec e_1$

where $e_1, e_2, e_3, e_4$ are elements (events) in $E$ [Kamp and Reyle, 1993].

This notion of events will be looked at further as a context for the fluents in the next section.

Much work in artificial intelligence has used Allen’s interval algebra as a basis when discussing temporal relations. The algebra consists of relations that exist between intervals of time, of which there may be as many as $2^{13}$ but can be expressed as vectors of definite simple relations, of which there are 13. These denote the set of possible simple relations that hold between two intervals.

Kamp has shown how to flesh out the notion of time implicit in an event structure through moments, relative to which events stretch over intervals [Fernando, 2007]. The relations between events described with $\prec$ and $\bigcirc$ can be expressed as disjunctions of the Allen interval relations.
Figure 2.2: A line representation of the Allen basic interval relations.
2.4 Allen interval relations

As stated above, the simple interval relations described by [Allen, 1983] are used as a basis for most systems looking to analyse temporal relations. They are normally used in the context of deduction of temporal information.

Further on we will show that like Russell-Wiener-Kamp event structures, finite-state temporality can also be used to extract the basic interval relations.

Below is a reformulation of figure 2.1 in terms of snapshots, from [Fernando, 2006]

<table>
<thead>
<tr>
<th>$p$ before $q$</th>
<th>$p$</th>
<th>$q$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p$ meets $q$</td>
<td>$p$</td>
<td>$q$</td>
</tr>
<tr>
<td>$p$ overlaps $q$</td>
<td>$p$</td>
<td>$p$, $q$</td>
</tr>
<tr>
<td>$p$ starts $q$</td>
<td>$p$, $q$</td>
<td>$q$</td>
</tr>
<tr>
<td>$p$ during $q$</td>
<td>$q$, $p$, $q$</td>
<td>$q$</td>
</tr>
<tr>
<td>$p$ finishes $q$</td>
<td>$q$, $p$, $q$</td>
<td>$q$</td>
</tr>
<tr>
<td>$p$ equals $q$</td>
<td>$p$, $q$</td>
<td>$p$, $q$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$p$ after $q$</th>
<th>$q$</th>
<th>$p$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p$ met-by $q$</td>
<td>$q$</td>
<td>$p$</td>
</tr>
<tr>
<td>$p$ overlapped-by $q$</td>
<td>$q$, $p$, $q$</td>
<td>$p$</td>
</tr>
<tr>
<td>$p$ started-by $q$</td>
<td>$p$, $q$</td>
<td>$q$</td>
</tr>
<tr>
<td>$p$ constrains $q$</td>
<td>$p$, $p$, $q$</td>
<td>$p$</td>
</tr>
<tr>
<td>$p$ finish-by $q$</td>
<td>$p$, $p$, $q$</td>
<td>$p$, $p$, $q$</td>
</tr>
</tbody>
</table>

The difference between “$p$ before $q$” and “$p$ meets $q$” is that when two snapshots meet, there is no waiting time between them. In the former there is an empty box (consisting of non-derivable formulae, not the empty set) between the two.
Chapter 3

Implementation

In the following sections, we describe operations on finite event structures designed for extraction of temporal information. Some theoretical aspects of finite state temporality will be explained and discussed in relation to the research discussed in the previous chapter. Definitions for the operations on events will be given and their implementation as finite-state automata in Prolog explained.

As will be stated below, a regular language \( L \) over \( \Phi \) is denoted by a set of strings in \( \text{Pow}(\Phi) \), where internally strings are sequences of sets of fluents. These strings are manipulated by the operations for the extraction of temporal information.

In Prolog, an FSA is represented as a list of the form:

\[
\text{FSA} = [S, T, F]
\]

where \( S \) is the start state (usually \( q_0 \)), \( F \) is a list of accepting states and \( T \) is a list of transitions from the current state to another state, where each transition is a list of three arguments written as:

\[
[\text{CurState}, \text{Label}, \text{NextState}]
\]

\text{Label} is the symbol input to the FSM. In an FST there is an second \text{Label} for the symbol output. The labels may be a single atom or a list of atoms, reflecting snapshots containing a list of propositions.
Apart from what was implemented in [Cummins, 2010], the following sections were previously only theoretical ideas without implementation. The operations previously written in Prolog were independently implemented here. Previous work in this area was extended by implementing the constrain, unpad, $\pi$ relations and beginning on the circumscribe relation.

The following sections will define an operation, explain it and then discuss its implementation in Prolog. The final section in this chapter will examine the problems encountered and methods employed to overcome them.

### 3.1 Superposition

The associative binary operation $\&$ of superposition on languages can be used in a more abstract sense to reveal the temporal structure of event sequences. When applied to two strings of the same length the operation forms their componentwise union:

$$\alpha_1 \cdots \alpha_n \& \alpha'_1 \cdots \alpha'_n \overset{\text{def}}{=} (\alpha_1 \cup \alpha'_1) \cdots (\alpha_n \cup \alpha'_n)$$

[Fernando, 2011]

Given a finite set $\Phi$ of formulas and languages $L, L' \subseteq \text{Pow}(\Phi)^*$, the definition of the superposition $L \& L'$ of $L$ and $L'$ is:

$$L \& L' \overset{\text{def}}{=} \bigcup_{n \geq 0} \{ (a_1 \cup a'_1) \cdots (a_n \cup a'_n) \mid a_1 \cdots a_n \in L \text{ and } a'_1 \cdots a'_n \in L' \}$$

This forms the componentwise union of nonempty strings in $L$ and $L'$ of the same length $n$ and produces an expansion of $L$ and $L'$, $L \& L'$, which subsumes them both. This property and subsumption will be explained further in this

Where we have two finite automata, $A$ and $A'$ accepting $L$ and $L'$ respectively, the finite automaton accepting $L \& L'$ is given as follows:

- The set of states is the Cartesian product $Q \times Q'$ where $Q$ and $Q'$ are the sets of states for $A$ and $A'$ respectively.
- The initial state is the pair $(q0, q'0)$ where $q0$ is the initial state of $A$ and $q'0$ is the initial state of $A'$.
• The set of final states is the product $F \times F'$ of final states of $A$ and $A'$.

• And the set of transitions consists of the transitions $(q,q') \xrightarrow{\alpha,\alpha'} (r,r')$ for all transitions $q \xrightarrow{\alpha} r$ in $A$ and transitions $q' \xrightarrow{\alpha'} r'$ in $A'$.

The predicate as written is Prolog is superpose/3 which requires two input FSMs and outputs an FSM as the superposition of the two input.

\[ \text{superpose}(+\text{FSM1}, +\text{FSM2}, ?\text{SupFSM}). \]

In the code it is written as:

\[ \text{superpose}([\text{Start1, Trans1, Final1}], [\text{Start2, Trans2, Final2}], [\text{Start3, Trans3, Final3}]) \]

\textbf{Start3} is seen, in terms of the overall finite-state machine, as a single state, however internally it is a pair consisting of the start states of the two input machines. It is then instantiated as a list consisting of \textbf{Start1} and \textbf{Start2}.

\textbf{Final3} is seen as a list of states as in the input FSMs, however it is in fact a list of pairs of final states taken from \textbf{FSM1} and \textbf{FSM2}.

The interesting part is the unification of labels for each transition, for each list in \textbf{Trans1} and \textbf{Trans2} we place the states the transitions start at in a pair, written as a list. We do the same with the states the transitions end on. The labels of both transition lists are placed in a list together. So from 1 and 2 we get 3 (Here the transitions for FSM1 and FSM2 are short for the sake of simplicity).

\textbf{Trans1} = [ [q0, [a,b], q1], [q1, c, q2] ].
\textbf{Trans2} = [ [r0, d, r1], [r1, e, r2] ].
\textbf{Trans3} = [[[q0,r0],[a,b,c],[q1,r1]],[[q1,r1],[b,d],[q2,r2]]].

Going back to our snapshot description from earlier we can again say the strings recognised by the FSMs are filmstrips and the FSMs are film projectors. In the case of superposition we can imagine two film projectors running together at the same time, both pointed towards the same screen. The two
film strips are then projected over each other and are thus superposed. The following are some examples of strings superposed:

\[
\begin{align*}
pq \land rs &= p, r, q, s \\
pq^+ \land r^+ s &= p, r, q, r, q, s \\
\end{align*}
\]

where \( pq \in L \) and \( rs \in L' \)

We can create the language to represent the event where it rains from dawn until dusk, namely: \textit{rain from dawn to dusk} from [Fernando, 2006], using superposition

\[
\begin{align*}
\text{rain}^+ \land \text{dawn}^+ \land \text{dusk}^+ &= \text{rain, dawn}[\text{rain}] \text{rain, dusk}
\end{align*}
\]

which consists of strings of the form \( \text{rain,dawn}[\text{rain}] \text{rain,dusk} \) for \( n \geq 0 \) there are \((n + 2)\) snapshots, all depicting rain, starting at dawn and ending at dusk. According to [Fernando, 2006], “the different values of \( n \) support models at different levels of granularity (the larger the \( n \), the finer the grain)”. 

3.2 Subsumption and Constraints

Subsumption between two strings in $\text{Pow}(\Phi)$ is given as:

$$\alpha_1 \cdots \alpha_n \supseteq \alpha'_1 \cdots \alpha'_n \overset{\text{def}}{\iff} n = m \text{ and } \alpha_i \supseteq \alpha'_i \text{ for } 1 \leq i \leq n$$

for all $\alpha, \alpha' \in \Phi$. Inclusion in sets is extended to our strings, so subsumption holds between strings when they are of the same length and $\subseteq$ holds componentwise.

When we say $\alpha_i \supseteq \alpha'_i$, it means that $\alpha_i$ is at least as informative as $\alpha'_i$. The set of formulae contained in $\alpha'_i$ is a subset of those contained in $\alpha_i$. \cite{Fernando and Nairn, 2005}

An example is as follows: string $s$ describes a commuter taking the 46a bus to its terminus, and string $s'$ simply describes the acts of boarding and
travelling on the 46a bus. These can be written in snapshot form as:

\[ s = \text{walk-to-bus-stop horizontal line board-46a horizontal line ride-46a horizontal line alight-at-terminus} \]

\[ s' = \text{board-46a horizontal line ride-46a} \]

Here, \( s \triangleright s' \) or, similar to above, it can equally be said that \( s \) contains more information than \( s' \). Looking at each snapshot individually in the above strings, we can see that each corresponding snapshot between them holds for \( \supseteq \).

\[ \begin{align*}
\text{walk-to-bus-stop} & \supseteq \Box \\
\text{board-46a} & \supseteq \text{board-46a} \\
\text{ride-46a} & \supseteq \text{ride-46a} \\
\text{alight-at-terminus} & \supseteq \Box
\end{align*} \]

Where \( \Box \) is an empty snapshot, not the empty set. It describes the case where the fluents it contains are unknown.

In terms of movie projectors, imagine \( s \) being played as a film, when the projector running through \( s \) plays \( \text{walk-to-bus-stop} \), the projector for \( s' \) plays \( \Box \). When \( \text{board-46a} \) is played for \( s \), \( \text{board-46a} \) is played for \( s' \) at the same time and so on.

Given a language \( L \) over \( \text{Pow}(\Phi) \), we let \( s \triangleright L \) means \( s \) belongs to the Pierce product of subsumption \( \langle \triangleright \rangle L \) for that language [Fernando, 2011]:

\[ s \triangleright L \iff s \in \langle \triangleright \rangle L \iff (\exists s' \in L) s \triangleright s' \]

\( \triangleright \) is a regular operation, if \( L \) is accepted by a finite-state machine \( \langle Q, q_0, F, \rightarrow \rangle \) then \( \langle \triangleright \rangle L \) is accepted by the FSM \( \langle Q, q_0, F, \rightarrow \rangle \) where

\[ q \sim q' \text{ iff } (\exists \alpha \subseteq \beta)q \overset{\alpha}{\rightarrow} q' \]
Subsumption between two languages $L$ and $L'$ is defined as

$$L \supset L' \text{ iff } (\forall s \in L)(\exists s' \in L')s \supseteq s'$$

So for all strings in $L$ there must be a string in $L'$ that is subsumed by it. We can also state that $L \supset L'$ if and only if $L \subseteq \langle \supset \rangle L$

As stated above $L & L'$ subsumes both $L$ and $L'$. In other words $L & L' \subseteq \langle \supset \rangle L$. As when two languages are superposed there is a growth of information, thus the resulting language subsumes both the component languages.

Subsumption is implemented in Prolog as the `subsume` predicate. As in [Fernando, 2011] transitions in the FSTs are of the form $0 \xrightarrow{\alpha \alpha'} 0$. Each symbol is compared between FSTs for $\subseteq$.

```
subsume(Phi, [S1,T1,F1], [S1,T2,F1]):-
    findall([0, L1, L, 0],(member([0, L1, L, 0],T1), member(L, Phi),
    subset(L,L1)), T2).
```

### 3.3 Constrain

Building on subsumption, the constrain operator is based on Koskenniemi’s restriction [Fernando and Nairn, 2005]. This is used to express entailments as regular languages.

$$\phi \vdash \check{\phi} \Rightarrow \emptyset$$

which states that a formula $\phi$ and its negation $\check{\phi}$ cannot exist in the same snapshot.

Given two languages $L$ and $L'$ over $\text{Pow}(\Phi)$, let $L \Rightarrow L'$ be the set of strings $s$ such that every factor of $s$ that subsumes $L$ also subsumes $L'$.

$$L \Rightarrow L' \overset{\text{def}}{=} \{ s \in \text{Pow}(\Phi) \mid \text{for every factor } s' \text{ of } s, s' \supseteq L \text{ implies } s' \supseteq L' \}$$
Here \( s' \) is a factor of \( s \) if, for some strings \( u,v \), \( s = us'v \).

We pronounce \( L \Rightarrow L' \) as “\( L \) constrains \( L' \)”. From the regularity of the relations used to define \( L \Rightarrow L' \) we can conclude this relation is regular if \( L \) and \( L' \) are regular [Fernando, 2011].

A more complex way of defining constrain is as follows
\[
L \Rightarrow L' = \text{Pow}(\Phi)^* \langle [\geq]L \cap [\geq]L' \rangle \text{Pow}(\Phi)^*
\]

where \( \overline{L} \) is the complement of \( L \) in \( \text{Pow}(\Phi) \) otherwise written as \( \overline{L} = \text{Pow}(\Phi)^* - L \).

The constrain operation is designed in Prolog as the predicate `constrain/3`, which takes three FSMs. Using `forall`, it checks whether the strings accepted by the third FSM subsume both the first and second FSMs passed, or just the second FSM passed.

\[
\text{constrain}([S1,Trans1,F1],[S2,Trans2,F2],[S3,Trans3,F3]):-
\]

### 3.4 Restriction

For any finite subset \( X \) of \( \Phi \), we form a restriction of a string (a restriction of observations) in \( \text{Pow}(\Phi) \) to one in \( \text{Pow}(X) \) by forming the componentwise intersection \( r_X \) with \( X \)

\[
r_X(\alpha_1 \ldots \alpha_n) = (\alpha_1 \cap X) \ldots (\alpha_n \cap X)
\]

so if \( X = \{e'\} \) then \( r_X \) projects the string

\[
\begin{align*}
\text{r}_X(\text{e} \text{e}, e, e, e') &= \text{e' e' e'}
\end{align*}
\]

The restrict operation was discussed at the beginning of this project, however other relations took precedence as time went on. Some simple code was written but it needs to be rewritten to allow it to be used with the rest of the code.
3.5 Composition

Here the implementation of composition between transducers as well as between a finite-state transducer and a finite-state machine will be discussed. Continuing from the definition of composition $\circ$ between two transducers in section 2.3, we go more in-depth with a definition which includes discussion of the input and output alphabets.

The operation of composition $\circ$ between two finite-state transducers, $T_1$ on alphabets $\Sigma$ and $\Delta$ and $T_2$ on $\Delta$ and $\Gamma$, produces a transducer $T_1 \circ T_2$ from $\Sigma$ to $\Gamma$ such that $x \rightarrow z$ maps $x$ to $z$ in $T_1 \circ T_2$ if there exists a string $y \in \Delta^*$ and there is a mapping for $x \rightarrow y$ in $T_1$ and for $y \rightarrow z$ in $T_2$.

In the Prolog code composition has two predicates, \texttt{composeFST} and \texttt{composeFSM}. \texttt{composeFST} takes two finite-state transducers in the list format specified at the beginning of this chapter as input and outputs an FST for their composition. It first collects pairs of the final states from both transducers, one state from each, to create the list of accepting states. It also creates a single state consisting of both start states, the start state for the returned FST. The transitions for this composed FST are formed by checking whether there is a transition in the first FST with an input $I$ and an output $R$ and whether there is a transition in the second FST with the input $R$ and output $O$. If this is the case then a new transition may be created which begins in a state consisting of the pair of the start states in the original transitions and ending on the pair of the next states taken from the original transitions.

This is done using a call to the \texttt{findall/3} predicate.

\begin{verbatim}
findall([[A,B],I,O,[C,D]],
  (member([A,I,R,C], Trans1), member([B,R,O,D], Trans2)),L).
\end{verbatim}

\texttt{composeFSM} takes an FSM and an FST as an input and outputs another FSM, the composition of the two. It takes the same approach to the start and final states as \texttt{composeFST}, however when creating the new list of transitions, it will use only the output of the FST as a transition label.

Using a method similar to the one employed above, this predicate finds
all transitions in the input FSM for which the label is an input symbol in a transition in the input FST. The output symbol for this input is taken from the FST and used to create a new transition for the resulting composed FSM. This transition is formed using the pair of the current states and the pair of the states to be transitioned to in the FST and FSM, with the output symbol used as the label.

\[
\text{findall([Q,0,R],}
\begin{align*}
\text{(member([A,I,C], Trans1), member([B,I,O,D], Trans2),} \\
\text{Q=[A,B],R=[C,D]),} \\
\text{Trans3)}
\end{align*}
\]

The implementation for composition will be especially useful when it comes to the \(\pi\) operation below.

Figure 3.2: Example of composition on two simple FSTs
3.6 Block Compression

Where we have \[\text{rain,dawn, rain, rain, dusk}\] from section 3.1, we can reduce a string of infinite size to a simple string of distinct events, namely the three snapshots \[\text{rain,dawn, rain} \] and \[\text{rain, dusk}\].

An operation for the reduction of a string, \(s = \alpha_1, \alpha_2, \ldots, \alpha_n\) consisting of \(n\) copies of \(\alpha\), to a string of one copy \(\alpha\) is defined as follows:

\[
bc(s) \overset{\text{def}}{=} \begin{cases} 
bc(\alpha s') & \text{if } s = \alpha \alpha s' \\
\alpha bc(\beta s') & \text{if } s = \alpha \beta s' \text{ with } \alpha \neq \beta \\
s & \text{otherwise (}\text{length}(s) \leq 1)\end{cases}
\]

for all \(\alpha, \beta \subseteq \Phi\). This operation is designed to reduce intervals of events, reducing a block \(\alpha \alpha\) to \(\alpha\), in line with the dictum “no time without change” [Fernando, 2007]. Meaning, we view time relative to events, it depends on the ordering of events. If there are no events (or in this case, no new events), there is no time.

Below are two examples of block compression of snapshots:

\[
bc(\text{a,b a,b a,b a,b,c a,b,c a,b}) = \text{a,b a,b,}c\text{a,b} \\
bc(\text{p p p q q p}) = \text{p q p}
\]

Block compression was implemented in Prolog as the predicate \(\text{bcfst}\), which creates a block-compressing finite-state transducer using the alphabet (list of symbols) given as an input.

\[
\text{bcfst}(\Phi, \text{FST})
\]

Apart from the start state \(q_0\), each state in this FST is given the name of the symbol input to reach it. That is, if the symbol \(\alpha\) is input when in the start state, the transducer will move to state \(\alpha\). This helps keep track of the input, especially if several identical symbols are input consecutively. For the input \(\alpha\), the transducer will move from \(q_0\) to state \(\alpha\), printing \(\alpha\). If \(\alpha\) is input when the transducer is already in state \(\alpha\), the transition will consist of
a loop on the state, printing $\epsilon$ (nothing).

To create the finite-state transducer for block-compression, this predicate creates the start state $q_0$ and a list of final states consisting of $q_0$ and the symbols of the alphabet, hence any state is an accepting state in this FST. It then forms the transitions for the FST with the `findall/3` predicate.

\[
\begin{align*}
\text{findall}([q_0,X,X,X], \text{member}(X, \Phi), L1) \\
\text{findall}([X,Y,Y,Y], (\text{member}(X, \Phi), \text{member}(Y, \Phi), Y \neq X), L2) \\
\text{findall}([X,X,0,X], \text{member}(X, \Phi), L3)
\end{align*}
\]

There are three types of transitions. The first is a transition from the start state to any other state for the elements of the alphabet. The second portrays transitions between states where the input symbol is not the same as the symbol for the current state, e.g. the transducer is in state $\alpha$ and input $\beta$ is received, so it moves to state $\beta$. The third type of transition is a loop for instances where the input symbol is the same as the name of the state, i.e. this symbol has already been input. In this case, $\epsilon$ (written in the code above as 0) is output. $L_1, L_2$ and $L_3$ are appended together to form the full list of transitions.
3.7 Unpad

Where there are empty snapshots at the beginning and end of a string, we can view it as uniformative padding [Fernando, 2007].

Given a string $s$, we can delete initial and final $\square$'s using the following method:

$$
\text{unpad}(s) \triangleq \begin{cases} 
\text{unpad}(s') & \text{if } s = \square s' \text{ or else if } s = s' \square \\
 s & \text{otherwise}
\end{cases}
$$

For example:

$$
\text{unpad}(\square s \square^*) = \text{unpad}(s)
$$

This operation was implemented in a similar fashion to block compression, the unpadfst/2 predicate forms an unpadded FST based on a given alphabet. Again the FST returned may be composed with an FSM to perform unpadding.

unpadfst(Phi,FST)
Although similar, there are only three states in the unpadding FST, q0, q1 and q2. As with the block-compressing FST, unpadfst creates a start state q0, however the set of final states only consists of q1 and q2. There are four types of transitions

- When in q0, if a box is input (represented in the code as sq), output $\epsilon$ (0 in the code) and stay in q0.
- When in q0, if any symbol other than a box is input, print that symbol and move to state q1.
- When in q1, for any symbol other than a box, output the symbol and loop on q1.
- When in q1 and a box is input, output epsilon and move to state q2.
- In q2, loop on q2 outputting epsilon when a box is input.

![Diagram of FST](image)

Figure 3.4: An FST for unpadding over the alphabet $\{\square, \alpha, \beta\}$, where $\square$ is represented as sq and 0 is $\epsilon$ (empty-transition).

The operations for block-compression and unpadding will be used further in the next section.
3.8 Pi and further superposition

unpad and bc are used in sequence to form the projection $\pi$ on a string $s$

$$\pi(s) = \text{unpad}\,(bc(s)) = bc(\text{unpad}(s))$$

$$\pi(s) = [p] \text{ iff } s \in [p]^+[\square]$$

$$\pi(\square^n \text{dawn} \text{rain} \text{dusk} \square^n) = \text{dawn} \text{rain} \text{dusk}$$

Fernando, 2007

for all $n, m \geq 0$ and for all $i, j, k > 0$.

Using the Kleene star and Kleene plus, we can create the inverse of $\pi$

$$\pi^{-1}(p) \overset{\text{def}}{=} □^* p □^*$$

and apply it to the rain from dawn to dusk example

$$\pi^{-1}(\text{dawn} \text{rain} \text{dusk}) = \square^\dagger \text{dawn} \text{rain} \text{dusk} □^\dagger \square^*$$

We can also describe $\pi^{-1}(s)$ as giving approximate observations on $s$ which allows for a period of waiting at the start and end. $\pi^{-1}$ does produce a string that can be weakly subsumed by the string returned by $\pi$, but it is not exactly the inverse of $\pi$.

Applying $\pi$ to a language $L$ over $\Phi$ yields

$$L_\pi \overset{\text{def}}{=} \{ s \in \text{Pow}(\Phi) \mid (\exists s' \in L) \pi (s) = \pi(s') \} = \pi^{-1}(\pi(L))$$

1Weak subsumption will not be covered in depth here, it appears in [Fernando, 2010]. Effectively $s \sqsupseteq s' = (\exists s_o, \text{unpad}(s_o) = \text{unpad}(s))(\exists s'_o, \text{unpad}(s'_o) = \text{unpad}(s')) s_o \geq s'_o$. This means that for $s$ and $s'$ to weakly subsume each other there must exist two strings, $s_o$ and $s'_o$ which, when unpadded, are the same as the unpadded versions of $s$ and $s'$ respectively, and $s_o$ subsumes $s'_o$. 28
resulting in the set of strings over \( \text{Pow}(\Phi) \) which, when \( \pi \) is applied, produces the same string as one would get if \( \pi \) was applied to a string in \( L \).

This is done in Prolog using composition. A block-compressing FST and an unpadding FST are created and composed together to create one FST which performs both block-compression and unpadding. The \( \pi \) predicate takes as input the alphabet \( \Phi \) and outputs the FST. This FST may then be composed with another FST or FSM to perform block-compression and unpadding on the latter.

\[
\pi(A, \text{FST})
\]

Figure 3.5: Example of an FSM before and after \( \pi \) is applied to it.

A predicate for \( \pi^{-1} \) was also defined. As with \( \pi \), this takes an alphabet \( \Phi \) and returns an FST. \( \text{unpi} \) calls two predicates named \( \text{unbcfst} \) and \( \text{repadfst} \) which effectively perform the reverse of block compression and unpadding respectively. \( \text{unbcfst} \) creates an FST which accepts one or more
(Kleene plus) symbols of the language. The FST returned by \texttt{repadfst} accepts zero or more (Kleene star) squares.

\[
\text{unpi}(\Phi, \text{FST}) :\neg
\begin{align*}
\text{unbcfst}(\Phi, \text{BCFST}), \\
\text{repadfst}(\text{BCFST}, \text{FST}).
\end{align*}
\]

Instead of composing the FSTs produced by \texttt{unbcfst} and \texttt{repadfst}, the operations are applied in sequence. The FST returned by \texttt{unbcfst} is taken as input to \texttt{repadfst} which extends it to accept $\Box^*$ at the beginning and end.

Using $\pi$, we can now define the pi-superposition operation as

\[
L \& \pi L' \overset{\text{def}}{=} \pi(L \& L'_\pi)
\]

In the Prolog code, this is implemented as the predicate \texttt{pisuperpose/4}, which takes an alphabet and two finite-state machines presented as lists and returns a “pisuperposed” FSM.

\[
\text{pisuperpose}(\Phi, \text{FSM1}, \text{FSM2}, \text{SupFSM}).
\]

This is one of the longer and more complex predicates, initially giving subtly incorrect or not broad enough answers and requiring changes in the predicates it calls to achieve correct responses. $\pi$ is called and \texttt{FSM1} and \texttt{FSM2} are composed with its output FST respectively. The FSMs retrieved from this are then composed with the FST returned by \texttt{unpi}, evolving the FSMs again to simulate $\pi^{-1}(\pi(L))$. The FSMs are superposed using the \texttt{superpose} predicate discussed above and then finally the returned FSM is composed with the FST of $\pi$ to get an FSM which acts as $L \& \pi L'$.

\[
\text{pisuperpose}(\Phi, \text{FSM1}, \text{FSM2}, \text{PiSupFSM}):\neg
\begin{align*}
\pi(\Phi, \text{FST}), \\
\text{composeFSM}(\text{FSM1}, \text{FST}, \text{TempCFSM1}), \\
\text{removeDeadTransFSM}(\text{TempCFSM1}, \text{CFSM1}), \\
\text{composeFSM}(\text{FSM2}, \text{FST}, \text{TempCFSM2}), \\
\text{removeDeadTransFSM}(\text{TempCFSM2}, \text{CFSM2}), \\
\text{unpi}(\Phi, \text{UFST}), \\
\end{align*}
\]\n
30
removeDeadTransFSM is a predicate that was designed to remove the unreachable transitions created by composeFSM. That is, using the setof /3 predicate and keeping the beginning state and the label static (disallowing backtracking on these variables with `), it picks out the transitions of which the beginning state is an ending state in another transition or if the beginning state is the same as the start state.

It also removes the singular final states produced, the states which have nothing transitioning to them.

removeDeadTransFSM([S,OrigTrans,F], [S,NewTrans,NewF]):-
    setof([CS,L,NS],
        Q Label (` member([CS,L,NS], OrigTrans),
        (member([Q,Label,CS],OrigTrans);CS=S) ), NewTrans),
    findall(Q, ( member(Q,F),
        (member([CS,L,Q], NewTrans);member(Q,S)) ), NewF).

3.8.1 Allen relations revisited

The language below represents the Allen relations from section 2.4

\[
\text{Allen}(p,q) \overset{\text{def}}{=} \left[ p | \epsilon \right] \left[ q | \epsilon \right] (\left[ p \right] | q) \left( \left[ p, q \right] | \epsilon \right) \left[ q | (\epsilon | p) \right]
\]

we can extract these strings using $\pi$ superposition $\&_{\pi}$ on events $\epsilon$ and $\epsilon'$

\[
\epsilon \&_{\pi} \epsilon' = \pi(\epsilon \epsilon' + \epsilon' \epsilon + \epsilon \epsilon' + \epsilon' \epsilon')
\]

according to [Fernando, 2006] we can partition the language $\epsilon \epsilon' + \epsilon' \epsilon + \epsilon \epsilon' + \epsilon' \epsilon'$ into three disjoint sublanguages
Applying $\pi$ to these strings we get

1. $\pi(\boxdot e \uparrow \boxdot e' \uparrow \square) = e | \square | e'$
2. $\pi'(\boxdot(e | e') \downarrow e,e' \uparrow (e' | e') \boxdot) = (e | e') | e,e' | e | e'$
3. $\pi'(e' \uparrow \square | e \uparrow \square) = e' | \square | e$

Thus we can state that $e \& \pi e'$ consists of 13 strings

which resemble the Allen($p,q$) language above. Three of these strings represent overlap of some form or another and the other ten are precedence relations. Thus we have shown that it is possible to extract the Allen interval relations from a language of snapshots using superposition and $\pi$. Furthermore this shows that finite-state temporality represents events and their temporal relations accurately as Russell-Wiener-Kamp event structures does.

### 3.9 Circumscribe

The definition for the circumscribe operations is given as

$$E_{\pm} \overset{\text{def}}{=} E \cup \{\text{pre}(e) \mid e \in E\} \cup \{\text{post}(e) \mid e \in E\}$$

The set of events $E$ is extended to $E_{\pm}$, thus enriching $\text{Pow}(E)$ to $\text{Pow}(E_{\pm})$.

becomes

$$e \rightarrow e,e'$

becomes

$$\text{pre}(e),\text{pre}(e') \rightarrow e,\text{pre}(e') \rightarrow e,e',\text{post}(e)$$
Unfortunately there was not enough time to implement this, it was suggested that \texttt{pisuperpose} be used on the following

\[
\text{pre}(e) (\epsilon | \emptyset) [\epsilon | \emptyset] \text{post}(e) \quad \&_{\pi} \quad \text{pre}(e') (\epsilon | \emptyset) [\epsilon | \emptyset] \text{post}(e')
\]

to see what disjunct of Allen’s simple interval relations would be returned. This would be an interesting next step for the project.

### 3.10 Output to xfst and dot

\textit{Graphviz} is an open source graph visualisation software, which can be used to render graphs in many formats including .pdf, .ps, .jpg, and more from plaintext files. Dot is a plaintext description language that comes as part of the graphviz package and is used best for ‘hierarchical’ graphs. Other simple languages for rendering graphs come with graphviz, however Dot was chosen for this project. It was used to generate the FSA images seen in this report.

Eventually the FSAs became too large for the Prolog list representation to be readable. It was decided to write a predicate to output the FSAs returned from the predicates above in xfst (Xerox Finite State Tool) and dot formats to make the finite-state automata easily viewable. In fact we wound up with two predicates (for FSTs and FSMs).

Creating this predicate involved writing the dot template as well as the FSM to a file with a .dot extension. There are two predicates for this, one for FSTs and one for FSMs, which run through the transitions and print each of their components in the format for transitions in dot. They also print each of the final states, stating that they should appear as double circles.

Outputting the FSAs to xfst involved similar steps of rewriting them to an xfst file in the correct format.
3.11 Problems encountered

Although Prolog can lead to vastly complex code, the approach to representing FSAs as lists turned out to be quite reasonable. However certain parts of the implementation didn’t go entirely to plan. Most of the problems that were encountered were resolved, although some less easily than others. If a certain approach to coding the relations didn’t work, it resulted in prolog either giving an “out of global stack” error or the code never returning an answer. Most problems of this nature were fixed using the prolog ‘trace’ and ‘debug’ modes.

Although even when the code did work, with some approaches it was difficult to tell whether it covered all possibilities. For example, the `unpad` and `bc` relations were originally designed to take a finite-state machine and return another finite-state machine. This code became difficult to perfect as there are many types of finite-state machines that would need to be compressed. Writing code that would cover the cases where an FSM would loop for multiple inputs of the same symbol, where an FSM would continue to change state for multiple identical symbols and moreover would become unwieldly. It was eventually decided to simply create an FST that would aid in the block-compression of a given FSM. The same scenario existed for the `unpad` relation, many different types of FSMS existed that would need unpaddings that eventually I decided to use an approach involving composition with an FST.

The decision to used FSTs for `unpad` and `bc` was also influenced by the fact that much time would have to be dedicated to converting the non-deterministic finite-state machines returned to deterministic ones.

However composition between FSTs and FSMS returned many unwanted transitions and states, simply a by-product of the operation picking technically valid but useless states. For example many of the “dead” transitions, as they were called in this project, were basically identical to the more correct transitions. After the removal of these transitions, the dot graph that was output was reduced greatly to just the transitions necessary. As was stated in section 3.8, two predicates `removeDeadTransFST/2` and `removeDeadTransFSM/2` were written and its implementation was discussed there.
Another predicate was written to help with a different problem arising from unwieldy FSTs from compositions. `removeEpsilon /2` was used to counter the problem that occurred during the running of the `pi` predicate. In this predicate the block-compressing and unpadding FSTs are composed
and the FST returned is output by $p_i$. This results in some epsilon transitions which increase the size of the FSM retrieved when the FST is eventually composed with it. This simple predicate picks out the transitions for which the output label is not 0 (i.e. not epsilon) and returns it as the new list of transitions.

There were problems encountered elsewhere in the implementation, in the case of subsumption the `subsume` predicate originally made incorrect assumptions about the handling of strings. In transitions it would read several symbols at once rather than one at a time. This was later rectified.
Chapter 4

Analysis

4.1 Future Work and improvements

In the near future, the operation circumscribe, which adds pre and post states to sequences of snapshots, enriching the alphabet to be dealt with, should be implemented. There was unfortunately not enough time in this project for it.

For quite a few FSAs created in the code, it would be preferable to have a better representation for □, epsilon and 0 to make it easier to distinguish between them. At the moment, in the unpadfst predicate the boxes to be unpadded are represented in the prolog code as sq, although sufficient for the time being, it is imperfect. Ideally, it would be valuable to investigate possible unicode characters and how they are written in Prolog. It also poses a problem to any input string that has ‘sq’ as a single character.

In terms of compose, it would be valuable to add some code to the current predicate to given the states in the returned finite-state automaton better names. For example, two FSAs could have transitions \([q0,a,q1]\) and \([r0,b,r1]\) respectively. When these are composed, the start states for each transition are placed in the pair \([q0,r0]\), which acts as on state for the returned FSA. For only one composition, this is fine. However after multiple compositions the state could look like: \([[[q0,r0],s0],t0]\) which is ugly and difficult to read, especially when the FSA is produced as a dot graph. It would be useful to rewrite the compose predicate so it renames states after composition. There is not yet a suitable alternative naming method other
than the concatenation of the two state names.

There are two predicates for outputting the FSAs to Dot format, which I believe are not entirely user-friendly. In future it would be beneficial to merge them and have a simple conditional based on the transitions given deciding whether the input is an FSM or an FST.

It would also be advantageous to make more use of xfst in future versions. Although at the moment the Prolog FSMs can be output to a xfst-friendly format and thus xfst operations (compose, for example) can be used on them, the implementation is Prolog was more often used. xfst may aid in creating a more advanced suite in future.

4.2 Conclusion

It is indeed possible to investigate the temporal structure of events in natural language semantics when we characterise the events as strings in regular languages. This project shows that finite-state temporality can be used to reason about events and their relations in time by implementing some theoretical aspects as finite-state machines in Prolog. The background theory was explained, as were the relations on event structures and their implementation. We took snapshots consisting of a list of formulae described to hold at some point in time as symbols in the alphabet of the finite-state automata. Strings of these snapshots represented events, and could be manipulated using regular expressions, leading to the conclusion that the occurrence of events could be described by a regular language. A discussion of how the implementation followed on from this theory was given and the problem which occurred during its writing were addressed. As we set out to show, the theoretical aspects of finite-state temporality can be soundly implemented in Prolog.

Although there is the possibility to go much further in depth with the application developed in this project, a lot has been accomplished. In future versions of this work I hope that more operations and aspects of finite-state temporality can be implemented. A practical execution of finite-state automata was developed in Prolog and was used to allow reasoning about events.
A usable setup completely written in Prolog has now been created. Using the predicates defined in this project allow future users to mix the current operations or define more complex ones based on what has been established. The potential to create a system that is fully capable of reasoning about events is there.
Appendices
Appendix A

Code Listings

A.1 Block Compression

\[
\text{bcfst}(\Phi, [\text{Start}, \text{Trans}, \text{Final}]):- \\
\quad \text{Start} = q_0, \\
\quad \text{Final} = [q_0|\Phi], \\
\quad \text{findall}([q_0, X, X, X], \text{member}(X, \Phi), L_1), \\
\quad \text{findall}([X, Y, Y, Y], (\text{member}(X, \Phi), \text{member}(Y, \Phi), Y \neq X), L_2), \\
\quad \text{findall}([X, X, 0, X], \text{member}(X, \Phi), L_3), \\
\quad \text{append}(L_3, L_2, L), \\
\quad \text{append}(L, L_1, \text{Trans}).
\]

A.2 Restriction

\[
\% \text{restrict}(\text{Alphabet}, X, \text{FST}, \text{Output}):- \text{restrict}(\text{Alphabet}, X, [\text{Start}, \text{Trans}, \text{Final}], \text{Out}):- \text{res}(\text{Start}, \text{Trans}, \text{Final}, \text{Out}, X, \text{Alphabet}).
\]

\[
\text{res}(\text{CurState}, _, \text{Final}, _, _, []):- \\
\quad \text{member}(\text{CurState}, \text{Final}),!.
\]
res(CurState, Trans, Final, Out, X, [H|T]):- 
    member([CurState, H, H, CurState], Trans), 
    (member(H,X) -> Out=[H|Out2]; Out=Out2), 
    res(CurState, Trans, Final, Out2, X, T).

A.3 Superposition

% superpose(+FSM1, +FSM2, ?FSM3)
% fsm1 - [q0, ilabel, q1]
% fsm2 - [r0, jlabel, r1]
% fsm3 - [[q0, r0], [ilabel, jlabel], [q1,r1]]

% Example query and response from the prolog interpreter
% ?- superpose([[[q0],[[q0,[a,b,c],q1]],[q1]],[[r0],[[r0,[x,y,z],r1]],[r1]],FSM3).
% FSM3 = [[[q0,r0]], [[[q0,r0], [a,b,c,x,y,z], [q1, r1]]], [[q1,r1]]].

superpose([Start1, Trans1, Final1], [Start2, Trans2, Final2], [Start3, 
      Trans3, Final3]):-
% Start3 = [a,b] |a in S1, b in S2
findall([A,B], (member(A,Start1), member(B,Start2)), Start3),
findall([C,D], (member(C,Final1), member(D,Final2)), Final3),
% findall: union the labels of Trans1 and Trans2
findall([ [Q1,R1],ULabels,[Q2,R2] ], (member([Q1,L1,Q2],Trans1),
    member([R1,L2,R2],Trans2),
    setof(X,(member(X,L1);member(X,L2)),ULabels)), Trans3).

A.4 Subsume

% two strings of the same length
% subsumeString(Str1,Str2):-
subsumeString([],[]).
subsumeString([H1|T1],[H2|T2]):-
    subset(H1,H2),

subsumeString(T1,T2).

subset(X,Y):-forall(member(A,Y), member(A,X)).

%subsume(+Phi, +FSM1, ?FSM2)
subsume(Phi, [S1,T1,F1], [S1,T2,F1]):-
% find a string in Phi that subsumes a string in FSM1/L1.
% this string from Phi is in the pierce product
  findall([0, L1, L, 0],(member([_,L1,_],T1), member(L, Phi),
    subset(L,L1)), T2).

A.5 Constrain

constrain([S1,Trans1,F1],[S2,Trans2,F2],[S3,Trans3,F3]):-
% subsume both
  (forall((member([_,L3,_],Trans3),member([_,L1,_],Trans1)),subsumeString(L1,L3)),
    forall((member([_,L3,_],Trans3),member([_,L2,_],Trans2)),subsumeString(L2,L3))
  );

% or subsume just L' (not L)
  (not(foreall((member([_,L3,_],Trans3),member([_,L1,_],Trans1)),
    subsumeString(L1,L3))
  ).
subsumeString([],[]).
subsumeString([H1|T1],[H2|T2]):-
  subset(H1,H2),
  subsumeString(T1,T2).
subset(X,Y):-forall(member(A,Y), member(A,X)).
A.6 Unpad

unpadfst(Phi, [Start, Trans, Final]) :-
    Start = q0,
    Final = [q1,q2],
    findall([q0, sq, 0, q0], member(sq,Phi), Tmp1),
    findall([q0, A, A, q1], (member(A,Phi),A \= sq), Tmp2),
    findall([q1, A, A, q1], (member(A,Phi),A \= sq), Tmp3),
    findall([q1, sq, 0, q2], member(sq,Phi), Tmp4),
    findall([q2, sq, 0, q2], member(sq,Phi), Tmp5),
    append([Tmp1,Tmp2,Tmp3,Tmp4,Tmp5],Trans).

A.7 Pisuperpose

pisuperpose(Phi, FSM1, FSM2, PiSupFSM):-
    pi(Phi,FST),
    composeFSM(FSM1,FST,TempCFSM1),
    removeDeadTransFSM(TempCFSM1, CFSM1),
    composeFSM(FSM2,FST,TempCFSM2),
    removeDeadTransFSM(TempCFSM2, CFSM2),
    unpi(Phi, UFST),
    composeFSM(CFSM1,UFST, TempC2FSM1),
    removeDeadTransFSM(TempC2FSM1, C2FSM1),
    composeFSM(CFSM2,UFST, TempC2FSM2),
    removeDeadTransFSM(TempC2FSM2, C2FSM2),
    superpose(C2FSM1, C2FSM2, TempSupFSM),
    composeFSM(TempSupFSM,FST, PiSupFSM2).

pi(Phi,FST):-
% block compress
    bcfst(Phi, BCFST),
% unpad
    unpadfst(Phi, UFST),
    composeFST(BCFST, UFST, TempFST),
    removeEpsilon(TempFST,TempFST2),

44
removeDeadTransFST(TempFST2, FST).

% unpi
% str -> 0*e+0*
unpi(Phi, FST):-
    unbcfst(Phi, BCFST),
    repadfst(BCFST,FST).

% unbc(FSM1,FSM2):- % str = abc -> a+b+c+
unbcfst(Phi, [Start,Trans,Final]) :-
    Start = q0,
    Final = [q0|Phi],
    findall([q0,X,X,X], member(X, Phi), L1),
    findall([X,Y,Y,Y], (member(X, Phi), member(Y, Phi), Y \= X), L2),
    findall([X,X,X,X], member(X, Phi), L3),
    append(L3, L2, L),
    append(L, L1, Trans).

% add box* to the beginning and end of the string
repadfst([Start,Trans,Final],[Start,NewTrans2,Final]) :-
    append(Trans, [[Start,0,Start]], NewTrans1),
    findall([R,0,R], member(R,Final), Temp2),
    append(NewTrans1,Temp2,NewTrans2).

A.8 Dead Transitions and Remove Epsilon

removeDeadTransFSM([S,OrigTrans,F], [S,NewTrans,NewF]) :-
    setof([CS,L,NS],
        Q `Label` ( member([CS,L,NS], OrigTrans),
        (member([Q,Label,CS],OrigTrans);CS=S) ), NewTrans),
        findall(Q, ( member(Q,F),
removeDeadTransFST([S,OrigTrans,F], [S,NewTrans,NewF]):-
    setof([CS,L,L1,NS],
        Q `Label`Label1` ( member([CS,L,L1,NS], OrigTrans),
        (member([[Q,Label,Label1,CS],OrigTrans];CS=S) ), NewTrans),
    findall(Q, ( member(Q,F),
        (member([CS,L,L1,Q], NewTrans);member(Q,S)) ), NewF).

removeEpsilon([S,OrigTrans,F], [S, Trans,F]):-
    findall([CS,L,L1,NS], ( member([CS,L,L1,NS],OrigTrans),L1 \=0),Trans).

A.9 Prolog to xfst

xfstFSM(Filename, [_,Trans,F]):-

    write("network("), write(Name), write(".") ), nl, write_arc(Trans),
    write("final(%w)",F).
write_arc([]).
write_arc([CS,L,NS|Tail]):-
    write("arc(superpose,"),
    write(CS),
    write("", "),
    write(NS),
    write("", "),
    write(L),
    write(""") ),
    nl,
    write_arc(Tail).

A.10 Prolog to dot

dotFSM(Filename,[__, Trans, Final]):-
tell(Filename),
% write the template
write('digraph FSM { \n rankdir=LR; \n graph [splines=true] \n node [shape=doublecircle]; ')
% Final states of the FSM have double circles
forall(member(Q,Final), (write(""),write(Q),write(""),write(""))),
write('; \n node [shape=circle]\n'),
% write the FSM
forall( member([CS,L,NS],Trans), ( tab(4), writef('"%w" -> ",label=%w;/n', [CS, NS, L]) ) ),
write('}'),nl,
told.

dotFST(Filename,[_, Trans, Final]):-
tell(Filename),
% write the template
write('digraph FSM { \n rankdir=LR; \n graph [splines=true] \n node [shape=doublecircle]; ')
% Accepting states have double circles
forall(member(Q,Final), (write(""),write(Q),write(""),write(""))),
write('; \n node [shape=circle]\n'),
% write the FST
forall( member([CS,L,NS],Trans), ( tab(4), writef('"%w" -> ",label=%w;/n', [CS, NS, L]) ) ),
write('}'),nl,
told.

47
Bibliography


50