Towards Finite-State Discourse Representation Theory

Jonathan Cummins
BA(Mod.) Computer Science, Linguistics and a Language
Final Year Project 2010

Supervisor: Dr. Tim Fernando

April 6, 2010
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 6, 2010

Jonathan Cummins
Permission to Lend

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

______________________________________  April 6, 2010
Jonathan Cummins
Acknowledgements

Firstly, I would like to thank my supervisor Tim Fernando for all his help and advice in the preparation of this project, and for agreeing to supervise it in the first place and for giving up so much time for our meetings.

I would like to thank my family and Aisling for their support during this and my previous years of study, and for putting up with my late nights throughout.

I also have to thank my classmates for putting up with me for this year, and for all of their advice and support.

Finally, I would like to thank Carl Vogel for all his advice, classes, inspiration and guidance throughout the 4 years on the course.
Contents

List of Figures 5

1 Introduction 1
   1.1 The Problem in Context ......................... 1
   1.2 Aims ........................................ 2
   1.3 The Layout .................................. 3

2 Formal Language Theory 4
   2.1 Regular Languages and Regular Expressions ....... 4
   2.2 Finite State Machines .......................... 7
   2.3 Finite State Transducers ....................... 9
   2.4 An Automata Representation with a set alphabet ... 11
   2.5 Where are we at? ............................... 13

3 Finite-State Temporality 14
   3.1 Time as a snapshot ............................ 14
   3.2 Events and States ............................. 16
   3.3 Grammatical Tense ............................ 18
   3.4 Aspect ....................................... 19
   3.5 Reichenbach’s E, R and S ....................... 21
   3.6 Sentences as Snapshot Strings and Superposition .. 23
   3.7 Block Reduction ............................... 26
   3.8 Unpadding .................................... 27
   3.9 Allen Relations and Constraints ................ 28
4 Implementation using the Xerox Finite-State Toolkit 30
  4.1 Xerox Finite-State Toolkit ......................................... 30
  4.2 How does XFST work? .................................................. 31
      4.2.1 The Interface .................................................. 32
      4.2.2 The Stack ...................................................... 32
      4.2.3 Using SWI-Prolog with XFST ................................... 35
  4.3 XFST scripting ........................................................... 37
  4.4 An XFST representation of the Snapshot ............................. 38
  4.5 Coding Superposition .................................................. 40
  4.6 Coding Blockreduction ................................................ 42
  4.7 Coding Unpad ........................................................... 45
  4.8 Combining Superpose, Blockreduction and Unpad .................... 47
  4.9 Deriving the Allen Relations ........................................ 48

5 Further Work and Conclusions 50
  5.1 What has been seen so far? .......................................... 50
  5.2 Coding Improvements .................................................. 50
  5.3 Discourse Representation Structures as Languages ............... 51
  5.4 Constraints and Inference ............................................ 51
  5.5 Was this project a success? ......................................... 52

A Complete Code Listings 55
  A.1 Superposition- compute1.pl ......................................... 55
  A.2 prolognormalis.pl ................................................... 57
  A.3 blockreduction.pl .................................................. 58
  A.4 br2xfst.pl ............................................................. 61
  A.5 unpad.pl ............................................................... 63
  A.6 unp.pl ................................................................. 65
List of Figures

2.1 A FSM with a set alphabet .................................. 12
3.1 The Block Reduction transducer ............................... 27
3.2 The Allen Relations .......................................... 29
4.1 An Screenshot of XFST ....................................... 32
4.2 Regular Expression Compiling with XFST .................... 33
4.3 Non-Deterministic,Non-Minimal,Non-pruned FSA ........... 35
4.4 Reading and Writing Prolog .................................. 36
4.5 Deterministic, Minimal and Pruned Version of 4.3 ......... 37
4.6 An XFST script .................................................. 38
4.7 A function mapping fluents to their character representations 39
4.8 Reading in a snapshot REGEX in XFST ....................... 40
4.9 The Block Reduction transducer ............................... 43
4.10 Reading the Languages onto the Stack ....................... 48
4.11 Printing the 13 Allen Relations with XFST ................. 49
Abstract

Language provides us with descriptions of the states of the world and events of the world. Reasoning about states and the events that change them is a primary concern in the field of Knowledge Representation and Reasoning, a subfield of Artificial Intelligence. Indeed events and states do have their place in Kamp’s Discourse Representation Theory. This project shows how the events and states of a sentence can be reasoned with in a finite state setting. This will be shown to be a fresh, flexible and alternative to the popular methods in use today.

The Xerox Finite State Toolkit (XFST) and Prolog will be used to develop the finite state networks that are used for reasoning with events and states. To illustrate what states and events are precisely they will be discussed in detail assist the readers understanding some more.

Events and states in this project will be represented in a form known widely as the snapshot. The aim of the project is to show that the linguistic phenomena of tense and aspect can be reasoned with using the Xerox Finite State Toolkit.
Chapter 1

Introduction

1.1 The Problem in Context

Humans can speak, listen, read and write. If two people are in conversation they do a huge amount of language processing every second. In the 20th century Chomsky among others began to investigate how humans actually process language. He believes that humans are born with an inbuilt language faculty and he has tried to come up with a scientific explanation of how this works. There are many explanations out there but they are all far from perfect. However, it is not essential that we know precisely how humans process language. As stated in [8] the goal of Artificial Intelligence is to create working reasoning systems. When we are talking about language processing it does not matter if the system reasons the same way as a human. What is important is that two processes yield the same result.

All people who can speak and understand language have at least a subconscious understanding of events and states. It does not need to be clear to a computer programmer how the human being does this. The programmer just wants the AI to produce the same results as the human every time. In this project we are going to try and reason with the events and states of natural language using finite state tools. In the human processing of sentences it is highly unlikely that finite-state machines suddenly appear in the humans brain but nevertheless an attempt to carry out the same reasoning
using finite-state techniques is the long term goal of work in this field.

**1.2 Aims**

Intelligent computational reasoning about events and states using finite-state tools is a rather new phenomenon. In the fields of Computational Linguistics and Natural Language Processing a lot of work has been done already, but most of this work has been undertaken in subfields such as computational morphology and syntax. The use of finite state machines to represent the temporality of sentences is very recent. According to [8] ".. the recent progress made in the development of compilers like the Xerox Finite State Toolkit(XFST) that 'make practical what was only a theoretical possibility a few years ago [1]\". In this project

The principal aim of this project is to show that the semantic notion of events and states can dealt with in terms of regular languages and hence finite-state machines. As well as the aim is to implement an application that works and demonstrates the central claims of the theory of finite-state temporality as proposed in papers such as [5], [2], [4] and [3]

A secondary aim is the desire to see just how far this finite-state approach to handling events can be pushed. From the outset it is well known that there are limitations of using finite-state methods. The aim is to see where these limits exactly lie and to see what elements of semantics can be represented this way.

Another aim would be to simply show off the elegance and power of finite state machines and what can be achieved with the Xerox Final State Toolkit. Finite-State machines have been used for many applications especially for hardware applications and low level programming. To show they are capable of carrying out high level tasks such as the tasks found in semantics would take the applicability of finite-state machines to places where they have never gone before.
1.3 The Layout

The layout of this project is as follows. In the next chapter a very brief introduction to formal language theory is given for the reader who is unfamiliar with these tools. In chapter 3 the ideas central to finite-state temporality are discussed in detail. Chapter 4 contains the more innovative part of the project. It demonstrates an application of the theory outline in chapter 2 and chapter 3 and this practical application is the first implementation of the tools needed to ensure finite-state temporality is successful. Chapter 5 discusses and evaluates the current work that is completed and discusses where the project is to go from here and concludes. The appendices contain the source code of the implementation.
Chapter 2

Formal Language Theory

In this chapter the relevant aspects of formal language theory are outlined. As this project makes use of many different finite-state methods it is worthwhile to have an understanding of the basics of formal language theory and automata. To begin Regular Languages are discussed, then finite-state machines, followed by Transducers and to conclude a formal specification of what a finite-state machine is to be viewed as in this project.

2.1 Regular Languages and Regular Expressions

Definition According to [10] the set of regular expressions over an alphabet \( \Sigma \) is recursively defined as follows.

1. 0 is a regular expression. (Also denoted \( \emptyset \)).
2. 1 is a regular expression. (Also denoted \( \epsilon \)).
3. \( \forall \sigma \in \Sigma, \sigma \) is a regular expression.
4. If \( \alpha \) and \( \beta \) are regular expressions then it follows that
   
   - \( \alpha^* \) is a regular expression.
   - \( \alpha \cup \beta \) is a regular expression.
• $\alpha \beta$ is a regular expression.

5. Every regular expression is constructed from the above rules

One of the virtues of regular expressions is the fact that they can represent strings of infinite length in a finite meta expression. For example consider the regular expression $\alpha^*$. This regular expression is a notation that can be used with and matches with strings such as $\alpha, \alpha \alpha, \ldots, \alpha$ and $\epsilon$. In the definition above the $\cup$ symbol should be viewed as an exclusive or. For example the regular expression $abc(a \cup b)c$ matches with two strings. They are $abcac$ and $abcbc$.

The $\alpha \beta$ should be read simply as $\alpha$ followed by $\beta$. This is also known as concatenation.

**Definition** A regular language $L(\gamma)$ is the set of all strings that are generated by some regular expression $\gamma$.

A regular language $L$ can be finite (eg $L(a + b + c) = \{a, b, c\}$) or infinite (eg $L(a^*) = \{a^0, a^1, \ldots, a^n\}$). $a^7$ can be thought of as the string consisting of $a$ concatenated seven times $aaaaaaa$. From the above definition we can conclude that regular expressions are meta expressions for describing the form of the strings in some set or regular language $L$.

The set of regular languages over an alphabet $\Sigma$ is the smallest set which contains the $\emptyset$ and $\{\sigma\}, \forall \sigma \in \Sigma$ and is closed under the following regular operations

1. Concatenation

2. Union $\cup$

3. Kleene Star $^*$

4. Complementation

5. Set Difference

6. Intersection
What this means is that given any regular language or languages and performing the above operations the result is another regular language for example

- Under Concatenation $L(a \cup b \cup c) L(e \cup f) = \{ae, af, be, bf, ce, cf\}$

- Under Union $L(a) \cup L(b) = \{a, b\}$

- Under Kleene Star $L(a)^* = \{\epsilon, a, aa, aaaa, aaaaa, ..., a^n\}$ $\forall n \text{Integer}(n)$

- Under Complementation If $\Sigma = \{a, b\}$ and $L(a)$ then $\neg L(a) = \{b\}$

- Under Difference $L(a + b + c + d) - L(a + b + e + f) = \{c, d\}$

- Under Intersection $L(a + b) \cap L(b + c) = \{b\}$

Some further properties of regular expressions are the following identities (again taken from [10]).

1. $1\alpha \equiv \alpha 1 \equiv \alpha$
2. $\emptyset 1 \equiv 1\emptyset \equiv \emptyset$
3. $\alpha \cup \beta \equiv \beta \cup \alpha$
4. $(\alpha \cup \beta)\gamma \equiv \alpha\gamma \cup \beta\gamma$ and $\gamma(\alpha \cup \beta) \equiv \gamma\alpha \cup \gamma\beta$
5. $\alpha^* \equiv \emptyset^* \cup \alpha\alpha^*$
6. $\emptyset^*\alpha \equiv \alpha$
7. $\alpha^+ \equiv \alpha^*\alpha$

**Definition** Two regular expressions $\alpha$ and $\beta$ are said to be equivalent if $L(\alpha) \equiv L(\beta)$

---

It should be noted that the $\cup$ in these examples is applied at the level of the regular expression and not at that of the regular language. This subtle semantic difference often gives rise to confusion.
Let $\alpha = ab(c + g)$ then $(\alpha) = \{abc, abg\}$ Let $\beta = ab(g + a)$ then $(\beta) = \{abc, abg\}$ Then $L(\alpha) \equiv L(\beta)$

To conclude this section it is worth noting that the main virtue of regular expressions is that infinite languages can be expressed using finite meta-expressions and as will be seen in the next section regular expressions come with all the benefits of using finite state languages.

2.2 Finite State Machines

Central to this project are finite-state machines therefore is worthwhile to step aside for a moment in order to understand what these machines are.

Definition A deterministic finite-state machine $A$ is defined as a quintuple $A = (Q, \Sigma, f, s, F)$ where

1. $Q$ is a set of states
2. $\Sigma$ is a finite alphabet
3. $f : Q \times \Sigma \rightarrow Q$ is a transition function
4. $s \in Q$ is a designated start state.
5. $F \subseteq Q$ is a designated set of final states.

A deterministic finite-state machine (DFSM) is a mathematical model and are a more restricted version of the Turing Machine. Every finite-state machine accepts a set of strings known as a language. Given a deterministic finite-state machine and an input string we can decide whether or not the string is a part of the finite-state machine language. One of the virtues of finite-state machines is that this can be decided in linear time. In fact this is the worst case scenario. So given a string of length $n$ the complexity involved in deciding whether or not the string is accepted in Big Oh notation is $O(n)$. It should be noted that this only applies to deterministic finite-state machines.

There are also what are known as non deterministic finite-state machines. It is often the case that the construction of a deterministic finite-state machine is difficult. Normally a non-deterministic finite-state machine can be
constructed easier than a deterministic machine. It can be proven that for every non-deterministic finite state machine, there exists a deterministic finite-state machine that accepts the same language. This is proven in [10] A non deterministic finite-state machine is defined as follows in [10].

**Definition** Given a quintuple \( A = (Q, \Sigma, \Delta, s, F) \):

1. \( Q \) is the same as for deterministic FSMs
2. \( \Sigma \) is the same as for deterministic FSMs
3. \( \Delta \subseteq Q \times \Sigma^* \times Q \) is a relation
4. \( s \) is the start state
5. \( F \) is the set of final states

\( \Delta \) is called the transition relation. Normally when working with finite state machines if the machine is quite complex, one would try to construct an NFSM(non deterministic finite-state machine) and then determinise it using the determinisation algorithm. However the drawback of this approach is that the determinisation step is expensive. Another point to note is that sometimes a deterministic machine needs pruning. Sometimes there are states that cannot be reached from the starting state and sometimes their are two states which cannot be told apart from each other. That is they are indistinguishable. In [10] the definition of indistinguishable states is given. Two states \( p, q \in Q \) are \( k \)-indistinguishable when \( (k \geq 0) \) and \( \forall w \in \Sigma^* \) when \( |w| \leq k \) iff \( f(p, w) \in F \Leftrightarrow f(q, w) \in F \). Two states are indistinguishable if they are \( k \)-indistinguishable \( \forall k \). After determinisation all FSM’s can be minimized to create smaller more efficient finite-state machines that do not have indistinguishable states or states unreachable from the starting state.

According to [10] every regular language is recognisable by a finite-state machine. What this means is means is that there is an algorithm which given a regular expression can construct an FSM which accepts the same language. Two well known methods of doing this are the Thompson Algorithm and Berri Setti Algorithm. These are both found in [10]. Furthermore given
a finite state machine $A$ accepting the language $L(A)$ it is also possible to construct a regular expression whose language is $L(A)$. Given these facts one can conclude that the set of regular languages and the set of finite-state machine languages are the same set$^2$. The fact that regular languages are equal to automata languages is what paves the way for this project because as will be seen in the next chapter the use of regular expressions to represent events and carry out operations on the finite-state machines accepting these languages is desired.

To conclude this section let us recap some of the properties which are useful when working with finite state machines

- For every NFSM there is a DFSM
- For every regular language there is an FSM that accepts that language and vice versa
- Sometimes a DFSM can be minimized
- DFSMs accept words of length $n$ in linear time (Big Oh notation: $O(n)$)

### 2.3 Finite State Transducers

As will be seen later on Finite State Transducers are used in the implementation part of this project. Therefore in this section an elementary introduction to these machines is given in this section. To begin with we define finite state transducers as is done in [6].

**Definition** A finite state transducer is a 7-tuple $A = \{Q, \Sigma, \Delta, q_0, F, \delta, \sigma\}$ where

1. $Q$ is a finite set of states.
2. $\Sigma$ is a finite alphabet of complex symbols. Each complex symbol is an input output pair $i:o$. $\Sigma \subseteq IO$.

$^2$For more information on this see the Kleene Theorem
3. $q_0 \in Q$ is the start state.

4. $F \subseteq Q$ is the set of finite states.

5. $\sigma(q, i : o)$ is the transition function between the states. Given a state $q \in Q$ and the complex symbol $i : o \in \Sigma$, $\sigma(q, i : o)$ returns a new state $q' \in Q$.

Finite-state transducers can be used for recognising or deciding whether a string pair is accepted by the transducer. FSTs can also be used to as generators. This means the transducer can output a pair of strings of the language. Sometimes they can be used as translators. That is the transducer reads a string and outputs another string. Just like finite state machines languages are regular languages, finite-state transducers are regular relations and can be used to computer relations between sets. In this project we will use the Finite State Transducers as a way of translating one finite state machine into another in what will be called unpadding.

Just like finite state machines transducers are closed under the following operations given two transducers $A$ and $B$.

1. **Union** $\exists X$ Transducer $X = A \cup B$ such that $x[A \cup B]y \iff x[A]y \lor x[B]y$

2. **Concatenation** $\exists X$ Transducer $X = A \cdot B$ such that $wx[A \cdot B]yz \iff w[A]y \land x[B]z$

3. **Intersection** Given Transducer $T = A \cap B$, $x[T]y \iff x[A]y \land x[B]y$

4. **Kleene Closure** $A^*$ has the properties (1)$\epsilon[A^*] \epsilon$ (2)$w[T^*]y \land x[T]y \Rightarrow wx[T^*]yz$ and $x[T^*]y$ does not hold unless it is a special case of (1) or (2)

5. **Composition** If $\Sigma$ and $\Delta$ are the alphabet of $A$ and $\Delta$ and $\Gamma$ the alphabet of $B$ then the composition of the two $A \otimes B$ has the alphabets $\Sigma$ and $\Gamma$ such that $x[A \otimes B]y \iff \exists z \in \Delta^* \land x[A]z \land x[B]z$

Another property of transducers that will be made use of in this project is the projection function. There are two of these functions for every transducer.
The first function returns an automata which accepts the upper or input language and the second returns an automata accepting the lower or output language. More formally they can be specified in the following manner:

**Definition** Let $T$ be a finite-state transducer let $A_{input}$ and $A_{output}$ be two finite state machines. The input projection function is defined as $\mu_{input} : T \rightarrow A_{input}$ and the output projection function is defined as $\mu_{output} : T \rightarrow A_{output}$. Where $A_{input}$ recognises the upper language of the Transducer and $A_{output}$ recognises the lower language of the Transducer.

One of the problems encountered when using transducers is that unlike Automata every non-deterministic transducer cannot be determinised. This means when when deciding whether or not a String is accepted by a transducer, techniques such as backtracking, forward tracking and parallel tracking are resorted to. Using these methods deciding whether or not a string is accepted by a finite state transducer is possible although expensive.

### 2.4 An Automata Representation with a set alphabet

In the next chapter the notion of a snapshot is introduced. Without explaining what these snapshots are until then, a description of the automata that are used in this project is given in this section.

The finite-state machines used in this project are no different to the finite-state machines encountered above. In this project we want to think the alphabet of the finite state machines as a set of sets. That is each element of the alphabet is a set. More particularly these sets are sets of fluents. Fluents in Artificial Intelligence are simple conditions that vary over time. For example a fluent $RAIN$ can be used to say that it is raining and $not(RAIN)$ is the negation of that fluent. A set of fluents can contain as many or as few fluents as necessary in our representation.

Now let us ground this with an example. Consider how the concept "raining from dawn to dusk" could be represented using a finite state machine. To
do so we need to represent three different ideas. The first is raining which we use \textit{RAIN}. The second is the time of day dawn which can be represented by the fluent \textit{DAWN} and the third fluent needed is \textit{DUSK} which can be understood to represent dusk. Consider the following deterministic finite state machine. 

\[ A = (Q, \Sigma, \sigma, s, F) \] 

- \( Q = \{0, 1, 2, 3\} \)
- \( \Sigma = \{\{\text{RAIN, DAWN}\}, \{\text{RAIN, DUSK}\}, \{\text{RAIN}\}\} \)
- \( \sigma(0, \{\text{RAIN, DAWN}\}) \to 1, \sigma(1, \{\}\) \to 2, 
  \( \sigma(2, \{\}) \to 2, \sigma(2, \{\text{RAIN, DUSK}\}) \to 3 \)
- \( s = 0 \)
- \( F = \{3\} \)

A diagram of the finite state machine which accepts strings of this language can be seen below. A diagram of the finite-state machine which accepts strings of this language can be seen in Figure 2.1.

![ FSM Diagram ]

Figure 2.1: A FSM with a set alphabet

The following is the regular expression which would be equivalent to the finite state machine defined above.

\[ \text{DAWN,RAIN} \text{RAIN}^+ \text{DUSK,RAIN} \]  \hspace{1cm} (2.1)
2.5 Where are we at?

In this chapter the fundamental principles of automata theory have been outlined. Indeed there is much more than could be discussed in this area but such a discussion is out of scope of this project. What is presented above is an aid to the reader and should make the following sections much more transparent. In the next chapter the ideas in this chapter are applied to a linguistic problem.
Chapter 3

Finite-State Temporality

In this chapter the central theoretical ideas of the project are outlined. Here the problem to be solved is presented and much of the research that has already been done in the area are explained and discussed. To begin with, linguistic tense and aspect are explained. After this the relationship between tense and aspect as characterised by Reichenbach’s event time (E), speech time (S) and reference time (R) are outlined. Then this is tied in with finite state methods.

3.1 Time as a snapshot

Time is a fundamental part of sentences and discourses. In sentences there is always a tense which gives an indication of when whatever is being discussed will occur, has occurred or is occurring. In human language there are often instances of time as well as intervals of time. Instances of time are discrete but intervals of time are continuous.

Although in theory time is infinite and can be divided up into units of infinitely smaller duration, in practice this is not the case. The fact that computers cannot be infinitely precise as well the fact that humans are vague means there must be in theory some time $t$ that cannot be broken down any further. This time is what is known as an instance. An instance can be viewed as a snapshot of the world at a given moment much like a photograph. If
a photograph is taken at any time a person can look at the photo and ask questions such as "Does the person have blue eyes or are they wearing a blue jumper?". The answer to these questions is yes or no. Therefore each snapshot contains facts about the world. This is a fundamental aspect of the project. We want to represent sentences consisting of certain facts in a way that makes it relatively simple for a machine to reason about the facts contained in the given sentences. For example consider the world where a man is standing in a garden and it is raining. A snapshot of this world looks like the following:

```
garden, raining
```

Raining and garden are both facts. The fact that these snapshots both appear in the same window means that they are both true at the same time. Anything that appears in a snapshot is true at that point in time. In this case raining represents the fact that it is raining and garden is understood to be the man's location. Anything not included in the snapshot should not be seen as false but non-derivable. For example it is incorrect to say that John is not wearing a jumper in the above the snapshot.

An interval is composed of instances. What this means is that something happens slowly over an arbitrary amount of time. If for example a man is standing in his garden for a year and the smallest unit of time is 1 day. We can decompose this time interval down into a certain 365 snapshots where each snapshot represents a day. If we call the interval a year and indeed that is what it is, then it contains 365 instances each which describes the world or part of the world on that day. Consider again the year 1989 in Germany. If we assume the smallest unit of measuring time is the day this is an instance. We can represent part of the year as the following snapshot:

```
Sunday-January-1 2 ...  Berlin-Wall-Falls  ...  December-31 365
```

This snapshot is just a glimpse of the year 1989 in Germany. It can be viewed of as the interval between 1988 and 1990 and as already said consists of snapshots of each day in Germany in 1989. The snapshots in this interval are chronologically ordered. That means that snapshot one can be contains facts about the world and these facts where true before the facts in snapshot 2.

Now clearly the above snapshot does not contain all the events that hap-
pened in the world in 1989 and indeed this list is enormous. But clearly it would make sense to be able to combine this interval with another interval in order that we have snapshots representing 1989 in Germany and also in, for example, the USA. This is possible and will be explained later on.

### 3.2 Events and States

In this section an account of events and states as they are understood in the project is given. In the next 2 sections the linguistic qualities tense and aspect are described and in the section after that the relationship between the two is pointed out.

In chapter five of [7] an integration of events and states into the DRT framework is shown. It is noted by the author however, that there is extreme difficulty in defining exactly what events and states actually are. Indeed the problem is so complex that it is decided to show the integration using a definition of states and events that is not mathematically or linguistically rigorous. The author argues that rigor and formality are not always essential in natural language semantics as “one of the central tasks of semantics is to articulate the conceptual structures that guide and support our human understanding of the languages we use. If that understanding crucially involves concepts which are to some degree underdetermined, then the semanticist has the task of spelling out precisely how and to what extent the concept is underdetermined; it will not do to substitute a fully determinate concept of one’s own conception for the underdetermined notion that is in actual use”\(^1\).

Kamp describes events as something a certain sentence describes.

\begin{equation}
John\text{ watered the flowers on Friday}
\end{equation}

As such (3.1) is a description of the event of John watering the flowers. Sentences in which some form of action occur can be thought of as events for the sake of our understanding. However if events are defined in terms of actions, then a definition of an action is needed. A car crashing, a plane

\(^1\)Quote from [7]
landing, a drop of water evaporating and bunny dying are all actions.

A state can be considered as denoting a condition. For example if we think of a gun. Imagine this gun can either be loaded or not loaded. When the gun is loaded it is said to be in a "loaded" state. When the gun is not loaded it is in an unloaded state so to speak.

Clearly states can change. Like the laws of physics, in order for the state of something to change an external "force" must act upon it. In linguistics these forces are similar to events. If for example the gun is not loaded in one instance and then is loaded in the next instance an event must occur to change the state of the gun from unloaded to loaded. Here are some example sentence:

\[ \text{The Gun is not loaded} \] (3.2)

\[ \text{John loads the gun} \] (3.3)

\[ \text{The Gun is loaded} \] (3.4)

(3.2) is a stative sentence. (3.3) is sentence describes an event. This event causes the state seen in (3.4). To tie this in with our previous representation of the snapshot, if the above three sentences are true and take place in the same order a string of snapshots representing the above three sentences is:

\[ \text{not loaded} \rightarrow \text{loading} \rightarrow \text{loaded} \] (3.5)

The set of events and the set of states is not crisp but rather fuzzy. This is outlined in [7]. Consider the following sentence:

\[ \text{The man stood on the grass} \] (3.6)

The above sentence highlights clearly the problem with the definition of events and states that Kamp uses. Part of the problem here is ambiguity. Was the man walking along minding his own business when he happened to stand on the grass. Or has the man been standing on the grass waiting for a long time. Is it a fact that the man is still standing on the grass? More so
is this an action or a state. This is not clear.

### 3.3 Grammatical Tense

Grammatical Tense is the quality of the verb which expresses the time at which (or when) the state or action, that the verb denotes, occurs. Contrary to popular belief, there are only two tenses in English. These are the present and past. Consider the following sentences for example:

\[
\text{John eats the pizza.} \quad (3.7)
\]

\[
\text{John ate the pizza.} \quad (3.8)
\]

(3.7) is the simple present tense form and (3.8) is the past tense. These are the only two tenses that need to be represented in the English language. Now consider:

\[
\text{John will eat the pizza.} \quad (3.9)
\]

This sentence appears to be what would be classed as a "future tense sentence". The reason for this is because the infinitival form of eat is used with an auxiliary. If we view auxiliaries as "helper" verbs, then we have an auxiliary verb "will" helping the infinitive "eat" to make the future tense. This is, strictly speaking, not what some linguists define as a future tense construction. If we assume that the tense of the verb is what defines whether or not we are describing an event in the past, present or future tenses, then strictly speaking there has to a specific morphological or syntactic form of the verb. For example in Irish the three verbs meaning "to be" in Irish in the three different tenses are:

\[
\text{Chuaigh}(\text{Past}) \quad (3.10)
\]

\[
\text{Teann}(\text{Present}) \quad (3.11)
\]

\[
\text{Rachaidh}(\text{Future}) \quad (3.12)
\]

Each is a specific form of the infinitive "Teigh". So for the English equivalent "to be" if we were to show it inflected in all possible tenses all we have will
be:

\[ am(Past) \]  \hspace{2cm} (3.13)

\[ are(Present) \]  \hspace{2cm} (3.14)

and nothing for the future tense.

### 3.4 Aspect

Grammatical aspect is the linguistic quality of the sentence which determines the chronological flow of events and states in the said sentence. In this project three different types of aspect are looked at. They are:

1. Simple Aspect
2. Perfect Aspect
3. Progressive Aspect

The simple aspect usually refers to events or states that are current or happening at the moment the tense refers to. For example the sentences:

\[ John \, eats \, pizza \]  \hspace{2cm} (3.15)

\[ John \, ate \, pizza \]  \hspace{2cm} (3.16)

Both of the sentences above are in what is known as the simple aspect. However it is worth noting that they are different tense.

The next sentence type that needs to be looked at is the perfect aspect. Again this can be combined with the two tenses in the English language to give rise to the present perfect and past perfect. Present perfect and past simple are very close in meaning and are often used interchangeably.

\[ John \, has \, eaten \, the \, duck \]  \hspace{2cm} (3.17)

\[ John \, had \, eaten \, the \, duck \]  \hspace{2cm} (3.18)
(3.17) is a present perfect sentence. John began to eat and finished eating his duck at some point before the time at which the sentence was uttered. (3.18) is a past perfect sentence. John began to eat and finished eating his duck by a point that is before the time the sentence is uttered (Past in the Past). Note that (3.18) and (3.16) appear to mean the same thing and are often considered equivalent. In both sentences the food has been eaten before the time at which the sentence is uttered.

The third grammatical aspect is known as the progressive or continuous aspect. Progressive sentences are slightly more complicated. The reason for this is that they omit certain information. According to [5] some progressive sentences can be either intensional or extensional. To clarify, consider the following sentences from [5]

\[
\begin{align*}
\text{Pat crossed the road} & \quad (3.19) \\
\text{Pat was crossing the road} & \quad (3.20) \\
\text{Pat is crossing the road} & \quad (3.21)
\end{align*}
\]

(3.19) is a simple past tense sentence. According to [5] it is extensional. This means that the event started and finished. There was a point in time \(t_1\) and at \(t_1\). Pat was standing at the side of the road at point \(a\) and then Pat crossed the road at \(t_2\) and ended up standing at the opposite side of the road \(b\). The important point to note here is that the event of crossing the road has elapsed and is safely in the past. (3.20) is a past progressive sentence. This sentence is according to [5] intensional. Essentially what this means is that it is never actually known, whether or not Pat manages to make it across the road. There is a point in time \(t_1\) when Pat was standing on one side of the road and a point in time \(t_2\) when Pat started to cross the road. He may or may not have made is across. When the progressive is combined with the present tense as in (3.21) this problem is not so much true. As in the present tense the progressive event is happening now. The event cannot possibly be finished yet. To summarise the main problem with the progressive is that the state of the world after the progressive event elapses is not clear. Progressive
sentences can be viewed describing events that are or were in progress and their conclusions are not clear.

3.5 Reichenbach’s E, R and S

Having described Tense, Aspect, Events and States we are now in a position to relate this to Reichenbach’s event time, reference time and speech time. We also want to show how events can be broken down into regular languages as in [5].

To start with speech time S should be understood to be the time at which the sentence is uttered. This is a unit for real time. If somebody says something at 16:00 on a Sunday that is the speech time. This time can always be thought of as the current time or the now and is the very latest time.

The reference time is the time we are referring to. Consider the following

\[ John \, was \, eating \, fish \, yesterday \] \hspace{1cm} (3.22)

\[ John \, was \, eating \, fish \] \hspace{1cm} (3.23)

In (3.22) the reference time is explicitly given as yesterday. The reference time however must not always be explicitly given as can be seen in (3.23). Here we are referring to some time in the past and this time is the reference time. In fact what should be noticed here is the relationship between the reference time and speech time. This relationship is defined by tense and according to [5] the following relations hold

1. **Present** \( R = S \)

2. **Past** \( R < S \)

3. **Future** \( R > S \)

1 says that in the present the tense speech time is the reference time. 2 says that in the past tense the reference time is before the speech time and 3 says that in the future tense the reference time is after the speech time.
What we can conclude from this is that the tense of a sentence determines the relationship between it’s speech time and reference time.

The event time E is the time at which an event occurs relative to the reference time. Consider the following sentence:

\[
\text{By 2pm the girl had died}
\]  
(3.24)

The reference time is 2pm but the event involving the death of the girl is further in the past.

According to [5] for each of the three different aspects, the following relations between the event time and the speech time hold. They are:

1. **Simple** \( R = E \).

2. **Progressive** \( R \subset E \)

3. **Perfect** \( E < R \)

The above three relations simply state that for the simple aspect the reference time and the event time are the same. For the progressive aspect the reference time is contained in the event time and for the perfect aspect that event time happens strictly before the reference time. Below are examples of sentences with each of these aspects

\[
\text{John eats vegetables}
\]  
(3.25)

\[
\text{John is eating vegetables}
\]  
(3.26)

\[
\text{John was eating vegetables}
\]  
(3.27)

\[
\text{John has eaten vegetables}
\]  
(3.28)

\[
\text{John had eaten vegetables}
\]  
(3.29)

In (3.25) the sentence is in the simple aspect and present tense. The reference time is the same as the speech time (R=S) and as it happens in this case the event time is also equal to the reference time (E=R). In (3.26) the sentence is in the present tense with progressive aspect. The present tense
again tells us that $S=R$ and the progressive aspect indicates that $E \subset R$. This means that the event or act of John eating is in progress as the sentence is constructed. (3.27) is the same sentence but in the past tense. The tense and aspect of this sentence are true of the following two relations $S > R$ and $E \subset R$. (3.28) is a present perfect sentence. Perfect sentences ensure that the events are safely in the past. The Reichenbach structure of the events in (3.28) is $S=R$ and $E < R$. In (3.29) the past perfect means the relations for the sentence are $S > R$ and $R > E$.

Now we want to tie this idea in the notion of snapshots as in section 3.1. This is seen in the next section.

### 3.6 Sentences as Snapshot Strings and Superposition

The aim of this section is to describe how knowing the tense and aspect of a sentence allows us to represent the sentence as a regular expression using the notion of snapshots as outlined in [5].

Superposition is one of the main operations that is implemented using xfst and prolog in this project. The idea of superposition here is similar in nature to superposition in physics. Imagine it is 16:30 on a Saturday and it is raining in Berlin. This fact can be represented by the simple snapshot

$$\text{raining-in-Berlin}$$

(3.30)

If the snapshot is taken to be list of facts about the world at a certain time then clearly this snapshot only contains a tiny bit of information. Now imagine for a second that at 16:30 on the same Saturday in Wien it is snowing. This can be represented as the snapshot

$$\text{snowing-in-Wien}$$

(3.31)

The idea of superposition is that we want to be able to have a single snapshot which contains the information of both the above snapshots because
they both contain information about the same world. For example:

\[
\text{raining-in-berlin} \& \text{snowing-in-wien} = \text{raining-in-berlin, snowing-in-wien}
\]  

(3.32)

Given two languages \(L\) and \(L'\), the formal definition of superposition taken from [5] is:

\[
L \& L' = \bigcup_{n \geq 1} \{ (\alpha_1 \cup \alpha'_1) ... (\alpha_n \cup \alpha'_n) | \alpha_1 ... \alpha_n \in L \land \alpha'_1 ... \alpha'_n \in L' \}
\]  

(3.33)

To compute the superposition of two languages first one needs to construct a finite state machine representation of each language using any one of the well known algorithms mentioned in Chapter 2.

**Definition** Given two finite state machines \(A = (Q_A, \Sigma_A, \Delta_A, s_A, F_A)\) and \(B = (Q_B, \Sigma_B, \Delta_B, s_B, F_B)\), the superposition of \(A\) and \(B\), \(A \& B = (Q_{A\&B}, \Sigma_{A\&B}, \Delta_{A\&B}, s_{A\&B}, F_{A\&B})\) has the following properties:

1. \(Q_{A\&B} = \{(x, y) | x \in Q_A \land y \in Q_B\}\)
2. \(\Sigma_{A\&B} = \text{all the } \alpha \text{ tuples seen in 3.33}\)
3. \(\Delta_{A\&B} = \{(a, b, \alpha, (c, d)) | a, c \in Q_A \land b, d \in Q_B \land (a, b), (c, d) \in Q_{(A\&B)} \land \alpha = \beta \cup \rho \land (a \beta, c) \in \Delta_A \land (b, \rho, d) \in \Delta_B\}\)
4. \(s_{A\&B} = (0, 0)\)
5. \(F_{A\&B} = \{(x, y) | x \in F_A \land y \in F_B\}\)

Now consider the uninflected sentence “Tom-reside-in-Amsterdam for 10 months”. Assume that for tom-reside-in-Amsterdam we have the fluent live(p,a). For the sentence the clock \(\gamma\) is initially set to 0 and ticks up until ten months. The frequency of this ticking is irrelevant and can happen arbitrarily often. This is expressed in the Kleene Plus As outlined in [5] This
can be expressed by the regular expression

\[
\gamma(0) \square^+ \gamma(10)
\]  

(3.34)

Given the regular expression

\[
\text{live}(p,a)^+
\]  

(3.35)

which should be read as pat-live-in-Amsterdam for some arbitrary amount of time we can superpose (2.34) and (2.35) to get the regular expression:

\[
(2.34) \& (2.35) = \gamma(0), \text{live}(p,a) \text{live}(p,a)^+ \gamma(2), \text{live}(p,a)
\]  

(3.36)

If the regular languages shown above are considered the event string the superposition operator can be used to superpose a fluent which represents the reference time into the event string. As there are three different aspects and as aspect relates the reference time to the speech time as was shown in the last section there are three different languages containing the representation of the reference time \(R\). Only two of these languages need to be superposed onto the event language because in the perfect aspect the event is safely in the past according to [5]. The methods of placing the reference time into the event string are given by the following three rules.

1. Simple\((L,R) = L \& \square^* R\)

2. Progressive\((L,R) = L \& \square^+ R \square^+\)

3. Perfect\((L,R) = L \square^* R\)

From Simple\((L,R)\) it can be seen that the reference time is inserted into the last box of the language on the right side of the superposition operator (\&). This ensures that the reference time is inserted in the final snapshot of the resulting superposed language. In the Progressive\((L,R)\) the \(R\) is inserted into the middle of the language on the right of the \&. This is because, as we have already discussed, the progressive aspect places the reference time inside the event time. There is no superposition involved in the perfect because as has
already been stated there is no overlap of the two. Concatenation is used instead as seen above. In order to illustrate the above take the language \( L \) such that:

\[
L = \gamma(0),\text{live}(p,a) \uparrow \gamma(10),\text{live}(p,a)
\]  

(3.37)

\[
\text{Simple}(L, R) = \gamma(0),\text{live}(p,a) \uparrow \gamma(10),\text{live}(p,a), R
\]  

(3.38)

\[
\text{Prog}(L, R) = \gamma(0),\text{live}(p,a) \uparrow \text{live}(p,a) \uparrow \gamma(10),\text{live}(p,a), R
\]  

(3.39)

The main point to take away here is that events can be represented as strings and different events can be combined using superposition.

### 3.7 Block Reduction

In this section block reduction is described. Block reduction is used to coarsen the strings. That is we don’t want two identical snapshots one after the other. For example given the string

\[
\text{a b b a}
\]  

(3.40)

the block reduction of the string is

\[
\text{a b a}
\]  

(3.41)

This can be seen as the coarsening of the language. For the above examples the block reduction is computed by generating the lower language of the transducer in Figure 4.9. For each snapshot there is a state. Each state is final except for the initial state. From each state there is a transition to every other state. If you are in state \( b \) and see a snapshot \( a \) you go to state
with the corresponding name that is the same as the snapshots name. The idea is that if the last state name is the same as the next, you go to the next state but write an epsilon or 0.

Figure 3.1: The Block Reduction transducer

3.8 Unpadding

In the section the unpadding of the language is discussed. This is a rather simple idea. The purpose of unpadding is to remove empty snapshots from the beginning of the strings in the language. For example given the block reduced language:

\[
\text{a} \text{a} \text{b} \text{b}
\]

\[(3.42)\]
the unpadding this language results in

\[ \begin{bmatrix} a \\ b \end{bmatrix} \]  

(3.43)

This is can be visualised as a transducer such as that in Figure 3.8. The

Unpad Transducer

unpad is also computed by generating the lower language of this transducer for each non-empty snapshot. In 3.8 es represents the empty snapshot while nes is a nonempty snapshot

3.9 Allen Relations and Constraints

In this section the Allen Relations and Constraints are outlined. Firstly what are the Allen relations. The Allen relations are 13 relations and they represent 13 possible temporal relations of two events. They are visible in Figure 3.2. In the next section we show how the code for XFST derives these
relations. One may wonder why one would derive these relations given that there are only 13 of them. This is done in order to demonstrate that finite-state techniques can be used for working with events. In the next chapter it

| $p$ before $q$ | $p \quad q$ |
| $p$ meets $q$  | $p \quad q$ |
| $p$ overlaps $q$ | $p \quad p, q \quad q$ |
| $p$ starts $q$  | $p, q \quad q$ |
| $p$ during $q$  | $q \quad p, q \quad q$ |
| $p$ finishes $q$ | $q \quad p, q$ |
| $p$ equals $q$  | $p, q$ |

| $p$ after $q$ | $q \quad p$ |
| $p$ met-by $q$ | $q \quad p$ |
| $p$ overlapped-by $q$ | $q \quad p, q \quad p$ |
| $p$ started-by $q$ | $p, q \quad p$ |
| $p$ contains $q$ | $p \quad p, q \quad p$ |
| $p$ finish-by $q$ | $p \quad p, q$ |

Figure 3.2: The Allen Relations

is shown how these Allen Strings can be derived using XFST.
Chapter 4

Implementation using the Xerox Finite-State Toolkit

This chapter is devoted to discussing the practical elements of the project. Up until now everything that has been discussed was the theoretic background behind what will appear here. The aim of implementing what has been done in theory is to have an application that carries out the superposition, block reduction and unpadding. As we want to have a method of handling events and states and the desire for software to do so this implementation is to provide some useful tools that will be necessary for further work. This chapter is written in such a way that assumes no previous experience working with the Xerox Finite-State Toolkit.

4.1 Xerox Finite-State Toolkit

The Xerox Finite State Toolkit is a toolkit that was developed in Palo Alto Research Centre(PARC) in California. Finite State Methods had been used long before the development of the finite state toolkit but they were thought to be too weak for carrying out any practical tasks. Most of the work done using them was very low level programming. [1] is book that demonstrates how finite state methods can be used to solve problems encountered in morphology. It covers various topics such as using transducers as translators and
generators. This book also doubles as a guide to using the xerox finite-state tools.

There are many different reasons for using XFST. XFST reads in regular expressions, prolog code and other data forms that represent finite-state networks. Once the networks are in the XFST we can use many different XFST commands. These commands do various different things such as testing whether or not a word is in a language. In the case of transducers we can do translation from the input language to the output language can be performed. The networks can be printed out to the command line or to a file or to prolog code (as is done in this project). One of the main benefits of XFST and the main reason that it is used in this project, is that all of the standard operations that one would carry out such as pruning, determinising, minimising, removing epsilons can be done in the toolkit. These algorithms are often tricky to write well and in the Xerox Finite State Toolkit they have already been written. To write all of these methods would have been reinventing the wheel, time consuming and would have taken away from the work that went into the project itself. XFST also allows one to write their own operations on finite-state networks and indeed this is the main part of this project.

There are 2 main programs that come as a part of the toolkit. XFST and LEXC. Lexc is a lexicon compiler. According to [11] XFST is:

- An interface that gives programmers access to the finite-state operations that are discussed above.
- A regular expression compiler

The regular expressions that are used by XFST are an extension of standard regular expressions.

4.2 How does XFST work?

In this section a description of how to use XFST and how it is used in the project is given. We start by describing the interface and how a person would
interact with it. The information found in this section is sourced from [1] and [11]

4.2.1 The Interface

Figure 4.1 is a screenshot of what the XFST interface actually looks like running on Windows.

Although there is a Graphical User Interface for XFST the command line version is used in this project. The GUI is more practical for when one wants to define finite state networks that would be useful for dealing with large languages. In this operations on finite-state networks are designed for use under the surface and therefore the use of the command line XFST program is sufficient. Indeed it would be very easy to extend the program written to be able to deal with these things.

4.2.2 The Stack

One of the most frequently used features of XFST is the regular expression compiler. Examples of regular expression compilers can be seen in Figure 4.2.
The “read regex REGEX” command is used to compile a regular expression.

This first regular expression read in Figure 4.2 is /baa+/ . This is the regular expression which accepts the infinite language
\[ L_{\text{sheep}} = \{ baa, baaa, baaaa, baaaaa, \cdots \} \]. When the regular expression is read it is pushed onto a stack. This is indicated in the screen shot by the 0 changing to a 1 after the XFST prompt. If more networks are read they will be pushed onto the stack in position 3, then 4, then 5 etc. We can test whether or not a string is part of a regular language by using either the apply up or apply down commands. The first apply up tests whether or not the string \( ba \in L_{\text{sheep}} \). We can tell it’s not because as seen in the next case with \( baa \) xfst prints the word again if it accepts the string. The result in the first example is the same using apply down. The reason for this is because
the regular expression \( /\textit{baa}^+ / \) is actually compiled as the regular relation\(^1\) \\
\( /[b : b][a : a][a : a]^+/ / \). The letters on the left hand side of the colon are on the upper/input side and the letters on the right hand side of the colon are on the lower/output side. When a finite-state transducers input and output alphabet are the same on all arcs the transducer is in fact the a finite-state machine represented differently.

Next the regular relation \\
\( /[c : c][a : a][t : t][+Sg : 0] + Pl : s]/ \) is read onto the stack. The pipe simply represents a disjunction. Using the apply down this time will print out the corresponding string of the lower language if it accepts the input upper string is accepted. Using the apply up has the reverse affect. It returns a lower language’s corresponding upper string if that string is accepted. In the screen shot example the 2nd network on the stack is an example of finite-state morpholy. If the upper language is Cat+Sg the lower language is the lexical form of the morphemes in the upper language. Cat+Sg is cat and Cat+Pl is cats.

Some other useful features of XFST are also visible in this screen shot. For starters we can print all the words of a finite language. They can be seen in the form of cat(+Sg:0) and cat(+Pl:s). The 0 in XFST is to be understood as epsilon. Aswell as pushing to networks onto the stack the pop command removes the topmost network on the stack. This is another important feature of the networks that is made use of in the implementation of the project. Notice that when the words of the infinite language \( /\textit{baa}^+ / \) are printed only one word is output \( /\textit{baa}/ \). The part of the infinite network that is printed out is the smallest string that is accepted before the iteration begins.

In the screenshot of XFST here it can also be seen that we can define variables to store regular expressions. This is not used in the implementation and therefore does not merit discussion here.

When compiling the regular expressions above it should also be noted that XFST determinises, minimises and prunes the finite-state networks as much as possible. Obviously this is done as a type of housekeeping to ensure that the finite state networks are as simple as possible. There will be more

\(^1\)Regular relations are isometric to Finite-State Transducers
about determinising, pruning and minimizing in the next section.

4.2.3 Using SWI-Prolog with XFST

SWI-Prolog is an open source implementation of the prolog programming language and is used as the primary programming language of this implementation of the main part of this project. Although there is no direct interface for working with prolog in XFST, XFST does have the ability to parse a certain set of prolog facts that represent a finite state network as well as write out the networks in it’s stack in terms of prolog facts. It is important here to note that XFST does not interpret the prolog but parses the prolog rules it is given. In order to understand how the implementation works it is necessary to show this in operation. Figure 4.3 is a finite-state

network(test).
arc(test,0,1,"a").
arc(test,0,2,"a").
arc(test,1,3,"b").
arc(test,2,3,"b").
arc(test,1,4,"c").
arc(test,2,4,"c").
arc(test,4,4,"a").
arc(test,4,4,"b").
arc(test,14,15,"REDUNDANT").
final(test,4).
final(test,3).

Figure 4.3: Non-Deterministic,Non-Minimal,Non-pruned FSA

machine. This finite state machine is written so that XFST can parse it. Line 0 just names the network as test. The predicate arc contains the name of the network the arc belongs to and in this case it’s the network “test”. The arc fact contains two numbers. The first number is the current state and the second is the next state. The last state is the label and must be enclosed between two ”’s. At the last two lines the facts denote the final states of the
test network as state 4 and state 3. A careful reader will notice that this is not explicitly stated. However when this network is read into XFST the state 0 is interpreted as the initial state\footnote{Clearly a machine with more than one start state is non-deterministic because there are multiple starting points. As every NFSA can be determinised this means that an equivalent machine with just one initial state can be created. This explains how XFST can just use 0 as the only possible starting state}. In XFST finite state machines can only have one start state. This Automata accepts the single string accepts an infinite language. A regular expression for it is $/ [a|b|c][a|b|^*] /$. The network is non-deterministic because immediately at the start there is a choice of two states for the label "a". This finite-state machine is also not minimal because there are two different paths that accept the word "bc" and they are identical and indistinguishable. This network is also not pruned because the arc four five is clearly redundant as there is no path for it. In the project we want to read networks of this kind into XFST. This is done with the read prolog command as seen in Figure 4.4. The read prolog command can be seen at the top of the screen shot. This reads the network onto the stack. The next thing that happens is that the network is minimized. XFST is able to prune, determinize and minimize networks. When one minimizes a network it is in fact pruned and determinized if this has not been done so already. Therefore calling minimize can be thought of as pruning determinising and minimising the network all at once. So to get the smallest most compact and efficient networks it is best to simply call minimize from the outset. Minimising the network in figure 4.3 results in the network in figure 4.5.

This minimized version of the network can then be written to a prolog file using the write prolog command. In this section and in the previous features of the XFST most of the features of XFST that are utilised during

Figure 4.4: Reading and Writing Prolog
network(test).
symbol(test, "REDUNDANT").
arc(test, 0, 1, "a").
arc(test, 1, 2, "b").
arc(test, 1, 2, "c").
arc(test, 2, 2, "a").
arc(test, 2, 2, "b").
final(test, 2).

Figure 4.5: Deterministic, Minimal and Pruned Version of 4.3

the project are detailed. There is one more XFST feature that is made use of. That is dealt with in the next section.

4.3 XFST scripting

In chapter 2 the use of XFST scripts is explained. Simply using the command line and entering commands one at a time is not very practical when we want to deal with large amounts of data. Indeed when using Prolog to process the XFST stack as is done here most of what we want to do needs to be done automatically. XFST has its own type of scripts and in this project we use them. XFST scripts are not programming languages. However they enable us to access the operating system using the “system” command and from there more powerful operations can be carried out using different scripting languages. Figure 4.6 is an example of an XFST script. It simply reads two regular expressions onto the stack. Defines a regular language VAR and that is pushed onto the stack and minimized. The “system” call at the end is an example of XFST invoking a Perl script. As will be seen later on XFST scripting is used to bind together and control the PROLOG and PERL processing of the finite state machines.
4.4 An XFST representation of the Snapshot

In the previous chapter the idea of the snapshot has been formalised. Now this formalisation needs to be represented somehow in XFST. Here the representation used is discussed. In the next section the actual coding is discussed in detail.

Recall that a snapshot is a set of fluents that hold at a particular point in time as discussed in chapter two. In order to keep things simple it was decided that for every fluent there would be a corresponding single character. This has an obvious drawback. As there is only a finite set of characters only a finite number of fluents can be represented. This is of course a problem that is easily fixed by using slightly different coding.

Now consider an example that has been seen before. Recall that the concept “rain from dawn until dusk can be represented” by the following string of snapshots:

\[ \text{RAIN,DAWN} \text{RAIN}^+ \text{RAIN,DUSK} \]  

(4.1)

To represent this in XFST one must encode a function which converts fluents into a single character. This is not a difficult task and such a function for the above snapshot would look like the in Figure 4.7.

It is important to note that when a fluent is mapped to a character by \( f \) then the character can be mapped to the fluent by the inverse \( f^{-1} \). This is important because it has to be possible to translate the characters back into the fluents they represent at the end.

This step might seem redundant but remember one can only represent
\[ f : FLUENTS \rightarrow CHARACTERS \]
\[ f(RAIN) \rightarrow r \]
\[ f(DAWN) \rightarrow d \]
\[ f(DUSK) \rightarrow a \]
\[ f^{-1} : CHARACTERS \rightarrow FLUENTS \]
\[ f^{-1}(r) \rightarrow RAIN \]
\[ f^{-1}(d) \rightarrow DAWN \]
\[ f^{-1}(a) \rightarrow DUSK \]

Figure 4.7: A function mapping fluents to their character representations

fluent by single characters. Given the function \( f \) the snapshot above would be represented as by the following regular expression.

\[ r, d \quad [r]^{+} \quad [r, a] \quad (4.2) \]

How would one represent this in XFST. In the above example each snapshot as always represents the time changing or passing. We can therefore represent a boundary in time using the special fluent \( \# \). Therefore in XFST as there are no boxes. If the \( \# \) is included in the snapshot this represents the passing or time. As it happens this fluent must always be placed either at the beginning of the snapshot or at the end of the snapshot. In the implementation presented here the end of the snapshot was chosen. Therefore in XFST the representation of rain from dawn to dusk will be.

\[ rd\# \quad r\# \quad + \quad ra\# \quad (4.3) \]

The spaces between the snapshots mean that everything beforehand is seen as one symbol. The minimal words of this infinite language can be printed in XFST. This is seen in Figure 4.8

As can be seen when the words of the language are printed the use of the \( \# \) makes the snapshots clearly visible. It is also worth pointing out here the \( \# \) is a special character in XFST and to represent the empty snapshot one has to use \( \%\# \). This will be seen again later. Now that the representation of the snapshots in XFST has been explained the coding of the operations
on finite-state machines can be discussed.

4.5 Coding Superposition

There are three main steps involved in the implementation. The first is the coding of the superposition operation on two finite-state machines. The second is the coding of the block reduction on a single finite-state machine and the third is the coding of the unpad on a finite-state machine. This section is devoted to superposition.

Firstly how does it work in general? Initially a regular expression is compiled by XFST and placed at position one on the stack. Then a second regular expression is compiled by XFST and placed on the second position on the stack. The user is trusted to input regular expressions in the correct form. There is no effort made to ensure that the input is valid. As in the last section it is assumed that every snapshot has got a # and this hash must be placed at the end of each snapshot in order for the prolog code to process it correctly. The idea is to then write the stack to a prolog file using the write prolog command. The the prolog interpreter is invoked with the file compute1.pl. The file compute1 is written in such a way that as soon as it runs it adds the prolog specification of the two finite-state machines being superposed to the facts. It tries to prove the predicate process/0. The code for discussion here is in appendix B.1. The process predicate tries to prove createFsms. The createFsms actually instantiates A and F to the arcs and final states of the superposed FSMS. Writer writes the superposed fsm to an output file.

Now let us get a better look at how createFsms/2 actually works. First
the arcs of the network are transformed into a list of the form [CurrentState,Label,NextState] using the findall and the arcs of the FSM are then given to the superpose predicate which creates the list of arcs in the superposed automaton. This is done according to the formal definition of superposition given in chapter 3. That is it superposes each arc of the first fsm with every arc of the second FSM. The finals predicate generates all possible finite states for the FSM.

It should also be noted how the states are treated here. The states are represented as pairs [x,y] where x is the state of the first fsm and y the state of the second FSM. As XFST is not able to read state names of the from [x,y] where x and y are integers a way of mapping these state names into a unique number was needed. The cantor pairing function is a function that maps a pair of natural numbers into a unique natural number ie $\pi : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$. It is defined as

$$\pi (x, y) = \frac{1}{2} (x + y)(x + y + 1) + y$$ \hfill (4.4)

The cantor predicate represents this. In prolog however the function always returns a unique number followed by a point zero(0). For example cantor([2,3],18.0). This 18.0 is still not readable by XFST. Therefore it needs to be changed to 18. It was decided to do this clean up operation outside of prolog using perl. Furthermore XFST requires labels in the form “some-label”. However in prolog the quotes are always interpreted. It was decided that it would be easier to solve this problem outside of prolog using perl also.

For example superposing the language $rd\# r\# + ra\#$ with $s\#+$ The output of the prolog program is seen below. network(net).

arc(net, 8.0, 8.0, [r, s, #]).
arc(net, 8.0, 13.0, [a, r, s, #]).
arc(net, 4.0, 8.0, [r, s, #]).
arc(net, 1.0, 4.0, [d, r, s, #]).
arc(net, 5.0, 8.0, [r, s, #]).
arc(net, 5.0, 13.0, [a, r, s, #]).
arc(net, 2.0, 8.0, [r, s, #]).
arc(net, 0.0, 4.0, [d, r, s, #]).
Perl has rich support for regular expressions and makes converting this into something that is XFST readable is quite easy. Running the perl code in appendix B.2 results in the following output given the input seen above.

```
network(net).
arc(net,8,8,"rs").
arc(net,8,13,"ars").
arc(net,4,8,"rs").
arc(net,1,4,"drs").
arc(net,5,8,"rs").
arc(net,5,13,"ars").
arc(net,2,8,"rs").
arc(net,0,4,"drs").
final(net,13).
```

This output is in the form that is desired. It is also visible here that a lot of extra arcs are here. These are gotten rid of by minimizing the network when it is read into XFST.

### 4.6 Coding Blockreduction

In this section the coding of the blockreduction is outlined. This block reduction is designed so that it is carried out on a finite-state machine that was created from superposition. As has been discussed in chapter two the purpose of the block reduction is to give a finite-state machine that does not contain two identical snapshots back to back. This means that time does not elapse unless something in the snapshot has changed. The prolog code for this part of the application is seen in appendix B.3.

The code here works very similar to the code for superposition. Immediately after the prolog code is compiled it automatically tries to satisfy the goal process. This in turn generates the arcs of the blockreduced finite-state machine. It works by using the barc predicate to find all possible arcs of the
Figure 4.9: The Block Reduction transducer

block-reduced machine. There are four barc predicates and each one specifies a different arc that belongs in the block reduced finite state machine. The final states are computed using the blockreducedFinals predicate and the writer outputs the file.

There is a complication here however. It manifests itself in the fact that the state names of the finite-state machine are named after the snapshot. As the snapshots are represented in prolog by character codes computing the statename using the cantor pairing function results in statenames which are huge numbers. As a result in the blockreduction program it was decided to simply concatenate each character code in the snapshot in order to deal with shorter numbers. This for the snapshot [a,b,§] the state name is 979835 as the character codes for a,b and § are 97, 98 and 35 respectively. These numbers are still very large however unlike using the cantor function they are never interpreted as numbers and therefore this does not matter. Essentially
what happens here is the lower language of the transducer in Figure 4.9 is generated as this lower language is the blockreduced finite-state machine. Note however that as already mentioned in chapter 3 the transducer in 4.9 only contains two distinct snapshots. Recall that $a$ and $b$ here are sets. The more snapshots there are the larger this transducer gets. This may not be the best way to compute the block reduction but here purity of methods is desirable. What this means is that one wants to do these operations using finite-state methods. The output of the block reduction for the same input used with superposition is the following:

```
network (red).
arc (red, [0, 0], [1, 10011411535], [d, r, s, #]).
arc (red, [1, 11411535], [2, 11411535], [0]).
arc (red, [2, 9711411535], [3, 9711411535], [0]).
arc (red, [2, 11411535], [2, 11411535], [0]).
arc (red, [1, 10011411535], [2, 11411535], [r, s, #]).
arc (red, [1, 9711411535], [2, 11411535], [r, s, #]).
arc (red, [2, 11411535], [3, 9711411535], [a, r, s, #]).
arc (red, [2, 10011411535], [3, 9711411535], [a, r, s, #]).
arc (red, [2, 11411535], [3, 9711411535], [a, r, s, #]).
arc (red, [2, 10011411535], [2, 11411535], [r, s, #]).
final (red, [3, 9711411535]).
```

Running the PERL script in appendix B.4 converts it into the XFST readable:

```
network (red).
arc (red, 0, 1, "drs ").
arc (red, 2, 3, "0").
arc (red, 4, 5, "0").
arc (red, 3, 3, "0").
arc (red, 1, 3, "rs ").
arc (red, 6, 3, "rs ").
arc (red, 7, 5, "ars ").
```

44
4.7 Coding Unpad

In this section we discuss the final operation defined. That is the unpad operation. This works again in a similar fashion to the blockreduction. As outlined in chapter 3 unpadding involves generating the lower language of a particular transducer. The code for unpadding a transducer is seen in appendix A.5. Again the predicate process. This finds all the arcs of the unpadded finite state machine. This machine happens to be the machine accepting the lower language of the transducer seen for unpadding in chapter 3. In the usual manner a perl script makes the output readable by XFST. To see unpad in working consider the following language.

\[#a#\] (4.5)

. Using the unpad programme on this results in the following fsm:

\[
\text{network(unpad).}
\]

\[
\text{arc(unpad, [0, 0], [1, 0], [0]).}
\]

\[
\text{arc(unpad, [2, 0], [3, 0], [0]).}
\]

\[
\text{arc(unpad, [1, 0], [2, 1], [a, ]).}
\]

\[
\text{arc(unpad, [1, 0], [2, 2], [a, ]).}
\]

\[
\text{arc(unpad, [0, 1], [1, 1], []).}
\]

\[
\text{arc(unpad, [2, 1], [3, 1], []).}
\]

\[
\text{arc(unpad, [1, 1], [2, 1], [a, ]).}
\]

\[
\text{arc(unpad, [1, 1], [2, 2], [a, ]).}
\]

\[
\text{arc(unpad, [0, 1], [1, 2], [0]).}
\]

Clearly this machine can be pruned, determinized and minimized and this is carried out before the unpadding is done on the machine.
arc(unpad, [2, 1], [3, 2], [0]).
arc(unpad, [0, 2], [1, 2], [0]).
arc(unpad, [2, 2], [3, 2], [0]).
final(unpad, [3, 0]).
final(unpad, [3, 2]).
The Perl script cleans this up to resemble:

network(unpad).
arc(unpad, 0, 1, "0").
arc(unpad, 2, 3, "0").
arc(unpad, 1, 4, "a").
arc(unpad, 1, 5, "a").
arc(unpad, 6, 7, "").
arc(unpad, 4, 8, "").
arc(unpad, 7, 4, "a").
arc(unpad, 7, 5, "a").
arc(unpad, 6, 9, "0").
arc(unpad, 4, 10, "0").
arc(unpad, 11, 9, "0").
arc(unpad, 5, 10, "0").
final(unpad, 3).
final(unpad, 10).

The above code obviously can be minimized and when it is the output is

network(unpad).
symbol(unpad, ").
arc(unpad, 0, 1, "a").
final(unpad, 1).

It is easy to see that this FSM only accepts the string a as we would expect.
4.8 Combining Superpose, Blockreduction and Unpad

For the demonstration of this project this was all put together to show how the Allen relations are derived using XFST. How this is does is actually using the following XFST script:

```plaintext
write prolog > fsm1.pl;
clear stack;
system plcon.exe -s compute1.pl
system perl prolognormalis.pl
read prolog < arcsout.pro;
prune;
determinize;
minimize;
write prolog > superposedfsm.pl;
clear stack;
system plcon.exe -s blockreduction.pl
system perl br2xfst.pl
read prolog < bredxfst.pro;
prune;
determinize;
minimize;
write prolog > result.pl;
clear stack;
system plcon.exe -s unpad.pl
system perl unp.pl
read prolog < unpad2.pro
prune;
determinize;
minimize;
```

This script assumes that at the beginning there are two networks on the stack that are to be superposed onto one another. It writes them to a prolog
file and involves prolog to carry out the superposition. It then invokes a perl script to make the output of the superposition readable to XFST. The output of this PERL script is read into XFST onto an empty stack. The script then minimizes the output and writes it to another file. This file is the source of the block reduction and the prolog code for blockreduction reads from this file and automatically outputs the blockreduced fsm to another file. This file is converted into an XFST readable file by perl and again read into XFST to be pruned, determinized and minimized etc. Once again the script invokes the unpad prolog file and this creates an unpadded version that is to be read onto the stack of XFST after it has been made XFST readable by PERL as is done with the previous two stages. In the next section this script is demonstrated in action.

4.9 Deriving the Allen Relations

In this section it is shown how the code is used to derive the Allen Strings in XFST. The 13 strings can be derived by superposing # + e# #+ and # + f# #+. The result of this is block reduced and then it is unpadded to yield the 13 Strings. The screen shot in figure 4.10 shows how to read the two languages into the stack in XFST. This is the first step involved in deriving the strings.

![Figure 4.10: Reading the Languages onto the Stack](image)

To carry out the superposition, blockreduction and unpadding all at once and to read the resulting language onto XFST’s stack one must simply invoke the XFST script superpose.xfst as shown in Figure 4.10. The result of printing the words in the network after this operation can be seen and
comparing the output strings in 4.11

Figure 4.11: Printing the 13 Allen Relations with XFST
Chapter 5

Further Work and Conclusions

So now that there is an implementation of superposition, blockreduction and unpad a natural question is what happens next. There are several limitations of the project that should be addressed. Here they are briefly discussed.

5.1 What has been seen so far?

What this section offers was an implementation of some of the techniques that will be needed for extending this project to deal with more events and states and implements tools that are necessary for making the theory in chapter 2 come to life.

5.2 Coding Improvements

The software that was developed has some limitations. It would work fine for dealing with short discourses where a large amount of fluents are not required. This is due to the fact that in the implementation we can only have as many fluents are there are in prolog. An obvious improvement here would be to alter the fluent representation strategy in order to be allow as many fluents as can be stored in the computers memory.

A second improvement to the code itself would be to remove the use of the cantor pairing function from the coding of superposition. As large finite
state networks in XFST are represent the states names as integers using the
cantor pairing function on large numbers becomes a problem pretty fast.
Storing the numbers as prolog atoms rather than numbers as is done for the
Blockreduction is a better solution.

As it currently stands the superposition code is defined in such a way
that it generates a finite-state machine that is not pruned, not deterministic
and not minimal. For all the examples we have seen this is not a problem
and XFST has the required tools for dealing with this. However all of the
networks that have been demonstrated in this project were relatively speaking
quite small. With larger networks pruning can take quite a while. It would
be better if these arcs were not generated in the first place.

5.3 Discourse Representation Structures as
Languages

This project works on the assumption that the Discourse Representation
Structures that are presented in [7] can be adequately expressed as languages.
The next thing that should be worked on is an algorithm for converting a
DRS to a Regular Language and the creation of events as strings as seen
above. The project as it is assumes this is possible. Obviously this would be
a critical next step in taking this work further. At the moment it does not
really look much like Discourse Representation Theory but it is important to
note that the code will definitely be necessary for dealing with events when
such an algorithm has been found.

5.4 Constraints and Inference

In [4] it is argued that by placing certain constraints on the event strings
inferences can be made. Informally this can be considered as for example
having a constraint that a week is made up of days. If one snapshot represents
an interval of a week it can be inferred that it can be broken down into 7
smaller time intervals (ie days) and that the facts are true on each day. This
is seen in [4] in the form of the subsumption operator. An obvious next step would be to try get the program to be able to work out these steps and to be able to carry out inferences using the event strings.

5.5 Was this project a success?

The code developed in this project provides a base for more work in this area. As has already been stated it is necessary for any practical application in this area. It makes certain assumptions and some of these assumptions have not as of yet been met but that is an area for further research. Despite the drawbacks of the code mentioned above it is usable by anyone who wants and is available on the cd accompanying this report.

The project is titled towards finite-state discourse representation theory as it is a step towards dealing with tense and aspect in a finite-state setting and is only a minor step towards full finite-state DRT.
Bibliography


Accessed 02/04/2010 at 16:00
Appendix A

Complete Code Listings

A.1 Superpostion- compute1.pl

:-[fsm1].

cantor([X,Y],Z):-
    A is X+Y,
    B is A+1,
    C is B*A*(1/2),
    Z is C+Y.

superpose(Arcs1,Arcs2,ArcsOut):-
    sup(Arcs1,Arcs2,Arcs2,[],ArcsOut).

sup([],[],X,X).

sup([T],[],Copy,Akk,Out):- sup(T,Copy,Copy,Akk,Out).

sup([[A,B,C]|D],[[E,F,G]|H],Copy,Akk,Out):-
    append(B,F,Wohoo),sort(Wohoo,Indigo),
    Indigo=[Yes|No],append(No,[Yes],Finally),
    sup([[A,B,C]|D],H,Copy,[[[A,E],
    Finally,[C,G]]|Akk],Out).
statepairscantored([],[]).
statepairscantored([[A,B],[C],[D,E]]|F),

cantor([A,B],G),
cantor([D,E],H),
statepairscantored(F,I).

createFsms(Arcs3,Fs):-
  network(A),network(B),not(A=B),
  findall([X,Y,Z],arc(A,X,Z,Y),ArcsN1),
  findall([J,K,L],arc(B,J,L,K),ArcsN2),
  superpose(ArcsN1,ArcsN2,Arcs3),
  finals(Fs).

finals(Fs):-
  network(A),network(B),not(A=B),
  findall(X,final(A,X),FinalStates1),
  findall(Y,final(B,Y),FinalStates2),
  findall(Yay,(member(F,FinalStates1),
    member(G,FinalStates2),
    cantor([F,G],Yay)),Fs).

process:-createFsms(A,Fs),
    writer(A,Fs),
    !.

formatarc([A,B,C],arc(net,X,Y,Z)):−
cantor(A,X),
cantor(C,Y),
numbersToLetters(B,Z).

writer(Arcs,Fs) :-
  open('ducksout.pro', write, ID),
  writeq(ID, network(net)), write(ID, '.\n'),
  write_arcs(ID, Arcs),
  write_finals(ID, Fs),
  close(ID).

write_arcs(ID,[H|T]) :-
  formatarc(H,K),
  writeq(ID,K),
  write(ID, '.\n'),
  write_arcs(ID,T).
write_arcs(_,[]).

write_finals(ID,[H|T]) :-
  writeq(ID, final(net,H)),
  write(ID, '.\n'),
  write_finals(ID,T).
write_finals(_,[]).

numbersToLetters([],[]).
numbersToLetters([H|T],[S|F]) :-
  name(S,[H]),
  numbersToLetters(T,F).

:- process, halt.

A.2 prolognormalis.pl

open(INFILE, "ducksout.pro");
open(OUTFILE,">arcsout.pro");
while(<INFILE>){
    $line=$_
    $beginning =$line;
    $beginning =~ s/([^d+])\.(^[d+])/$1/g;
    $beginning =~ s/\[/"/g;
    $beginning =~ s/\]/"/g;
    $beginning =~ s/ //g;
    $string=$1 . $2;
    $count = ($string =~ tr/./);  
    for ($i=0; $i<$count; $i++){
        $beginning =~ s/(arc\(net,\d+,\d+,[a-z]*\\),\(.*\))/$1$2/g;
    }
    print OUTFILE $beginning;
}

A.3 blockreduction.pl

:-[superposedfsm].
sym(X,SYX):- arc(X,_,_,SYX).
cantor([X,Y],[X,Y]).
um(SY,N):-
  concat_atom(SY,Atom),
  atom_chars(Atom,Chars),
  number_chars(N,Chars).
diffSym(SY,SX,X):-
  sym(X,SX),
  not(SY=SX).

arcs(Arcs):-
  findall(X,(barc(red,B,C,D),
    X=barc(red,B,C,D)) ,Arcs).

barc(red,Qn,Rn,SY):-
\begin{verbatim}
arc(net,0,R,Sym),
cantor([0,0],Qn),
num(Sym,N),
cantor([R,N],Rn).
barc(red,Qn,Rn,"0") :-
arc(net,Q,R,Sym),
not(Q=0),
num(Sym,N),
cantor([Q,N],Qn),
cantor([R,N],Rn).
barc(red,Qn,Rn,Sym) :-
arc(net,Q,R,Sym),
not(Q=0),
sym(net,SyX), not(SyX=Sym),
num(SyX,Nx),
num(Sym,N),
cantor([Q,Nx],Qn),
cantor([R,N],Rn).
formatarc(barc(red,X,Y,C),arc(red,X,Y,Z)) :-
numbersToLetters(C,Z).
writer(Arcs,Fs) :-
  open('bred.pro', write, ID),
  writeq(ID, network(red)), write(ID, ".n"),
  write_arcs(ID, Arcs),
  write_finals(ID, Fs),
  close(ID).
write_arcs(ID,[H|T]) :-
  formatarc(H,K),
  writeq(ID,K),
  write(ID, ".n"),
  write_arcs(ID,T).
write_arcs(_,[]). 
write_finals(ID,[H|T]) :-
\end{verbatim}
writeq(ID, final(red,H)),
write(ID, ' \n'),
write_finals(ID,T).
write_finals(_,[]).

input_finals(Finals):— findall(X, final(net,X), Finals).

blockreduced_finals(BRF):—
  input_finals(F),
  arcs(Arcs),
  finals(Arcs, Finals),
  sort(Finals, Finals1),
  findall([X,Y], (member(X,F), member([X,Y], Finals1)), BRF).

process:—
  arcs(Arcs),
  blockreduced_finals(B),
  writer(Arcs,B).

numbersToLetters([],[]).
numbersToLetters([H|T],[S|F]):—
  name(S,[H]),
  numbersToLetters(T,F).

finals(Arcs, Finals):—
  findall(A, member(barc(red,A,_,_), Arcs), Finals1),
  findall(A, member(barc(red,_,A,_), Arcs), Finals2),
  append(Finals1, Finals2, Finals).

:- process, halt.
A.4  br2xfst.pl

%states =();
@usedNames=(−1);
$count=−1;
open(INFILE,” bred.pro”);
open(OUTFILE,” > bredxfst.pro”);
while(<INFILE>){
  if ($ =~ m/arc\(red , \(\[.*,.\]\) , \(\[.*,.\]\) , (.\)\)/){

    if (! exists($states{$1})){

      while(&member){
        $count++;
      }

      $states{$1}=$count;
push (@usedNames,$count);
    }
  }
  if (! exists($states{$2})){

    while(&member){
      $count++;
    }

    $states{$2}=$count;
push (@usedNames,$count);
  }
}
$tmp1 = $states{$1};
$tmp2 = $states{$2};
$string = "arc\(red, \" . \$tmp1" . \" . \$tmp2" . \" . \" . $3";
$string =~ s/[\[/\]/g;
$string =~ s/[\]/\]/g;
$string =~ s///g;
$counttt = ($string =~ tr//);   
for ($i = 0; $i <$counttt; $i++){
    $string =~ s/(arc\(\(red, \d+, \d+, [a-z]*\), (.*)$/ $1$2/g;
}

print OUTFILE "$string\n";
} else{
    if ($_=~/m/network\(.*\)\)/{
        print OUTFILE "$_";
    }
}
} else{
    if ($_=~/m/final\(red, (.*)\)\)/{
        print $1;
        if (!exists ($states{$1})){
            print "bang\n";
            while(&member){
                $count++;
            }
        }
    }
}

$states{$1} = $count;
push (@usedNames, $count);
$tmp3 =$states{$1};
$str="final(red," . "$tmp3").";
print OUTFILE "$str
";

sub member{

foreach (@usedNames){
    if ($_==$count){
        return 1;
    }
}
}

A.5 unpad.pl

:-[result].

barc(unpad,[P,0],[Q,0],"0"):-
    arc(red,P,Q,"#").

barc(unpad,[P,0],[Q,1],X):-
    arc(red,P,Q,X),
    not(X="#").
barc (unpad, [P, 0], [Q, 2], X): −
   arc (red, P, Q, X),
   not (X="#”).

barc (unpad, [P, 1], [Q, 1], ", #"): −
   arc (red, P, Q,” #”).

barc (unpad, [P, 1], [Q, 1], X): −
   arc (red, P, Q, X), not (X=” #”).

barc (unpad, [P, 1], [Q, 2], X): −
   arc (red, P, Q, X),
   not (X=" #”).

barc (unpad, [P, 1], [Q, 2], ”0”): −
   arc (red, P, Q,” #”).

barc (unpad, [P, 2], [Q, 2], ”0”): −
   arc (red, P, Q,” #”).

arcs (AA): −

writer (Arcs, Fs) :-
   open (’unpad.pro’, write, ID),
   writeq (ID, network (unpad)), write (ID, ’.\n’),
   write_arcs (ID, Arcs),
   write_finals (ID, Fs),
   close (ID).

numbersToLetters ([], []).
numbersToLetters ([H|T], [S|F]): = name(S, [H]), numbersToLetters(T, F).

formatarc (barc(A, B, C, D), arc(A, B, C, Z)): = numbersToLetters(D, Z).

write_arcs(ID, [H|T]) :-
    formatarc(H, K),
    writeq(ID, K),
    write(ID, ".\n"),
    write_arcs(ID, T).
write_arcs(_, []).

write_finals(ID, [H|T]) :-
    writeq(ID, final(unpad, H)),
    write(ID, ".\n"),
    write_finals(ID, T).
write_finals(_, []).

finals(F): -
    findall([X, 0], final(_, X), F1),
    findall([Y, 2], final(_, Y), F2),
    append(F1, F2, F).

process: -
    arcs(A), finals(Fins), writer(A, Fins).

:- process, halt.

A.6  unb.pl

%states = ("[0, 0]" => 0);

@usedNames=(-1,0);
$count=-1;
open(INFILE," unpad.pro");
open(OUTFILE," > unpad2.pro");
while(<INFILE>){
    if ($_ =~ m/arc\s\((unpad, (\[.\.,.\]\)), (\[.\.,.\]\))\)/{
        #print "###########
        #print "$1\n";
        #print "$2\n";
        #print "###########

        if (!exists ($states{$_})){
            #print "*************\n";

            print "$1\n";
            while(&member){
                $count ++;
            }

            $states{$_} = $count;
            push (@usedNames, $count);
        } else {
        }

        if (!exists ($states{$_})){
            print "$2\n";
            while(&member){
                $count ++;
            }

            $states{$_} = $count;
            push (@usedNames, $count);
        } else {

        }
    } else {
    }
}
$tmp1=$ states{$1};
$tmp2=$ states{$2};
$string= "arc\((unpad, \text{"$tmp1"}, \text{"$tmp2"}, \text{"$3"};
$string=~ s/\[/"/g;
$string=~ s/\]/"/g;
$string=~s/ //g;
$counttt = ($string =~ tr /,//); print "$counttt
";
for ($i=0; $i<$counttt; $i++){
    $string =~ s/(arc\((unpad,\d+\d+,[a-z]*),(.*))/$1$2/g;
}
print OUTFILE "$string
";
} else{
    if ($_=~m/network\((.*))/.){
        print OUTFILE "$_";
    } else{
        if ($_=~m/final\((unpad, (.*)\))/.){
            print $1;
            if (!exists($states{$1})){
                print "bang\n";

                while(&member){
                    $count++;
                }
            }
        }
    }

    $states{$1}=$count;
    push (@usedNames, $count);
}
} else {

}
$tmp3=$states{$1};
$str="final(unpad,".$tmp3.");"
print OUTFILE "$str\n";

}

}}

sub member{

    foreach (@usedNames){
        if ($_==$count){
            return 1;
        }
    }

}