Regular Languages for Siskind's Event Logic

Sarah Fortune
B.A. (Mod.) Computer Science
Final Year Project, May 2007
Supervisor: Dr. Tim Fernando

May 3, 2007

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.

May 3, 2007

Sarah Fortune

Permission to Lend

Acknowledgments

I wish to thank Tim Fernando for supervising this project and for all his help and encouragement over the year. I also thank Andrei Barbu for his continual loving support.

Time is an illusion. Lunchtime doubly so.
Douglas Adams (1952 - 2001)

Time is nature's way of keeping everything from happening at once.
Woody Allen (1935 -)

Contents

List of Figures

Abstract

This project shows that the Event Logic introduced in [Sis01] can be expressed in Finite State Temporality, a temporal logic based on regular languages [Fer04]. Finite State Temporality is a more powerful logic than Event Logic. It describes the implementation of a program which can build finite state representations of Event Logic formulae, which can be used to perform event recognition.

Chapter 1

Introduction

1.1 Project and Purpose

The project will show how the temporal logic of [Sis01], Event Logic, can be expressed in Finite State Temporality [Fer04]. Event Logic is used to describe the semantics of verbs in a way that can be tested empirically, by attempting to recognise concrete occurrences of those verbs from short video sequences. It has an intuitive syntax and semantics, however its main drawback is that it is model dependent. It does not, however, provide a means of reasoning about Event Logic expressions outside of a particular model, about what is true in all models. However, Finite State Temporality, a temporal logic based on regular languages, does provide this capability. In Finite State Temporality intervals are represented as regular languages, meaning that all the properties of regular languages apply to intervals in Finite State Temporality. This makes the method quite powerful, as well as much simpler to deal with theoretically as the theory of regular languages has been well established and explored. If it were possible to express Event Logic in Finite State Temporality it would gain these advantages while still keeping the same syntax and semantics, thus allowing a more powerful method of reasoning about lexical semantics.

1.2 Report Outline

The report is divided into four main sections described below.

1.2.1 Background

Chapter 2 will give a description of Finite State Temporality, FST, the ideas behind it and how its operations work. FST is based on regular languages, so a description will be given of the main properties of regular languages and the various forms in which they may be represented: finite state machines and regular expressions.

The second part of this chapter will describe Event Logic, where it is used and how it has been implemented in other systems.

1.2.2 Translating Event Logic into FST

Chapter 3 describes the main work of the project. It demonstrates how to describe Event Logic formulae in FST. It will show how the Event Logic intervals can be represented as languages, and how the Event Logic operators can be defined in terms of languages. It will show how to implement the inference method, which gives the intervals in a model where an event occurred. It ends by examining some of the open problems relating to this work.

1.2.3 Implementation

Chapter 4 will describe the implementation of a program which can perform event recognition using finite state machines. It takes Event Logic formulae and using the FST techniques from the previous chapter builds a finite state machine which accepts the intervals where the event occurred. It will describe the operation of the program, some of the algorithms used and show examples of the program in use. It compares this method to a previous implementation of Event Logic. Also, the tools used in the implementation will be described.

1.2.4 Conclusion

Finally, Chapter 5 will describe the results of the project. It will examine how sucessfully Event Logic could be expressed in FST, what the advantages and drawbacks of this approach are, as well as some areas for future work.

Chapter 2

Background

2.1 Finite State Temporality

Temporal logics are ways of representing systems where what is true can change over time. Finite State Temporality, FST, is a temporal logic based on the principle that intervals are represented as strings of snapshots [Fer04]. A snapshot is the set of conditions that are true at a particular point in time. More complex intervals can be represented by languages. FST restricts itself to regular languages, as well as allowing FST to exploit the closure and decidability properties of regular languages. This restriction is warranted because more expressive languages, like context free languages, have not been necessary so far in the development and use of FST.

2.1.1 Regular Languages

A regular language is a formal language that can be accepted by a finite state machine. Regular languages can be described by a regular expression or a regular grammar. They occupy the lowest rung in the Chomsky Hierarchy, type 3, making them the most restrictive language in the hierarchy [Eil74].

Regular languages are widely used in text processing and linguistics; and the associated finite state technology is used extensively in hardware design, model checking and logic.

A deterministic finite state automaton[1], DFA, is describe by the the tuple $< \Sigma, S, s_0, \delta, F >$, where Σ is the alphabet; S is the set of states; s_0 is the starting state, an element of S; δ is the transition function $\delta : S \times \Sigma \to S$; and F is the set of accepting states, a subset of S.

A DFA operates by starting in the initial state, s_0, and accepting input symbols until the end of input. For each input symbol it will enter a new state determined by the transition function, δ. If, at the end of input, the state

[1]In this report, the term finite state machine will refer collectively to deterministic and non-deterministic automata.

machine is in an accepting state, then the input string is part of the language encoded by the DFA.

A non-deterministic state machine, NFA, allows more than one transition from state over an input symbol. The transition function becomes $\delta : S \times \Sigma \rightarrow \mathcal{P}(S)$. An epsilon non-deterministic state machine, ϵNFAs, is one that allows transitions over the empty string, ϵ, meaning that it does not consume any input for that transition [SIP96].

NFAs and ϵNFAs are equivalent in power to DFAs, and can be converted to a DFA by powerset construction [ASU86]. This means that any of these can accept any regular language.

2.1.1.1 Regular Language Operations

Regular languages are closed, applying any of these operators yields a regular language, under Kleene star, concatenation, union, intersection and complement.

Union

The union of two regular languages is the set of strings which are in either of the two languages.

$$L_1 \cup L_2 = \{s \mid s \in L_1 \vee s \in L_2\}$$

Concatenation

The concatenation of two regular languages is the language consisting of strings composed by concatenating strings from one language with the strings from another.

$$L \circ L_2 = \{ss_2 \mid s_1 \in L_1, s_2 \in L_2\}$$

For clarity the concatenation operator will be omitted, so $L \circ L\prime$ will be written as $LL\prime$.

Kleene Closure

The closure of an alphabet, denoted by Σ^*, is the set of all possible strings over an alphabet, Σ. It is also called the universal language.

$$\Sigma^* = \bigcup_{i=0}^{\infty} \Sigma^i$$

where $\Sigma^i$is the set of possible strings of length i.

The closure of a language is the set of strings formed from concatenation with itself any number of times. Where L^i represents the concatenation of L with itself i times, and $L^0$is ϵ:

$$L^* = \bigcup_{i=0}^{\infty} L^i$$

The positive Kleene closure of a language is the similar to the closure of a language, except it does not include the empty string.

$$L^+ = \bigcup_{i=1}^{\infty} L^i$$

2.1.1.2 Regular Expressions

Regular expressions describe a regular languages, meaning they are equivalent in power to DFAs. A regular expression is composed of the symbols: ϵ, $a \in \Sigma$, and $\emptyset$. These symbols can be combined using the operators: concatenation, alternation and Kleene closure [Wei05].

Concatenation

$R_1 \circ R_2$ The concatenation of two regular expressions is the concatenation of the regular languages they represent. As with regular language concatenation, the concatenation operator will be omitted.

Alternation

$R_1 \mid R_2$ The alternation or choice of two regular expressions is the union of the languages they represent.

Closure

R^* Closure, the closure of a regular expression is the closure of the language it represents.
An example of a regular expression is $a(b|c)^*$, this accepts strings which consist of an a followed by any number of b or c's.

Regular expressions can be converted to an ϵNFA by Thompson's algorithm[ASU86].

2.1.1.3 Finite State Transducers

Finite state transducers map between regular languages. They extend finite state machines by adding an output language, the transition function now has both input and an output symbols. They map between strings in the input language and strings in the output language; also called the upper and lower languages respectively. The input and output languages may have different alphabets. Finite state transducers support the same operations as finite state machines: union, concatenation, and Kleene closure. However, finite state transducers are not closed under intersection, in the general case [BK03]. In addition, it is possible to compose transducers. The composition of two transducers takes the output of one transducer and uses it as the input to the second transducer.

2.1.2 Basics of Finite State Temporality

In FST the world is described by a set of fluents, Φ. Fluents represent properties of the world which can change over time. It is not necessary for fluents to be atomic, but this report will only consider atomic fluents. Sets of fluents are usually written inside a box instead of between braces. The notation is intended to convey the notion of comic strip, each set of fluents forms a frame in the comic. So, the set of fluents $\{a, b, c\}$ is written as $\boxed{a, b, c}$, and empty set of fluents, $\{\}$, is denoted by $\square$.

For example, given a set of fluents which describe some the actions of a person in their home:

$$\Phi = \{Eating(Bob), Sleeping(Bob), WatchingTV(Bob), WashingUp(Bob), Vacuuming(Bob)\}$$

The snapshot which represents the Bob sleeping is $\boxed{Sleeping(Bob)}$, the snapshot representing the Bob eating while watching TV would be denoted by $\boxed{Eating(Bob), WatchingTV(Bob)}$. Finally, if none of these fluents occurred, it is denoted by the empty snapshot, $\Box$.

Intervals are represented by a string of snapshots. The alphabet of these strings is the powerset of fluents, $\mathcal{P}(\Phi)$[2].

An example of an interval would be:

$$\boxed{Eating(Bob)}\Box\boxed{WashingUp(Bob)}$$

This represents an interval of length three: in the first snapshot the Bob was eating, this was followed by an interval in which none of the fluents held, and in the third snapshot Bob was washing up.

However, it is insufficient to use only strings to describe events. Strings constrain events to fixed lengths, and do not allow for choice (alternation). Instead, events are represented as regular languages.

Using the set of fluents from the previous example, if we wanted to represent the intervals during which Bob was cleaning it might be represented as any interval of positive length, where the Bob was washing up or vacuuming:

$$Cleaning(Bob) = (\boxed{WashingUp(Bob)} \mid \boxed{Vacuuming(Bob)})^+$$

Where $|$ is the regular language operator for alternation, and $^+$ is the positive Kleene closure.

It is important to note the distinction between the empty snapshot, $\Box$, and the empty string, ϵ. $\Box$ is a string of length one which represents an interval of length one in which no fluents held. ϵ is an interval of length zero, and does not convey any information about the world.

2.1.3 Superposition

FST defines an operation called superposition[Fer04]. The superposition of two events corresponds to the co-occurrence of those two events. It is defined as the component-wise union of strings of the same length.

$$L\&L\prime = \bigcup_{n\geq 0}\{(a_0 \cup b_0)\dots(a_n \cup b_n) \mid a_0 \dots a_n \in L, b \dots b_n \in L\prime\}$$

[2]It has been shown in [Fer06] that strings of snapshots are equivalent to strings over the alphabet of fluents with the addition of marker symbol, a 'clock tick' symbol. In other words, strings over the alphabet $\mathcal{P}(\Phi)$ can be converted to strings over $\Phi \cup \wr$, using $\wr$ to represent a clock tick, by a finite state transducer which preserves regularity.

For example, the superposition of $\boxed{a}\boxed{a,b}\boxed{a}$ with $\boxed{a}\boxed{c}\boxed{b}$ would be $\boxed{a}\boxed{a,b,c}\boxed{a,b}$. Superposition is only defined between strings of the same length, superposing of two strings of different length will give the empty language.

2.1.4 Subsumption

FST provides a way to say that a string contains at least as much information as another. One string *subsumes* another if every symbol in the string is a superset of the corresponding symbol in the other string.

$$a_1a_2\dots a_n \unrhd b_1b_2\dots b_m \iff (n=m)\wedge(a \supseteq b_i) : 1 \leq i \leq m$$

One language subsumes another if every string inthe language subsumes a string in the other.

$$L \unrhd L' \iff (\forall s \in L)(\exists s' \in L')s \unrhd s'$$

Looking at a concrete example, it is true that: $\boxed{a,b} \unrhd \boxed{a} \unrhd (\boxed{a}|\boxed{b})$. While $\boxed{a} \unrhd (\boxed{a}|\boxed{b})$ looks incorrect, the relationship holds because we only require that all strings in the left subsume one string in the right.

The *subsumptive closure* of a language, L, is all the strings that subsume a string in L. The superposition of L with the universal language gives its subsumptive closure.

$$L^{\unrhd} = L\&\mathcal{P}(\Phi)^*$$

For example, for a language consisting of one string: $L = \{\boxed{a}\boxed{b}\}$ and a set of fluents $\Phi = \{a,b\}$, the subsumptive closure of L will be all the strings of length two which subsume L.

$$\begin{aligned} L^{\unrhd} \quad = \quad & \{\boxed{a}\boxed{b}, \\ & \boxed{a,b}\boxed{b}, \\ & \boxed{a}\boxed{a,b}, \\ & \boxed{a,b}\boxed{a,b}\} \end{aligned}$$

2.1.5 Constraints

Constraints are a method of doing implication in FST. The constraint $L \Rightarrow L'$ will give the language where L never occurs without L'. In Boolean logic, implication is defined as

$$A \Rightarrow B = \neg(A \wedge \neg B)$$

It is defined similarly in FST, however there are some complications.

Naively, it would be $\overline{L \cap \overline{L'}}$. But the resulting language would be restricted to strings of the same length as $\overline{L \cap \overline{L'}}$. To overcome this, it is necessary to extend the intersection in either direction by concatenating it with the universal language: $\overline{\mathcal{P}(\Phi)^* L \cap \overline{L'} \mathcal{P}(\Phi)^*}$.

However, there is still a problem. The implication should be defined for all the strings that subsume L and L' since subsumption states that those strings contain at least as much information as these ones, the relation should still hold. So the final definition of constraint is as follows:

$$L \Rightarrow L' = \overline{\mathcal{P}(\Phi)^* L^{\unrhd} \cap \overline{L'^{\unrhd}} \mathcal{P}(\Phi)^*}$$

Two of the typical uses of constraints are to define the $\phi - consistent$ and $\phi - bivalent$ languages.

The $\phi - consistent$ language does not allow a fluent and its negation to occur at the same time. For some fluent ϕ, where $\emptyset$ is the empty language, the $\phi - consistent$ language is given by the constraint:

$$\boxed{\phi, \neg\phi} \Rightarrow \emptyset \tag{2.1}$$

The $\phi - bivalent$ language demands that either ϕ or its negation must occur in every interval:

$$\Box \Rightarrow \boxed{\phi} \mid \boxed{\neg\phi} \tag{2.2}$$

2.2 Event Logic

[Sis01] is an attempt to describe the lexical semantics of verbs in a way that can be empirically tested. It takes simple verbs such as *pick-up* and *put-down* and describes them using force-dynamics and Event Logic. The accuracy of these definitions can be tested by performing event recognition on video sequences containing these actions. This is in contrast to previous attempts to represent lexical semantics, where the semantics were not fully specified and representations were derived intuitively. Force-dynamics [Tal88] describe the relationships between between objects in terms of *contact*, *support*, and *attachment*. This is in contrast to previous approaches to model reconstruction which used motion profile, i.e. the velocity, rotation, etc. of the objects[SM96]. Force-dynamics coincide more closely than motion profile with people's intuitive idea of events: the speeds and positions of the objects are less important than how the objects are connected or supported. Force-dynamics also allow a more concise definition of events than would be possible using motion profile.

This attempt to ground lexical semantics in perception is continued in [FGS02]. However, instead of using the hand written definitions, event definitions are learned from video sequences. Only a subset of Event Logic is used in these definitions, called And-Meets-And. And-Meets-And, AMA, is a severely restricted subset of Event Logic, which only uses the sequencing, *meets*, and co-occurrence, *and*, operators, as well as only allowing them in certain combinations. However, despite these restrictions, learned event definitions performed almost as well as the handwritten ones using the full Event Logic.[3]

2.2.1 Leonard

Leonard is the implemented system described in [Sis01] which uses Event Logic to perform event classification from video. It takes a sequence of images, performs event recognition and outputs labels for the observed events. It is able recognise events corresponding to the simple verbs: pick-up, put-down, stack, unstack, move, assemble and disassemble.

It has three main stages: image segmentation and tracking, model reconstruction and event classification. The segmentation stage takes in a sequence of images from a camera and produces a sequence of scene descriptions, i.e. a list of objects it detected in each scene. It uses color and motion based segmentation, described in [SM96].

The model reconstruction stage uses force dynamics to interpret the scenes. Force-dynamics exploit some common sense knowledge of the world to make an interpretation of the scene, e.g. if an object is not falling, then it must be supported. The model reconstruction outputs the intervals during which force-dynamic relations hold. The force-dynamic relations it recognises are: *Supported(x)*, *Supports(x,y)*, *Contacts(x,y)*, and *Attached(x,y)* .

The event classification stage takes in the force-dynamic relations and uses Event Logic to infer occurrences of the compound event types. The event classification is able to recognise simultaneous occurrences of events, as well as the non-occurrence of events.

2.2.2 The Allen Relations

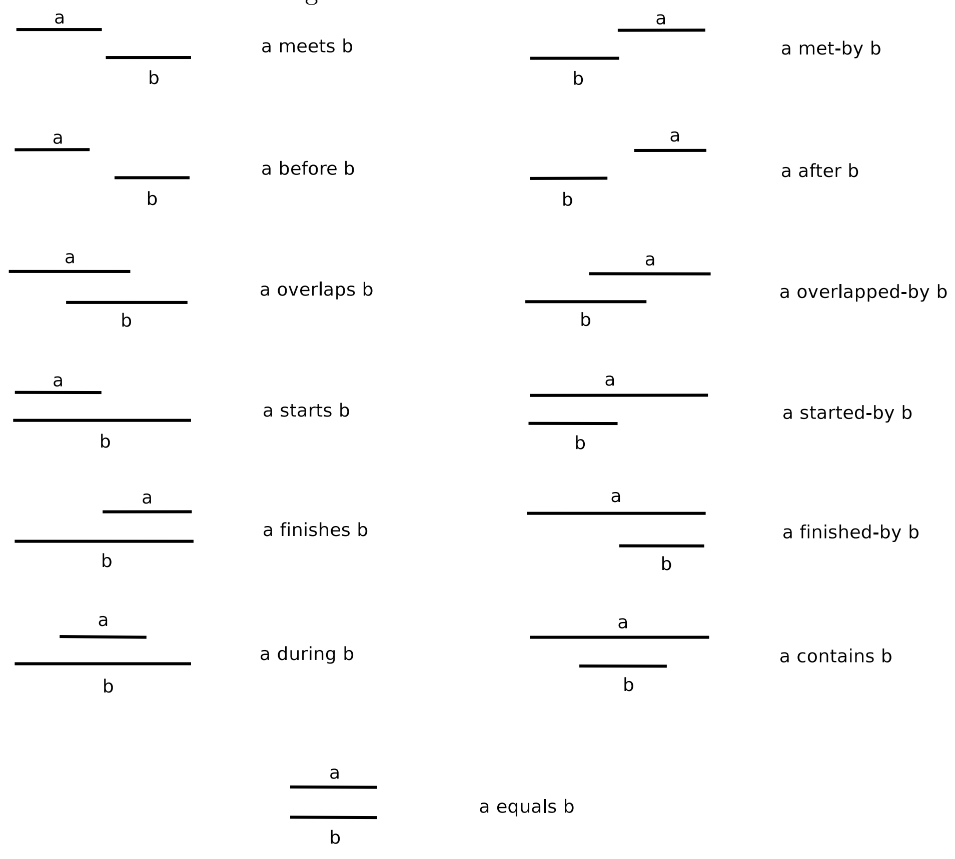
Event Logic uses the Allen relations to specify the allowed relations between events. The Allen relations are a set of interval relations described by [All83], and illustrated in 3.1. They describe all the possible relations between two intervals. They also have the property that they are distinct, it is impossible for a pair of intervals to be described by more than one Allen relation. The Allen relations are particularly useful in AI because they can express knowledge about incomplete and qualitative data. [KHMW99]

2.2.3 Event Logic Syntax and Semantics

Event logic provides a way to describe complex events in terms of primitive events. It specifies what must have taken place for an event to have occurred: the primitive events that must occur, and how they may occur in relation to

[3]Learned definitions out-performed the hand coded definitions from [Sis01], but performed slightly worse than a revised set of hand coded definitions written specifically to work with the particular model reconstruction algorithm.

Figure 2.1: The Allen Relations



one another. It does not provide a way to specify the particular duration of the primitive events.

For example, here is a simplified definition of a *pick-up* event, where x picks up y from z. It describes the *pick-up* event as beginning with a state where y is supported by contact with z, which is immediately followed by a state where y is supported by x and attached to x. The $\wedge$ operator denotes coincidental occurrence, the two events started and ended at the same time, ; is the sequencing operator, denoting one event having happened directly after another.[4]

$$\mathit{PickUp}(x,y,x) = (\mathit{Supports}(z,y) \wedge \mathit{Contacts}(z,y)); ((\mathit{Supports}(x,y) \wedge \mathit{Attached}(x,y))$$

The basic elements of Event Logic formulae are primitive events, in LEONARD these are the force-dynamic relations. Primitive events are assumed to be *liquid,* also called *homogeneous* [Sho87]. A liquid event which is true during a certain interval is also true in all the sub-intervals of that interval.

Event Logic expressions are made up from primitive events and the compound events. The Event Logic operators are: $\neg\Phi$, $\Phi \vee \Psi$, $\Phi \wedge_R \Psi$, $\Diamond_R \Phi$, where Φ and Ψ are compound Event Logic expressions and R is a set of Allen Relations. Their semantics will be explained in the next section.

2.2.3.1 Semantics of Event Logic

- $\Phi@i$ *Coincidence*: denotes that the Event Logic expression, Φ, directly coincided with the interval i. For example, $\mathit{PickUp}(\mathit{hand}, \mathit{redblock}, \mathit{greenblock})@[5, 20)$ means that a pick-up event occurred in the interval from Frame 5 up to Frame 20.[5]

- $\neg\Phi@i$ *Negation:* Φ did not occur at interval i. It may be the case that Φ overlapped with i, but $\neg\Phi@i$ is only true if Φ did not coincide with i.

- $\Phi \vee \Psi@i$ *Or*: Either Φ or Ψ occurred at i.

- $\Phi \wedge_R \Psi@i$ *And:* Both Φ and Ψ occurred, how they may occur in relation to each other is given by R, a set of Allen relations. It is important to note that if Φ occurred at interval j and Ψ occurred at interval k, then i is the the smallest interval that contains both j and k. If R is not specified the default relation, *equals*, is assumed.

- $\Diamond_R\Phi@i$ *Tense:* Denotes that Φ occurred at some interval j, such that j and i are related to each other by the Allen relation $r \in R$. This could be used, for example, to specify that Φ occurred some time in the past by using $\Diamond_{\{before, meets\}}\Phi$. If R is not given, the default is an interval which overlaps with i, i.e. r is one of equals, overlaps, overlapped-by, starts, started-by, finishes, finished-by, during, or contains.

[4] $\Phi; \Psi$is short hand for $\Phi \wedge_{\{meets\}} \Psi$, or Φ meets Ψ.

[5] The end-points of an interval may be open or closed, open intervals do not include the end point and are denoted by parenthesis,); closed end-points include the end-point and are denoted by square brackets,].

2.2.4 Implementation of Event Logic

The interval algebra described by [All83] does not use the numerical values of the end points to represent intervals. Instead intervals and their relations are represented as constraint networks. In a constraint network nodes represent events, and arcs represent the relations between them. It is possible to reason about the network using constraint propagation [All84].

However, because Event Logic deals with concrete values for the end points of intervals it uses an inference method based on spanning-intervals.

For primitive events, which are liquid, if $\Phi@i$, then Φ is also true in all the sub-intervals of i. However, this is not the case for compound event types. However, any compound expression made up of two primitive events will be *semi-liquid*. A semi-liquid event allows the end-points of the interval to hold over a certain range of values. For example, if $\mathit{Supports}(\mathit{redblock}, \mathit{greenblock})@[0,7)$ and $\mathit{Attached}(\mathit{hand}, \mathit{greenblock})@[4,10)$, then $\mathit{Supports}(\mathit{redblock}, \mathit{greenblock}) \wedge_{\{\mathit{overlaps}\}}$ $\mathit{Attached}(\mathit{hand}, \mathit{greenblock})$ will hold at the interval $[0,10)$. But it will also hold in any of the sub-intervals which contain $[3,8)$, as this is the interval which satisfies the *overlaps* relation. A *spanning interval* describes a semi-liquid event. In this example the spanning interval would be [[0,3),[8,10]).

Spanning intervals provide a way of efficiently representing the intervals in which a compound event holds. A primitive event can hold in quadratically many sub-intervals, however the spanning interval representation provides a means of reasoning about when compound events hold without representing the sub-intervals directly.

Chapter 3

Translating Event Logic into FST

This project shows that Event Logic can be expressed in FST. FST offers advantages over Event Logic in that it allows reasoning independent of a model. Event Logic only provides semantic entailment, which can tell if a formula is true in a given model. However, if Event Logic formulae are represented in FST, it would be possible to make deductions about what is true in all models.

One of the advantages of Event Logic over other event recognition methods, such as Hidden Markov Models, is that it makes introspection easier [FGS02]. The definition of an event is not encoded as weights in the network, but in a semantically meaningful form. Encoding events as regular languages takes this a step further. It allows one to use all the existing knowledge about regular languages to examine event definitions.

3.1 Basics

The Event Logic formulae define the truth conditions that must hold during an event. In FST, an Event Logic formula will be represented as the language that represents occurrences of the event.

Consider an Event Logic formula for making tea:

$$MakeTea = BoilKettle; PourWater; AddTea; (AddMilk \vee AddSugar)$$

The FST representation of this would be the language:

$$MakeTea = \boxed{BoilKettle}^{+}\boxed{PourWater}^{+}\boxed{AddTea}^{+}(\boxed{AddMilk}^{+} \mid \boxed{AddSugar}^{+})$$

This language will be represented as a finite state machine. Deciding if this event occurred in a model becomes simply a matter of running the state machine on the model, if the state machine subsumes the model then the event has occurred, other it has not.

It is important to note that these are the minimum requirements for an event to happen. For example $\Box^+$ does not require any fluent to hold, it will accept when nothing or anything occurred. To explicitly state that something did not occur it is necessary to use the negation of that language, $\neg L$, which will be defined later.

To be able to represent all Event Logic formulas in FST, not just this simple example requires the ability to:

- represent the Allen relations as regular languages
- implement all of the Event Logic operators: *negation, or, and, tense*
- be able to determine the truth value of an Event Logic formula for a particular model using FST.

3.2 The Allen Relations as Regular Languages

The Allen relations can be implemented in FST by using superposition along with the standard regular language operators. The Allen relations have been expressed in FST before, [Fer06], however this only defines relations between strings, not between languages. The first seven relations are defined here, the remaining six are the inverses of these, and can be defined in terms of them, e.g. $after(x, y) = before(y, x)$.

Figure 3.1: The Allen Relations between languages

Allen Relation	FST
$equal(L_1, L_2)$	$L_1 \& L_2$
$before(L_1, L_2)$	$L_1 \Box^+ L_2$
$meets(L_1, L_2)$	$L_1 L_2$
$overlaps(L_1, L_2)$	$((L_1 \Box^+) \& (\Box^+ L_2)) - (L_1 \Box^* L_2)$
$starts(L_1, L_2)$	$(L_1 \Box^+) \& L_2$
$finishes(L_1, L_2)$	$(\Box^+ L_1) \& L_2$
$contains(L_1, L_2)$	$L_1 \& (\Box^+ L_2 \Box^+)$

- $equal(L_1, L_2) = L_1 \& L_2$ This comes directly from the definition of superposition. $L_1$and L_2 must start and end at exactly the same time.
- $before(L_1, L_2) = L_1 \Box^+ L_2$ The $before$ relation specifies there is must be an interval of length at least one between the two events. The regular language for this relation inserts empty snapshots between the two languages. The empty snaphots demand that an interval occur, but do not require that any of fluents hold at that point.

- $meets(L_2, L_2) = L_1 L_2$ The *meet* relation requires that the one interval begin immediately after the end of the other. This is expressed as the concatenation of the two languages.
- $overlaps(L_1, L_2) = ((L_1 \Box^+) \& (\Box^+ L_2)) - (L_1 \Box^* L_2)$ The *overlaps* relation demands that the second interval must begin sometime after the first. This is achieved by prepending empty intervals to the beginning of the second interval. Similarly, the second interval must end after the first, hence empty intervals are appended to the first interval. However, just using the superposition of these two padded intervals causes a problem. It will accept intervals where $L_1$is $before$ or *meets* L_2, as the padding at the beginning and end is allowed to extend out to any length, potentially past the end of L_1. Hence, it is necessary to subtract the strings where no part of $L_1$and L_2 overlap. This is given by the concatenation of L_1 and L_2 with zero or more empty intervals between them.
- $starts(L_1, L_2) = (L_1 \Box^+) \& L_2$ For L_1 to *start* L_2 they must begin at the same time and L_1must end before L_2. This is defined as the superposition of $L_1$and L_2, but with at least one empty interval following L_1.
- $finishes(L_1, L_2) = (\Box^+ L_1) \& L_2$ The $finishes$ relation is similar to *starts*, except that the first interval must start after the second, and they must end at the same point. Hence, empty intervals are prepended to the beginning of L_1.
- $contains(L_1, L_2) = L_1 \& (\Box^+ L_2 \Box^+)$ If an interval starts before and finishes after another, it *contains* the interval. To obtain the regular language for this relation, L_2 is padded on both sides with empty intervals, and then superposed with L_1.

3.3 Event Logic Operators

The set of fluents in FST are the set of primitive Event Logic symbols. However, it will be seen that it is necessary to have fluents that represent the non-occurrence of primitive events. For each primitive event ϕ there will be a corresponding fluent $\sim \phi$ which denotes its non-occurrence. Event Logic only allows finite sets of symbols, making the alphabet $\mathcal{P}(Symbols \cup NegatedSymbols)$, will also be finite which is a requirement for regular languages.

3.3.1 Primitive Events

Primitive events in Event Logic formulae must hold for an interval of length at least one, they also have the property of liquidity which makes them true in all sub-intervals. Primitive events will be represented in FST as the language:

$$L(\phi) = \boxed{\phi}^+$$

where ϕ is the primitive event in question.

All the strings in $\boxed{\phi}^+$ will be of length greater than or equal to one, from the definition of positive Kleene closure. For a string in $\boxed{\phi}^+$, all the non-empty substrings of it will also be a member of $\boxed{\phi}^+$ as they can only consist of a sequence of $\boxed{\phi}$'s. This satisfies the requirement that primitive events are liquid.

3.3.2 Or

Or in FST becomes the alternation of the languages for Φ and Ψ. This language accepts either Φ or Ψ.

$$L(\Phi \vee \Psi) = L(\Phi) \mid L(\Psi)$$

3.3.3 And

The *and* operator takes a set of Allen relations and allows the two intervals to occur in any of those relations to each other. This corresponds to the regular language which is the alternation of each of the Allen relations between the two languages.

$$L(\Phi \wedge_R \Psi) = r_1(L(\Phi), L(\Psi)) \mid r_2(L(\Phi), L(\Psi)) \mid \cdots \mid r_n(L(\Phi), L(\Psi)) : r \in R$$

3.3.4 Negation

Negation is more difficult to represent in FST. It is necessary to introduction fluents to explicitly represent the non-occurrence of a fluent. Each fluent ϕ will have a corresponding negated fluent $\sim \phi$. In this system $\boxed{\phi}$ represents an interval where ϕ was true, $\boxed{\sim \phi}$ denotes an interval where ϕ was not true, and $\Box$ does not specify whether ϕ held or not.

To define the negation of a language is it necessary to have the *complete and consistent language* L_C. This is the language of all possible strings which obey certain rules about negation of fluents. L_C should be complete; either a fluent or it's negation should be present in each symbol. The $\phi - complete$ (2.1) language for a fluent gives the language where ϕ or $\sim \phi$ are present in every symbol. It will also contain all the strings which subsume the ones which satisfy this constraint. The intersection of the $\phi - complete$ languages for all fluents gives the language where every symbol contains either a fluent or its negation.

$$L_{\Phi complete} = \bigcap_{\phi \in \Phi} (\Box \Rightarrow \boxed{\phi} \mid \boxed{\sim \phi}) \tag{3.1}$$

However the $\phi - complete$ language allows strings in which a fluent and its negation co-occur. The complete and consistent language should reject strings any strings which contain a fluent and its negation in the same symbol. The $\phi - \mathit{bivalent}$ (2.2) language implements the constraint that only allows a fluent or its negation to occurr at the same time. The $\phi - \mathit{bivalent}$ language for fluents is given by:

$$L_{\Phi bivalent} = \bigcap_{\phi \in \Phi} (\boxed{\phi, \sim \phi} \Rightarrow \emptyset)$$

The complete and consistent language L_C is given the by the intersection of the $\Phi - \mathit{complete}$ and $\Phi - \mathit{bivalent}$ languages.

$$L_C = L_{\Phi complete} \cap L_{\Phi bivalent}$$

The negation of a language, L, is the set of strings that do not contain L. This is given by the complement of L. However, the negation should not contain the strings which subsume L. For example, if $L = \{\boxed{\phi}\}$, the complement of L should not include strings such as $\boxed{\phi, \psi}$. The subsumptive closure of L, $L^{\trianglerighteq}$, will give all give all the strings that subsume L. The negation can be obtained by substracting the subsumptive closure from all the consistent and complete strings.

So, the negation of a language, L, will be all the complete and consistent language less all the strings that subsume L:

$$L(\neg\Phi) = L_C - (\Box^* L(\Phi)^{\trianglerighteq} \Box^*)$$

3.3.5 Tense

The *tense* operator holds if there exists an interval in relation to Φ, for a given set of Allen relations. It is expressed similarly to the *and* operator. Another interval, σ, that can hold at any point is introduced. It is necessary to make each σ interval distinct, as each σ can only occur in a certain relation to Φ.

$$L(\Diamond_R \Phi) = r_1(L(\Phi), \sigma_1) \mid r_2(L(\Phi), \sigma_2) \mid \cdots \mid r_n(L(\Phi), \sigma_n) : r \in R \tag{3.2}$$

where $\sigma_i = \boxed{\mathit{begin\_i}} \Box^* \boxed{\mathit{end\_i}}$.

However, there is a problem with this method. The intended semantics of the *tense* operator are that it holds at an interval i, $\Diamond_R \Phi @ i$, if for some interval j, $\Phi @ j$ and $i\ r\ j$ hold. The interval i corresponds to the interval σ in the FST definition. However, the FST definition will return the interval which includes i and j.[1] It should return the interval i where σ occurs, but instead returns the interval $r(L(\Phi), \sigma)$. This problem stems from the fact $L(\Diamond_R \Phi)$, as well as the FST definitions of the Event Logic operators, do not actually give the particular intervals in which that operator held. They give all the strings that can satisfy

[1]This is actually $\mathit{Span}(i, j)$, the smallest interval which contains both i and j.

that operation. However, for the *tense* operation the strings in where it occurs do not coincide with the intervals where it holds.

One possible solution for problem would be to define the *tense* operator as a finite state transducer which accepts the language given in (3.2) and outputs only the interval which is part of σ. However, using finite state transducers as a representation introduces another problem, we require the *negation* operator. Unfortunately intersection is required in its definition, but intersection of finite state transducers is only regular between equal length relations[KK94]. The relations for the *tense* operator will only be of equal length if the Allen relation is *equals*, namely $R = \{equals\}$.

At present, the implementation uses the definition of *tense* in (3.2), while this works in most common cases, the results of this operator can not be used in further computation due to the reasons stated above. For example, $(\Diamond_{\{before\}}A) \wedge B$, when entered into the program[2] will produce incorrect results. It should accept strings where B coincides with an interval which is before with A. However, the program will only accept strings where B overlaps with A.

A correct method of representing this is an interesting problem for future research.

3.4 Event Recognition

This section will discuss how to use the finite state representation of events to perform event recognition. Inference in Event Logic gives the intervals in a model where an event occurred. The inference method for the FST representation of events is much simpler than for than for the Event Logic implementation in LEONARD. In FST, the model, M, becomes a string which is the sequence of observations. For all implemented systems M will be finite, which makes it a regular language consisting of one string. The model must be complete, i.e. it must be a member of (3.1) for the representation of negation to work correctly.

Checking if an event occurred in the model is a matter of checking whether M subsumes the language representing the event. It is necessary to pad the event on either side with empty intervals, to allow the event to start and finish anywhere in the model, i.e. it need not directly coincide with the set of observations, for example if another event occured in the background, entirely unrelated to our event it should not affect it. A more concrete way of seeing this is to look at an event, for example Bob boils water, this event happens regardless of the fact that Mary is not boiling water; the two are unrelated.

$$M \models \Phi = M \trianglerighteq (\Box^* L(\Phi) \Box^*) \tag{3.3}$$

Event Logic provides an inference mechanism $\mathcal{E}(M, \Phi)$ will return all the intervals where Φ occurred in M. In FST, inference can be described as all the

[2] $(and\ (tense\ A\ (before))\ B)$

substrings of M subsume $L(\Phi)$. This is same as the intersection of M with the subsumptive closure of $L(\Phi)$:

$$\mathcal{E}(M, \Phi) = factors(M) \cap (L(\Phi)^{\trianglerighteq}) \tag{3.4}$$

where $factors(M)$ are the substrings of M.

3.5 Future Work

One of the shortcomings of the FST representation is that Event Logic can represent intervals of infinite length, which FST cannot describe. Regular languages can only contain strings of finite length, making FST unable to represent infinite intervals. One possible solution would be to expand FST to include ω-regular languages. ω-regular languages are a generalisation of regular languages which include infinite strings [Muk]. They are described by Buchi automata. A Buchi automaton is similar to a finite state machine, except that instead of having a set of accepting *Final* states, a string is accepted if it causes the automaton to go into a *Good* state. Entering a *Good* state need not coincide with the end of the string, as in finite state machines.

Similarly, FST cannot represent intervals over real numbers, which Event Logic is potentially able to.

In real systems neither of these issues arise as they only produce a finite set of observations, and with a certain granularity.

3.6 Conclusion

This chapter has shown a means of representing Event Logic formulae in FST. It has been demonstrated how to encode the Allen relations as regular languages. A method for representing each of the Event Logic operators as an FST operation on regular languages has been shown. Problems with the representation of the *tense* operator were described as well as potential solutions. Some of the drawbacks of the FST method and interesting areas for future research have been shown.

Chapter 4

Implementation

A program has been implemented which uses the FST representation of Event Logic formulae to perform event recognition.

It accepts descriptions of event types as Event Logic formulae along with a model and outputs whether the event occurred in the model. This is acheived by converting the event formulae to a finite state representation, followed by checking if the finite state machine can subsume the model. If it can subsume the model then the event has occurred in the model, otherwise it was not observed.

The conversion of an event formula to a finite state machine requires a number of different algorithms. The regular expression operators: concatenation, alternation and closure, are implemented using Thompson's algorithm. Determinisation of ϵNFAs to DFAs is acheived by the subset construction algorithm. Finite state machines are manipulated using intersection, inverse, complement and minimisation. These algorithms are all clearly described throughout the regular language literature as they are the foundations of the techniques used, they are implemented in this program as in [ASU86] and [SIP96]. The former is a more concrete implementation-driven exposition whereas the latter spends spends much of the book detailing the proofs of regular languages.

The program fully implements the method of producing a finite state machine from an Event Logic expression described in 3.3 on page 15. The inference method used to tell if an event holds in a model has also been implemented.

The program is implemented in Scheme using the Bigloo Scheme compiler and standard libraries. State diagrams of the finite state machines are produced using the Graphviz package.

4.1 Event Logic syntax

Here we present the grammar for Event Logic formulae that is implemented in the program. They are represented as symbolic expressions, which is very convenient since Scheme is able to parse symbolic expressions, thus avoiding the need to write a parser.

Figure 4.1: Program Overview

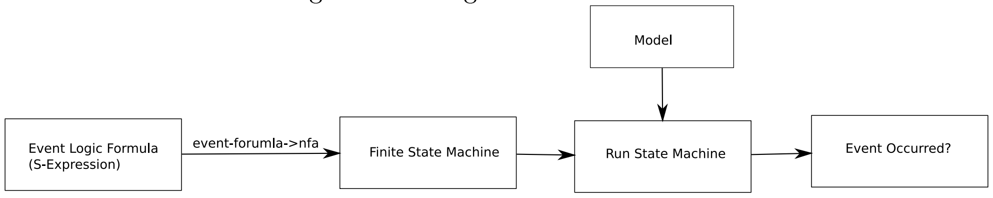


Figure 4.2: Grammar for Event Logic formulae

$$\begin{array}{rcl}
\langle exp\rangle & \rightarrow & (not\ \langle exp\rangle) \\
& | & (or\ \langle exp\rangle\ \langle exp\rangle) \\
& | & (and\ \langle exp\rangle\ \langle exp\rangle) \\
& | & (and\ \langle exp\rangle\ \langle exp\rangle\ \langle relations\rangle) \\
& | & (tense\ \langle exp\rangle) \\
& | & (tense\ \langle exp\rangle\ \langle relations\rangle) \\
& | & fluent \\
\langle relations\rangle & \rightarrow & (\langle allen\rangle\ \langle allenlist\rangle) \\
\langle allenlist\rangle & \rightarrow & \langle allen\rangle\ \langle allenlist\rangle \\
& | & \epsilon \\
\langle allen\rangle & \rightarrow & equals\ |\ before\ |\ after\ |\ meets\ |\ met-by \\
& & |\ overlaps\ |\ overlapped-by\ |\ contains \\
& & |\ during\ |\ starts\ |\ started-by\ |\ ends \\
& & |\ end-by
\end{array}$$

4.2 Building the Finite State Representation

The event formula is converted to a finite state machine by implementing the methods described in 3.3 on page 15. The program performs a post-order traversal of the event formula expression, at each node using the methods described in 3.3 on page 15 to produce an ϵNFA. The alphabet for the ϵNFA can be given beforehand or it can be taken to be the set of fluents used in the event formula.

This ϵNFA is then determinised to produce a DFA. The DFA is minimised, first by removing states that are not a path to an accepting state, this includes unreachable-states, by removing duplicate edges and then by coalescing equivalent states. Two states in DFA are equivalent if for all symbols in the alphabet there is a transition from those states to the same state.

4.3 FST Algorithms

4.3.1 Superposition

The superposition of two state machines is implemented as described in [Fer04]. The method takes two DFAs, D_1 and D_2, and produces a DFA, D_3, that is their superposition. The states of D_3 will be pair of states from D_1 and D_2. The intuition behind this is that D_3 is the result of running D_1 and D_2in parallel, it accepts symbols which are the union of the symbols accepted by D_1 and D_2.

The starting state of D_3 will be $(q0_0, q0_1)$, where $q0_0$ and $q0_1$ are the start states of D_1 and D_2 respectively.

The final states of D_3 will be the pairs which are made up of final states from $D_1$and D_2 This corresponds to $QFinal_1 \times QFinal_2$ where $QFinal_1$ and $QFinal_2$ are the final states of D_1 and D_2 respectively.

The transitions of D_3 will be the set of transitions:

$$(q_1, q_2)\underrightarrow{s_1 \cup s_2}(r_1, r_2)$$

for all the transitions $q_1 \underrightarrow{s_1} r_1$ and $q_2 \underrightarrow{s_2} r_2$ of D_1 and D_2 respectively.

One note about this implementation of superposition, it will produce a state machine with a set of states $Q_1 \times Q_2$, where Q_1 and Q_2 are the set of states of D_1 and D_2 respectively. However, a large number of these states will be unreachable. This happens because building the transitions of D_3 do not take into account whether it is actually possible for both D_1 and D_2 to enter those states at the same time. These unreachable states can be removed by repeatedly removing the states that are not the destination any of transition, until no more states can be removed.

4.3.2 Subsumption

The subsumptive closure is used to check if a model subsumes an event. It is defined, as follows, in terms of superposition with the universal language:

$L^{\trianglerighteq} = L\&\mathcal{P}(\Phi)^*$

It is only necessary to check if the model is a member of the subsumptive closure of the event, since the subsumptive closure of a language will give all the strings that subsume it. As the model is just a string and not a language, this can be implemented by running the DFA of the event with the model as input.

4.4 Examples of use

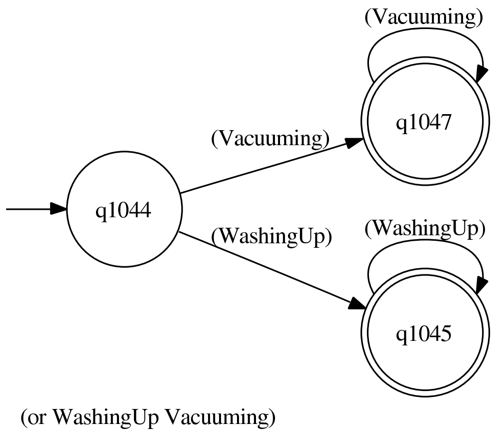
This section will show the state diagrams of the state machines produced for various event formulae. The finite state machines presented here have the minimum set of fluents that must hold for an event to have occurred, their subsumptive closure will be used when accepting input; although for clarity we omit the latter, it does not add information and makes the machines significantly more complicated.

Using the set of fluents from 2.1.2 on page 5:
$\Phi = \{Eating, Sleeping, WatchingTV, WashingUp, Vacuuming\}$
The event formula for a $Cleaning$ event would be:

$$Cleaning = (or\ WashingUp\ Vacuuming)$$

The state machine can be seen in 4.3.

Figure 4.3: Example 1: *Cleaning*

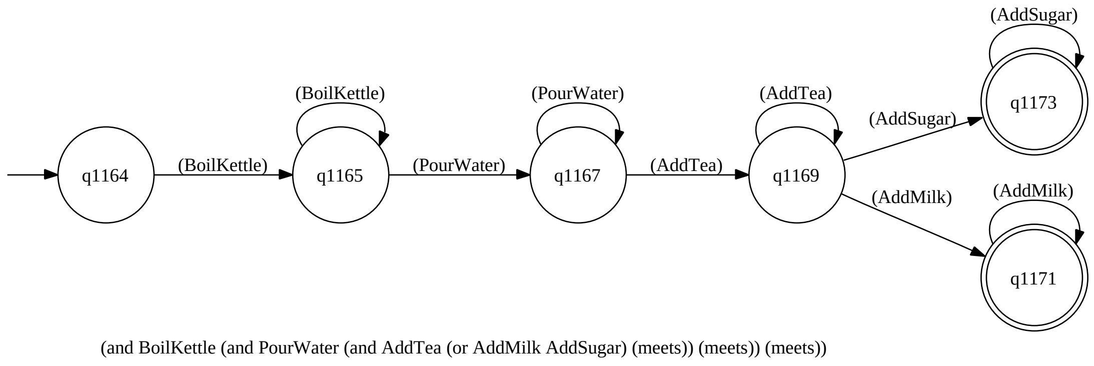


We can now see a more interesting example from 3.1 on page 13:

$$MakingTea = (and\, BoilKettle\, (and\, PourWater\, (and\, AddTea\, (or\, AddMilk\, AddSugar))))$$

The state machine is illustrated in 4.4.

Figure 4.4: Example 2: *MakingTeaCleaning*

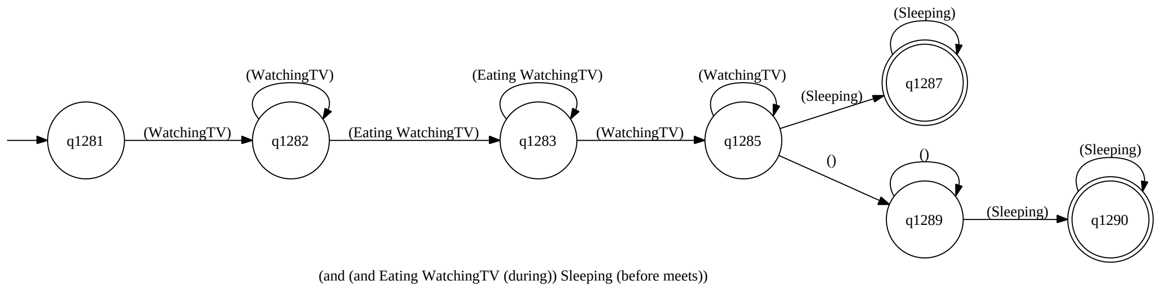


A more complex example which could represent the normal evening activities of our character Bob, who watches TV while eating dinner and some time after that goes to sleep.

$Evening = (and\ (and\ Eating\ WatchingTV\ (during))\ Sleeping\ (before\ meets))$

The state machine can be seen in 4.5.

Figure 4.5: Example 3: Evening



An example which makes use of negation, as follows:

$Leisure = (and\ (or\ Sport\ Watching)\ (not\ (or\ Vacuuming\ Washing)))$

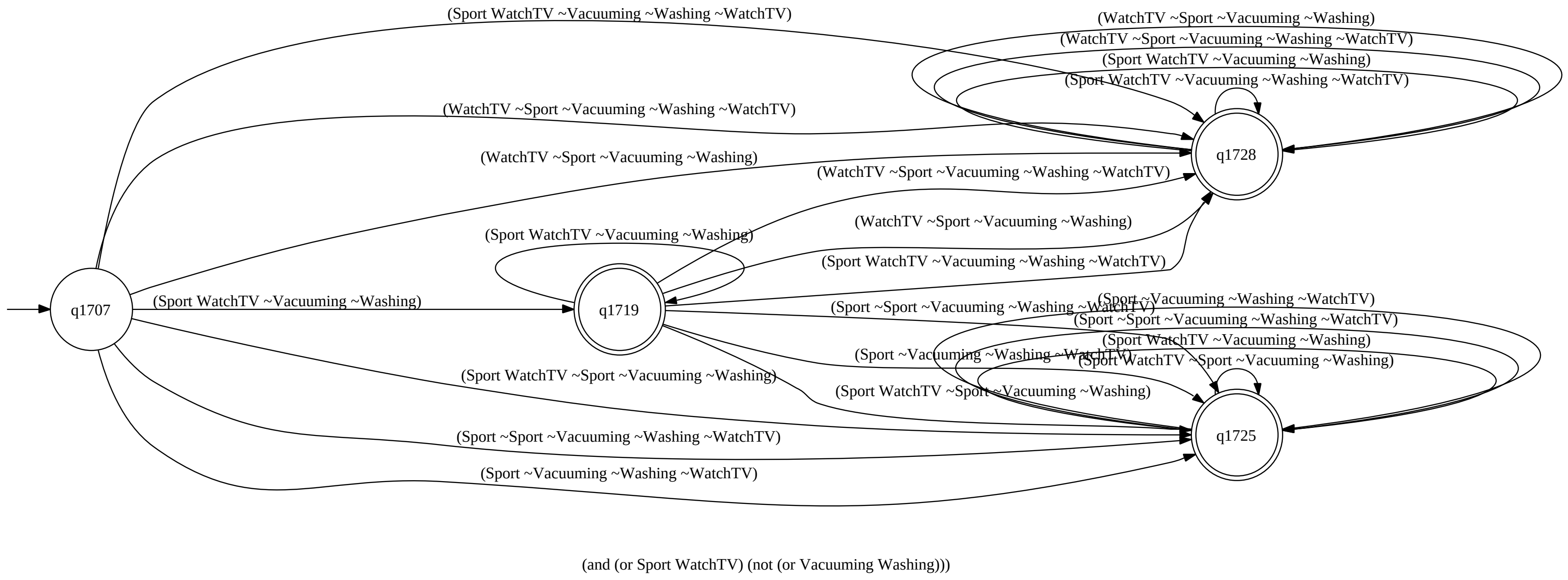
can be seen in 4.6 on the next page.

4.5 Analysis of Implementation

The implementation is able to correctly convert Event Logic formulae into finite state machines, with the exception of certain uses of the *tense* operator described in 3.3.5 on page 17, it is able to use the finite state representation to recognise event occurrences.

However, one part of the program which is unimplemented is the inference method, $\mathcal{E}(M, \Phi)$, which will give all the intervals in which an event occurred. The program currently only gives the first occurrence of an event, or false if the event did not occur. This was due to a shortage of time. The program required a large amount of implementation work, and to completely finish it would require several more weeks. If further time were available it would be desirable to implement $\mathcal{E}(M, \Phi)$, in addition to a method of emitting labels for occurrences of event types. This could be implemented by taking the finite state machine representing all the occurrences of the event in the model and using that to create a finite state transducer. The input language of the transducer would be the event occurrences, and the output language would labels for the beginning and end of events.

Figure 4.6: Example 4: Leisure


q1707
q1719
q1728
q1725
(Sport WatchTV ~Vacuuming ~Washing ~WatchTV)
(WatchTV ~Sport ~Vacuuming ~Washing ~WatchTV)
(WatchTV ~Sport ~Vacuuming ~Washing)
(Sport WatchTV ~Vacuuming ~Washing)
(Sport WatchTV ~Vacuuming ~Washing)
(Sport WatchTV ~Sport ~Vacuuming ~Washing)
(Sport ~Sport ~Vacuuming ~Washing ~WatchTV)
(Sport ~Vacuuming ~Washing ~WatchTV)
(WatchTV ~Sport ~Vacuuming ~Washing)
(WatchTV ~Sport ~Vacuuming ~Washing ~WatchTV)
(Sport WatchTV ~Vacuuming ~Washing)
(Sport WatchTV ~Vacuuming ~Washing ~WatchTV)
(WatchTV ~Sport ~Vacuuming ~Washing ~WatchTV)
(WatchTV ~Sport ~Vacuuming ~Washing)
(Sport WatchTV ~Vacuuming ~Washing ~WatchTV)
(Sport ~Sport ~Vacuuming ~Washing ~WatchTV)
(Sport ~Vacuuming ~Washing ~WatchTV)
(Sport WatchTV ~Sport ~Vacuuming ~Washing)
(Sport ~Vacuuming ~Washing ~WatchTV)
(Sport ~Sport ~Vacuuming ~Washing ~WatchTV)
(Sport WatchTV ~Vacuuming ~Washing)
(Sport WatchTV ~Sport ~Vacuuming ~Washing)
(and (or Sport WatchTV) (not (or Vacuuming Washing)))

4.5.1 Comparison to Leonard

This section will compare the FST implementation to the original implementation of Event Logic in Leonard which was based on spanning intervals, 2.2.1 on page 9.

The FST implementation has an advantage over the Event Logic inference mechanism implemented in Leonard. In FST the bulk of the computation is done when the event formula is compiled into a finite state machine, instead of at query time when it is compared to the model, as in Leonard. This would be advantageous for a system where the same event definitions are reused multiple times, or a system where run time performance was a high priority and finite state machines could be pre-compiled beforehand.

One of problems that the implementation runs into is that the size of the alphabet for the state machines is exponential, $2^{2|\Phi|}$, where Φ is the set of fluents. It is twice the size of Φ because every fluent must have a corresponding negation. This can lead to huge state machines for complex event formulae. Combined with the fact that some of the finite state algorithms such as determinisation have quadratic complexity, this can lead to problems such as unacceptable run times and stack overflow.

There are several approaches that could be taken to alleviate this problem. One possibility is to work throughout the program with ϵNFAs. At several points in the program the ϵNFAs are converted DFAs because the algorithm for superposition has only been described for DFAs, another interesting future research topic. Converting from an ϵNFA to a DFA can exponentially increase the number of states [ASU86]. So working only with ϵNFAs could potentially cause a huge decrease the size of the state machines. Another possibility would be to use a generalised NFA, GNFA. A GNFA is similar to a ϵNFA except that transitions are labeled with regular expressions instead of symbols [SIP96][1]. If GNFAs were used to label the transitions with simple regular expressions representing sets of symbols it could greatly reduce the number of transitions. Reducing the number of transitions is a particularly pertinent problem for state machines with large alphabets, as the complete state machine will have a transition for each symbol in the alphabet.

4.6 Tools

4.6.1 Bigloo Scheme

The program is implemented in Scheme using the Bigloo Scheme compiler. Scheme is a functional programming language. It is a dialect of lisp widely used in academia and teaching. The main properties of scheme are tail recursion, lexical scoping, lexical closures and continuations as first class objects. The current Scheme standard is R^5RS [KCE98], with R^6RS on the way.

[1]This is equivalent to using predicate labeled transitions, where predicates test membership of a set of symbols.

Bigloo is an implementation of Scheme written by Manuel Serrano at INRIA [SW95]. Bigloo was originally designed to have fronts ends for both Scheme and ML. It has backends which compile to C, JVM, and .NET bytecode which makes it highly portable and allows easy access to the Java and .NET libraries.

Scheme was ultimately chosen as the implementation language because functional programming is a much more convenient expression of algorithms than imperative programming. Allowing composition of functions seemed natural and the author is sure she saved a lot of time by using it. Logic programming was considered but rejected because some concepts did not map well onto it. Scheme was the functional language chosen over others like Haskell or ML due to the authors knowledge of it and because it has numerous implementations on all modern platforms.

4.6.2 Graphviz

Graphviz[2] is used to produce the state diagrams of finite state machines seen in 4.4 on page 22. It is a software package for producing visualisations of directed graphs. It takes in graph descriptions in a simple text format and produces images in a variety of different formats. The *dot* program from the Graphviz package which was used in the implementation to produce state transition graphs. *Dot* produces hierarchical directed graphs with the minimum number of edge crossings, and shortest edge lengths.

4.6.3 Other Finite State Libraries

The possibility of using a third party library for finite state operations was considered, however there are no finite state libraries for Scheme with the same power and flexibility as the FSA toolkit for Prolog[vN96], or the Grail library[3] for C++. There are some options, for example the LAML project [4] includes a library for finite state operations; as well as several regular expression libraries, for example pregexp[5]. However, the LAML library does not support a large set of finite state operations, it is intended primarily for building lexers and does not implement all the required finite state operations such as intersection, complement, etc. While the regular expression libraries support a greater set of operations, they are not intended to work outside the domain of text processing, and hence do not work well over arbitrary alphabets.

[2]Graphviz homepage: http://www.graphviz.org/

[3]Grail homepage: http://www.csd.uwo.ca/Research/grail/grail.html

[4]Lisp Abstracted Markup Language, LAML, homepage: http://www.cs.auc.dk/~normark/scheme/

[5]Portable Regular Expressions for Scheme and Common Lisp, pregexp, homepage: http://www.ccs.neu.edu/home/dorai/pregexp/pregexp.html

Chapter 5

Conclusion

A method of expressing Event Logic formulae in terms of FST has been given, barring certain cases of the *tense* operator described in 3.3.5 on page 17. In addition, a definition of the inference method for the FST representation is given.

A program has been implemented which can build finite state machines from event formulae, which in turn can be used to recognise event occurrences. This program was then compared to Leonard, another implementation of Event Logic. This lead to the fact that the FST representation of Event Logic moved the bulk of the work to compile time of event recognition through the use of finite state machines. However, the FST representation faces the problem of an alphabet of exponential size. Some possible solutions to this, such as working with ϵNFAs or GNFAs are discussed.

This report has shown how Event Logic can become more powerful by expressing it in FST. FST offers the benefits of model independence, in addition to allowing all the existing knowledge about regular languages to be applied to Event Logic expressions. Representing Event Logic in FST is a step towards the ultimate aim of [Sis01], finding a concrete and testable representation of the semantics of verbs. Using regular languages to define verbs opens them up them up potentially deeper examination, by allowing all methods that may be used to reason about regular languages to be applied to them.

Bibliography

[All83] James F. Allen. Maintaining knowledge about temporal intervals. *CACM, November 1983. Also in "Readings in knowledge representation", ed. Ronald J. Brachman and Hector J. Levesque, Morgan Kaufman, 1985*, 26(11):832–843, 1983.

[All84] James F. Allen. Towards a general theory of action and time. *Artif. Intell.*, 23(2):123–154, 1984.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[BK03] Kenneth R. Beesley and Lauri Karttunen. *Finite State Morphology.* CSLI Publications, Stanford, California, 2003.

[Eil74] Samuel Eilenberg. *Automata, Languages, and Machines.* Academic Press, Inc., Orlando, FL, USA, 1974.

[Fer04] Tim Fernando. A finite-state approach to events in natural language semantics. *Journal of Logic and Computation*, 14(1):79–92, 2004.

[Fer06] Tim Fernando. Finite-state temporal projection. In *CIAA*, pages 230–241, 2006.

[FGS02] Alan Fern, Robert Givan, and Jeffrey Mark Siskind. Specific-to-general learning for temporal events. In *Eighteenth national conference on Artificial intelligence*, pages 152–158, Menlo Park, CA, USA, 2002. American Association for Artificial Intelligence.

[KCE98] Richard Kelsey, William Clinger, and Jonathan Rees (Editors). Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, 1998.

[KHMW99] Jörg Kahl, Lothar Hotz, Heiko Milde, and Stephanie Wessel. A more efficient knowledge representation for allen's algebra and point algebra. In *IEA/AIE*, pages 747–752, 1999.

[KK94] Ronald M. Kaplan and Martin Kay. Regular models of phonological rule systems. *Comput. Linguist.*, 20(3):331–378, 1994.

[Muk] Madhavan Mukund. Finite-state automata on infinite inputs.

[Sho87] Yoav Shoham. Temporal logics in ai: Semantical and ontological considerations. *Artif. Intell.*, 33(1):89–104, 1987.

[SIP96] M. SIPSER. *Introduction to the Theory of Computation.* PWS, Boston, MA, 1996.

[Sis01] Jeffrey Mark Siskind. Grounding the lexical semantics of verbs in visual perception using force dynamics and event logic. *J. Artif. Intell. Res. (JAIR)*, 15:31–90, 2001.

[SM96] Jeffrey Mark Siskind and Quaid Morris. A maximum-likelihood approach to visual event classification. In *ECCV '96: Proceedings of the 4th European Conference on Computer Vision-Volume II*, pages 347–360, London, UK, 1996. Springer-Verlag.

[SW95] Manuel Serrano and Pierre Weis. Bigloo: A portable and optimizing compiler for strict functional languages. In *Static Analysis Symposium*, pages 366–381, 1995.

[Tal88] Leonard Talmy. Force dynamics in language and cognition. *Cognitive Science*, 12:49–100, 1988.

[vN96] Gertjan van Noord. FSA utilities: A toolbox to manipulate finite-state automata. In *Workshop on Implementing Automata*, pages 87–108, 1996.

[Wei05] Nikolai Weibull. Theoretical foundation of regular expressions and text editors. Master's thesis, Gothenburg University, July 2005.