From Temporal Markup to Monadic Second-order Logic

Seán Healy

B.A. (Mod.) Computer Science and Language
School of Computer Science and Statistics
Supervisor: Dr. Tim Fernando
April 4, 2017

Contents

Chapter 1

Introduction

1.1 TimeML

TimeML is an ISO-standard markup language for the description of temporal events and relations in documents. Events and temporal expressions are marked with XML tags, and relations are later specified between these events using tags typically placed at the end of the document. TimeML was first introduced by Pustejovsky et al. (2002). Several corpora have made use of TimeML since then, among the largest freely available and accessible corpora is *TIMEBANK* 1.0, introduced in Pustejovsky et al. (2003), which was relied on heavily in this study of TimeML as a language (§1.1.2).

Another interesting corpus (for language diversity), was the French TimeBank corpus discussed and outlined in Bittar et al. (2011).

1.1.1 TimeML Specification

An extensive specification of the tags used in TimeML is given in Pustejovsky et al. (2002). Syntactically, TimeML is described by the document type definition files listed in the appendices – which were included with the TimeML corpora used in this paper.

EVENT **and** *TIMEX3* The tags central to all others in TimeML are the pair *EVENT* and *TIMEX3*, which together label periods of time in which an event occurs or a temporal expression holds true. All events and temporal expressions in TimeML are treated as *potential* intervals.

LINK **tags** TimeML makes use of three *LINK* tags, namely *TLINK*, *ALINK* and *SLINK*. The most significant of the three tags, and the most extensively used is the *TLINK tag*. All *LINK* tags serve the purpose of marking temporal relations between events, temporal expressions or aspectual markers.

TLINK The tag *TLINK* is used to mark relations between events or temporal expressions, where the attribute *relType* may hold a relation of type comparable to those introduced by Allen (1983).

The temporal relations of TimeML The relation set from which TimeML may annotate a document using *TLINK* is in fact only a subset of the 13 Allen relations. Table 1.1 shows the mapping between TimeML relations and Allen relations assumed in this paper. Note that *IDENTITY* and *SIMULTANEOUS* both map to $=$. It was difficult to find a discernible difference between these two relations in online resources, so it is assumed that the difference is not one in the temporal relation among two events, but in the anchoring of the events – are the two events one and the same or are they simply simultaneous?

TimeML *relType*	$R \in Allen$
BEFORE	$<$
AFTER	$>$
IBEFORE	m
IAFTER	mi
INCLUDES	di
IS_INCLUDED	d
BEGINS	s
ENDS	f
BEGUN_BY	si
ENDED_BY	fi
SIMULTANEOUS	$=$
IDENTITY	$=$

Table 1.1: Mapping TimeML temporal relations to Allen relations

The Allen relations o and oi (*overlaps* and *overlapped-by*) are not present in the original TimeML specification for *TLINK* relations. This is not to say that TimeML cannot model an overlaps relation between two events, it simply cannot do so using a single *TLINK* tag.

The overlaps relation $x\ o\ x'$ is described using the string $\boxed{x}\boxed{x, x'}\boxed{x'}$ in Fernando (2012). In practice a lot more event pairs may fulfill the o or oi relation when compared to m or mi, though we may not be likely to specify this relation explicitly in language – and TimeML is, after all, a markup language whose usage has been thus far limited to natural language corpora. Nonetheless, overlapping events can be modelled; by the *ALINK* tag for example. This involves explicitly acknowledging the overlapping segment of two events. E.g. $x\ TERMINATES\ y$, $x'\ INITIATES\ y$; "The Ussher reading room is closed by the *changeover*. The 24-hour reading room is opened by the *changeover*.". TimeML may use an aspectual link *ALINK* to connect *the changeover* to bother events x and $x^{p}rime$, one *started by* y, and the other *ended by* y, the relation between x and x' is thus o. The overlaps relation was in fact added to later versions of TimeML as *OVERLAP_BEFORE* and *OVERLAP_AFTER*, but not surprisingly its occurrence (in the corpora encountered in this paper) is extremely rare.

1.1.2 The *TIMEBANK* Corpus

The *TIMEBANK* corpus is a corpus of news articles from the late 90s annotated, by a semi-automated process, in TimeML. The corpus available for download freely on timeml.org consists of 78 of these articles annotated to varying degrees of success with rich temporal

event and relation markup.

Although Pustejovsky et al. (2002) gives its own specification of TimeML, as with natural language, descriptive insight is much more powerful than prescriptive specification. To begin with, a statistical analysis of the *TIMEBANK* corpus was carried out . Although Pustejovsky et al. (2003) includes its own statistical analysis of TimeML usage across the corpus, since this author was working off only a selection of 78 (publicly available) items from the *TIMEBANK* 1.0 corpus, an additional statistical study of the *sub-corpus* was carried out.

It was found that, although *ALINK* is given equally energetic specification efforts in many of the papers on TimeML, it is in practice in the genre of news articles at least, very rarely used. *SLINK*s are used to a larger degree, but *TLINK* emerges as the most prolific tag by a large margin. The class of *TLINK* is rarely a *SIMULTANEOUS* relation in *TIMEBANK* 1.0. The *IDENTITY* relation is a much more common occurrence. More detailed statistical data can be found in 1.1.4.

1.1.3 The French TimeBank Corpus

The French TimeBank, which is also easily accessible on the internet, was another source of insight into the way TimeML is commonly used. One thing that became apparent in *TIMEBANK* 1.0 was the redundancy of the *MAKEINSTANCE* tag. The purpose of this tag is described in Pustejovsky et al. (2002) as being a necessity for creating *instances* of events. However, in *TIMEBANK* 1.0, there is never more than one *MAKEINSTANCE* tag for each event. The one-to-one (bijective) relationship between the occurence of an *EVENT* tag, and a *MAKEINSTANCE* was noticed before Bittar et al. (2011), as the corpus of annotated French articles doesn't feature *MAKEINSTANCE*, nor is the tag included in the DLD file for their definition of TimeML, which is built on ISO-TimeML.

Tag	Count
$TLINK$	9249
$EVENT$	8919
$TIMEX3$	1565
$SLINK$	1899
$ALINK$	192

Table 1.2: Comparison of the occurrence of the main tags in TimeML

1.1.4 Statistical Analysis of TimeML Usage

The following statistics are gathered from three TimeML corpora freely available on the internet: *TIMEBANK* 1.0, French TimeBank, and several articles from *TIMEBANK* 1.1.

The purpose of this statistical analysis was to figure out which parts of TimeML are most used in practice, so that a well-used fragment of the language can be selected for study. TimeML is a vast language, easily extensible, and it would be difficult to work cohesively on connect-

Tag	Count
BEFORE	2792
IS_INCLUDED	1847
INCLUDES	1409
AFTER	1071
IDENTITY	791
DURING	513
SIMULTANEOUS	350
BEGINS	116
ENDED_BY	96
BEGUN_BY	91
IBEFORE	67
ENDS	56
IAFTER	40
OVERLAP_BEFORE	6
OVERLAP_AFTER	4

Table 1.3: Comparison of the prominence of *relType* attribute values across the corpora for *TLINK*

Tag	Count
MODAL	988
EVIDENTIAL	664
FACTIVE	162
COUNTER_FACTIVE	32
CONDITIONAL	25
NEG_EVIDENTIAL	21
NEGATIVE	7

Table 1.4: Comparison of the prominence of *relType* attribute values across the corpora for *SLINK*

Tag	Count
INITIATES	82
CONTINUES	44
TERMINATES	41
CULMINATES	17
REINITIATES	8

Table 1.5: Comparison of the prominence of *relType* attribute values across the corpora for *ALINK*

Tag	Count
DATE	1136
DURATION	209
TIME	185

Table 1.6: Comparison of the prominence of *TIMEX3* types across the corpora

Tag	Count
OCCURRENCE	5685
REPORTING	994
STATE	838
I_ACTION	613
I_STATE	446
ASPECTUAL	162
MODAL	72
PERCEPTION	67
CAUSE	22
EVENT_CONTAINER	20

Table 1.7: Comparison of the prominence of *EVENT* types across the corpora

ing the entire language to MSO. Instead, the most prominent features in the language were chosen for further study, namely the combination of events and temporal links ($EVENT$ and $TLINK$).

1.2 Interval Logic

Definition 1.2.1. *Interval* A period of time during which some fluent holds from beginning to end. Unlike event classes, which may categorise multiple intervals, an interval may not define several disconnected periods in a timeline.

For the purposes of this paper, it is assumed that all events and temporal expressions in TimeML are intervals. This is not strictly-speaking always the case, due to the possibility of *event classes* or intermittent events, as discussed in James et al. (2010).

However it was found, during this author's statistical analysis of the *TIMEBANK* corpus, that most of the events tagged are basic intervals in structure, and in most cases a successful conversion from TML to Allen temporal relation network diagrams is possible.

1.3 Temporal Strings

In this paper, *temporal strings*, *temporal fluent strings*, *timelines*, and so on, may all be considered synonymous terminology for a textual representation of intervals. A temporal string is a string of sets, each containing *fluents* describing the state of the world at that position in the string. Fluents may be ABoxes (e.g. negated expressions) or TBoxes (e.g. intervals). A timeline is consistent with another timeline if both can be spliced together to form a new, more complete timeline. A formal definition of splicing as an operation between temporal strings, and temporal string languages, is given in great depth in section 2.2.

Temporal strings and MSO Monadic second-order logic (MSO) was linked to regular languages by Büchi (1962), meaning formal verification methods using finite-state techniques can be built from MSO, a subset of second-order logic limited to the quantification of second-order predicates of arity one. It has been noted (e.g. Fernando (2016)), that the temporal string representation (e.g. $\boxed{E}\,\boxed{\,}\,\boxed{decidingR}$) is essentially equivalent to MSO. There are clear states between positions in the string. The goal of this paper is to study fragments of TimeML and develop an approach to attaining these temporal strings from TML documents. The methodology is described in detail in Chapter 2, but the summary of what the author's approach entailed is as follows:

- Apply Allen constraint propagation to the knowledge represented in a TML document to infer unwritten relations between events.
- From the relations represented in a TML document, and from those inferred from Allen constraint propagation, build temporal strings according to the mapping described in

Fernando (2013).

- Develop an operation for merging these strings together regardless of length (traditionally superposition is only possible between strings of equal length). This operation can be used iteratively to build a temporal string representation of a TML document.
- Develop a heuristic for deciding the order in which merging these strings should be carried out, based on the logical structure of events.

1.3.1 The Satisfiability Problem

One of the main unavoidable drawbacks of Allen constraint propagation is the fact that the algorithm does not detect invalid input, except if the inconsistency is evident between three nodes in the temporal network Allen (1983). Given a knowledge base of temporal relations between intervals, determining if the network is contradictory is equivalent to the satisfiability problem, and thus NP-hard. The Prolog code snippet in Figure 1.1 evaluates to true if a network of temporal relations has at least one valid interpretation (though there is no guarantee this code will run fast). The predicate *rmember*, behaves similarly to the Prolog built-in *member* of list predicate, though it unifies a 3rd predicate with a version of the list excluding the member.

```
consistent_net(Edges) :-
  consistent_state(Edges),
  ((rmember(Edge,Edges,X),
      disjunct(Edge)) ->
  guess(Edge,D),
  consistent_net([D|X]); true).

consistent_state(Edges) :-
  forall(
    (
      member([A,R1,B],Edges),
      member([B,R2,C],Edges),
      member([A,R,C],Edges)
    ), (
      constraints(R1,R2,RC),
      R subset RC
    )
  ).
guess(L,[M]) :-
  member(M,L).
disjunct([_,[_],_]) :- !, fail.
disjunct(_).
```

Figure 1.1: Testing a temporal network for satisfiability in Prolog

Chapter 2

Methodology

2.1 Constraint Propagation

Before beginning the first step mentioned in the introduction, some improvements are proposed for the constraint propagation algorithm first specified by citeauthorallen. In Allen (1983), the time complexity of the program for adding n nodes to an empty interval network is inaccurately stated as $O(n^2)$. Though a time-complexity of $O(n^2)$ is attainable, there is a major flaw with Allen's *Add* function (which is called n times), and that is the usage of a simple queue, rather than the usage of a queue, hash table hybrid structure. Allen's algorithm uses a queue to schedule $ToDo$ tasks during the main loop of the algorithm. The conclusion of the algorithm depends on the emptiness of the $ToDo$ list. Although it is assumed during Allen's analysis that the queue will never contain duplicate items, this is not the case in reality, and this causes a lot of redundant computation.

Through the use of both a hash table and a queue, one can ensure that items are only scheduled in the queue when they are not already present in there. The $O(1)$ lookup time for the hash table is responsible for no impact on the time-complexity of the algorithm. Algorithm 1 is the version given in Allen (1983) for constraint propagation.

Analysis of Allen's algorithm Allen noted, since the progressive enlargement of the $ToDo$ queue was determined by a proper subset operation between the *current* state of a relation set, and the state of the relation set once it is affected by a constraint from the transitivity table, that the $ToDo$ queue will only ever grow at most $13n^2$ times. There are n^2 edges between n nodes in a directed graph, where each node is connected to itself and all other nodes. Allen notes that although *Add* may add more than n nodes to the queue, on average, each call to *Add* *enqueue*s n times, thus the overall time complexity of n calls to *Add* is $\Theta(n^2)$. We can go further to say that n calls to *Add* in an algorithm for iterative constraint propagation is *worst case* On^2. If we imagine an accumulator $\mathcal{Q}$ for all the queues in n calls to *Add*,

which tracks the number of enqueue calls to all queues; It should never grow to a size greater than $13n^2$, as $13n^2$ valid modifications (adding new information) of an interval network would result in a fully defined network with no ambiguity; at such a point, the parameter necessary for *enqueue* would be unattainable: a modification of the network must add new information.

Algorithm 2 shows the modified algorithm, relying on a hash set in conjunction with the $ToDo$ queue, and shifting the computation of new relations to a new location within the two for loops.

Allen carried out this analysis while assuming that the $ToDo$ queue would never contain duplicates. We found, in practice, that each call to *Add* would use a queue structure growing to include duplicate constraints scheduled for propagation. A call to *Add* that would have n *enqueue*s, disregarding duplicates in the queue, would have as many as n^2 when duplicate entries in the queue are considered; this decelerates the running time of the constraint propagation program from the intended $O(n^2)$ to a much slower figure. Furthermore, it requires a queue that may grow in length to be larger than n^2. Through some tweaking of Allen's approach, and by the introduction of a hash table, these bugs were fully rectified, and reaching $O(n^2)$ time was possible.

There is another problem, in the area of memory this time. Allen's justification for the "reference interval" data structure described in detail in his paper is the reduction in space complex-

Algorithm 1 Allen constraint propagation

```
1: procedure C(R1, R2)                ▷ Collect constraints from the transitivity table.
2:     S ← ∅
3:     for r1 ∈ R1 do
4:         for r2 ∈ R2 do
5:             S ← S ∪ T(r1, r2)
6:         end for
7:     end for
8:     return S
9: end procedure
10: procedure ADD(iRj)
11:     empty queue q
12:     enqueue ⟨i, j⟩
13:     while ¬empty(q) do
14:         dequeue ⟨i, j⟩
15:         iNj ← iRj
16:         for k satisfies comp(k, j) do
17:             kRj ← kNj ∪ C(kNi, iRj)
18:             if kRj ⊂ kNj then
19:                 enqueue ⟨k, j⟩
20:             end if
21:         end for
22:         for k satisfies comp(i, k) do
23:             iRk ← iNk ∪ C(iNj, jRk)
24:             if iRj ⊂ iNk then
25:                 enqueue ⟨i, k⟩
26:             end if
27:         end for
28:     end while
29: end procedure
```

Algorithm 2 Allen constraint propagation

```
1: procedure ADD(iRj)
2:     empty queue q
3:     empty hashset h
4:     enqueue ⟨i, j⟩
5:     iNj ← iRj
6:     while ¬empty(q) do
7:         dequeue ⟨i, j⟩
8:         h ← h − {⟨i, j⟩}
9:         for k satisfies comp(k, j) do
10:            kRj ← kNj ∪ C(kNj, kRj)
11:            if kRj ⊂ kNj then
12:                if ⟨k, j⟩ ∉ h then
13:                    enqueue ⟨k, j⟩
14:                    h ← h ∪ {⟨k, j⟩}
15:                end if
16:                kRj ← kNj ∪ C(kNi, iRj)
17:            end if
18:        end for
19:        for k satisfies comp(i, k) do
20:            if iRj ⊂ iNk then
21:                if ⟨i, k⟩ ∉ h then
22:                    enqueue ⟨i, k⟩
23:                    h ← h ∪ {⟨i, k⟩}
24:                end if
25:                iRk ← iNk ∪ C(iNj, jRk)
26:            end if
27:        end for
28:    end while
29: end procedure
```

ity from $O(n^2)$ (problematic with 80s RAM) to $O(n)$. However, the $ToDo$ queue is entirely disregarded in this analysis. I found, through erroneously allocating the queue some memory on a linear scale, that this would often lead to segmentation faults in the C++ implementation of Allen's algorithm. The problem lies in the fact that there are n^2 possible relations between intervals in a temporal network, so the $ToDo$ list, although not often reaching that far, will occasionally grow to n^2. Thankfully, n^2 memory for a corpora of short articles is trivial by today's standard. In fact, the C++ implementation of Allen's algorithm (listed in the appendices), uses a low-level 2-dimensional array (or matrix) to model a temporal network rather than a reference intervals data structure. The reason for this is that memory is now a abundant resource, and the improved constant factor of doing things on such a low level allows for the almost instantaneous processing of the entire corpus.

2.1.1 Low-level Optimisations for Allen's Algorithm

Some low-level optimisations are proposed and implemented to improve the running time of the Allen constraint propagation algorithm.

Sets Allen's algorithm makes use of sets and several operations between sets. However, the types of sets seen are special because they never grow

larger than 13 in size, and they always contain elements from the same pool of 13: The 13 Allen relations. Rather than storing the sets literally in memory, they can be represented as 13bit binary integers, where a 1 at a given bit means the relation corresponding to that bit is *in* the set. This also allows for bitwise operations (the lowest level operation a CPU can perform) and boolean logic to be applied for common set operations.

Set operation	Binary operations
Set union	$a\ OR\ b$
Set intersection	$a\ AND\ b$
Set equality	$a\ EQ\ b$
Subset	$(a\ AND\ b)\ EQ\ a$
Proper subset	$(a\ AND\ b)\ EQ\ a \wedge a\ NEQ\ b$

The queue The queue may grow to consume worst-case $O(n^2)$ space. Assuming that is within the allowances of your hardware (it was in the author's case), allocating this space in advance can allow for the usage of a structure called a *circular queue*.

Unlike a linked list, which dynamically increases in size, using an unpredictable amount of memory, often relying on a garbage collector to so so, a *circular list* has a fixed capacity. A queue can only be implemented by the doubley-linked variant of the linked list, as it performs operations on both sides, top and bottom. Two reference pointers, in addition to the integer value stored by a node in the chain, means that a linked list may use up to 3 times as much memory as the circular queue if the worst-case scenario is realised. There is a clear time-advantage to using a circular queue. Enqueue and dequeue operations on a linked list must modify not only the reference pointer of the first or last item in the queue, but also the penultimate nodes pointer in the opposite direction. A linked list structure also relies on garbage collection. In short, it is noted that the memory usage by Allen's algorithm is predictable (polynomial), and so the implementation in this paper (written in C++) uses a circular queue structure for the $ToDo$ list. (Cormen et al., 2009, p 232-235) gives a clear analysis of the circular queue structure.

2.2 Superposition to Splicing

We propose a new declarative model for fluent string *superposition* that bounds the space needed for a superposition operation to the size of the largest possible string length in the resulting set. In recent papers, the definition of string superposition used to describe fluents a and b occurring in the same time frame, is as follows:

$$a@b := trim\Big(bc\Big(\square^* \boxed{\text{a}}^+ \square^* \,\&\, \square^* \boxed{\text{b}}^+ \square^*\Big)\Big) \tag{2.1}$$

$$trim(s) := \begin{cases} trim(s') & \text{if } (\exists s')s \in \Big\{ \square\, s', s' \,\square \Big\} \\ s & \text{otherwise} \end{cases} \tag{2.2}$$

$$bc(\alpha\beta s) := \begin{cases} s & \text{if } \alpha = \beta = s = \varepsilon \\ bc(\beta s) & \text{otherwise if } \alpha = \beta \\ \alpha\ bc(\beta s) & \text{otherwise} \end{cases} \tag{2.3}$$

As in (2.1), in this paper the binary op-

eration @ is used to mean two fluents occur in the same time frame. This operation is both commutative and associative ($a@b = b@a$, $(a@b)@c = a@(b@c)$). The superposition of two distinct intervals, $a@b$, models a set of 13 strings, where each element corresponds to one of the Allen relations (a finite set). Logically, then, a star-free language should suffice to describe superposition as a binary relation, and a star-free language should exist for general superposition of n intervals. With $\mathcal{S}$ as the set of star-free languages (the set of all finite sets), $\mathcal{I}$ as the infinite set of intervals, and $\mathcal{F}$, the set of binary operations such that $\forall(f \in \mathcal{F})(f : \mathcal{I} \times \mathcal{I} \mapsto \mathcal{S})$, one has $(\exists\varphi \subseteq_{\mathcal{L}} S)\varphi \models f$. $A \subseteq_{\mathcal{L}} B$ means that all formal language expressions used in model A are elements of the set B.

To begin with, while superposing two intervals a and b, the resulting set of strings only contains elements s:

$$trim(\pi_a s) = \boxed{\text{a}}^{\,n} \qquad (2.4)$$
$$\text{where } n \leq 2$$

Furthermore, the superposition of m intervals will only ever produce strings of length $\leq (2m-1)$. The proof for this can be carried out by inductively superposing all intervals over an empty string ε (updating the frame to the maximum length superposition each time), until the time-frame contains all intervals, and is at its largest length. Regardless of the string's composition, to superpose an interval over it will only ever increase the length of the resultant string by a maximum of 2, except when superposing over ε. To superpose i over ε would simply produce $\boxed{\text{i}}$.

Note that ε, denoting an empty temporal string, is not the same as $\boxed{\ }$, which denotes a temporal string of length 1, where the set of fluents holding at 1 is $\emptyset$.

Given a string onto which an interval is superposed, there are 10 unique ways (excluding rotations) of choosing point pairs for the superposition operation (Figure 2.1). Choosing a point in the string can either involve choosing a boundary between two consecutive sets, or choosing a single set, creating a new boundary by duplicating the set at that position, then choosing that boundary. The latter method of choosing a point produces a resultant string with an incremented length. Since one chooses a *pair* of points for segmentation, the resultant superposed string may extend the input string by (up to) 2 sets.

(2.5) to (2.8) denote point choices beyond the edge of the string. In these edge-cases, an additional empty set must be appended or prepended to the side of the string before the component-wise union of the superposed interval is applied. Thus choosing a point here increases the overall string length by 1. $\boxed{\ }\cdot\boxed{\ }$ depicts choosing a point for superposition at a boundary between sets, causing no increase in the length of the resultant superposed string over the original string argument. However, $\boxed{\ |\cdot|\ }$ duplicates a set in the string (before component-wise union with the interval), causing an increase of 1 in the resultant superposed

string. (2.7), (2.9) and (2.14) capture cases where an increase of 2 is caused by applying the superposition operation. Two additional sets are added to the string before component-wise union begins. Both sets are added to produce new respective boundaries between which the component-wise union is applied.

$$\mathop{\textcircled{@}}_{j=1}^{m} i_j = (\epsilon @ i_1) @ i_2 @ \ldots @ i_m \qquad (2.15)$$

We know that $(\epsilon @ i_1)$ can only produce a singleton set containing a string of length one, and after that there are $(m-1)$ operations applied sequentially to the resultant string set. Each of these may increase elements of the string set by a length of two. Therefore the maximum length of a string in the final set is $2(m-1)+1 = (2m-1)$.

The basic operation of superposing (@) time intervals $i_1 \ldots i_m$ (where m is a constant) in the same time-frame is to place the intervals in their own respective *string wrappers*, and perform conventional superposition over these string sets. Since no superposition of m intervals will contain a string of length $> (2m-1)$, one could easily refine our earlier definition of @ (2.1) to use only star-free languages, i.e.

$$\mathop{\textcircled{@}}_{j=0}^{m} i_j \;:= trim\Big(bc\Big(\qquad (2.16)$$

$$\mathop{\&}_{j=0}^{m} \square^{\{0,2m-2\}} \boxed{i_j}^{\{1,2m-1\}} \square^{\{0,2m-2\}}$$

$$\Big)\Big)$$

We can apply another reduction to

$\$ \bullet \square \ldots \square \bullet \square \ldots$ (2.5)

$\$ \bullet \square \ldots \square\bullet\square \ldots$ (2.6)

$\bullet \$ \square \ldots \square \bullet \square \ldots$ (2.7)

$\bullet \$ \square \ldots \square\bullet\square \ldots$ (2.8)

$\bullet \$ \square \ldots \square \$ \bullet$ (2.9)

$\$ \bullet \square \ldots \square \bullet \$$ (2.10)

$\bullet \$ \square \ldots \square \bullet \$$ (2.11)

$\ldots \square \bullet \square \ldots \square\bullet\square \ldots$ (2.12)

$\ldots \square \bullet \square \ldots \square \bullet \square \ldots$ (2.13)

$\ldots \square\bullet\square \ldots \square\bullet\square \ldots$ (2.14)

Figure 2.1: The 10 unique ways to select a portion of a string for superposition (allowing set splitting). The $\bullet$ points may be placed at a string position, between string positions, at the string edge, or beyond the string edge.

the star-free regular expression

$$\boxed{}^{\{0,2m-2\}}\ \boxed{i_j}^{\{1,2m-1\}}\ \boxed{}^{\{0,2m-2\}} \qquad (2.17)$$

by intersecting it with the set of non-empty strings of length $\leq (2m-1)$:

$$\{s \mid 1 \leq |s| \leq 2m-1\} \qquad (2.18)$$

In this paper, the shorthand $\#(1, 2m-1)$ is used to denote the type of set outlined in (2.18). The result of this language intersection is the finite language

$$\boxed{}^{a}\ \boxed{i_j}^{b}\ \boxed{}^{c} \qquad (2.19)$$

where $b \geq 1$ and $a + b + c = m$ (noting that m is a known constant).

Beyond star-free Omitting any usage of the costly Kleene star from our definition of superposition is a fruitful first step, but there is one major problem with our definition of superposition thus far, and that is the reliance on block compression (*bc*) and *trim* functions. We will see that this involves redundant computation, and unnecessary ambiguity when theorem provers like Prolog are brought into the picture.

An ideal finite-state model for superposition would be one which is not only star-free, but also unambiguous. If we imagine a nondeterministic chain of finite-state transducers, $\mathcal{T} = t_1 \ldots t_n$, where the input of t_p is the output of t_{p-1}, for $2 \leq p \leq n$, and the overall input and output of the system is the input to t_1 from $\mathcal{T}$ and the output of t_n to $\mathcal{T}$ respectively – if such a system takes as input a set of m intervals $\xi = i_1 \ldots i_m$, and produces as output a valid superposition $\zeta \in @_{j=1}^{m} i_j$, then there must only be one valid path bringing the input ξ to produce ζ by the system $\mathcal{T}$. This is not yet true for our model.

Ambiguity in finite-state systems We define ambiguity not by non-determinism but by the paths taken to arrive at a given conclusion ζ from a given input ξ. A finite-state machine contains (among other things) a state-transition function δ, a set of states S, a set of accept states $F \subseteq S$, a starting state $s_0 \in S$.

δ can be thought of as a set of triples. An input ξ reaches a valid conclusion if there exists a string of tuples uvw, where $u = \langle s_0, \alpha_1, X \rangle$, $X \in S$, and $v \in \delta^*$, and $w = \langle Y, \alpha_1, f \rangle$, $Y \in S$, $f \in F$, such that $\pi_\alpha uvw = \xi$. The conclusion ζ of a finite-state transducer (or a chain thereof) is a tuple of the end-state and the output produced before the end-state was reached, $\langle f, o \rangle$. A finite-state system, represented as a predicate $g(\xi, uvw, \zeta)$, is unambiguous iff

$$(\forall path)(\forall alt) \qquad (2.20)$$

$$(g(\xi, path, \zeta) \wedge g(\xi, alt, \zeta) \rightarrow path = alt) \qquad (2.21)$$

Here is one case where the *bc* and *trim* approach displays ambiguity:

$$a@b = trim(bc(\boxed{\text{y}}\boxed{\text{y}}\boxed{\phantom{\text{y}}} \ \& \ \boxed{\phantom{\text{z}}}\boxed{\phantom{\text{z}}}\boxed{\text{z}})) \qquad (2.22)$$

$$a@b = trim(bc(\boxed{\text{y}}\boxed{\phantom{\text{y}}} \ \& \ \boxed{\phantom{\text{z}}}\boxed{\text{z}})) \qquad (2.23)$$

(2.22) and (2.23) both produce the same output, $\boxed{\text{y}}\boxed{\text{z}}$, though they evidently

don't use the same process (state transitions) to reach the output; (2.22) performs traditional superposition on strings of length 3 taken from $\boxed{\ }^{a}\boxed{i_j}^{b}\boxed{\ }^{c}$, and (2.23) performs the operation on strings of length 2. If one were to define this superposition as a Prolog predicate, *spose*, with the purpose of feeding input to a theorem solver, multiple, identical solutions would be found in the search graph. Ideally, we'd like

findall(S,spose([a,b],S),L).

to produce a list L of length 13, for the number of ways there are to superpose two intervals. Using block compression and trimming in *spose* causes a larger number of results however, due to its ambiguous non-deterministic approach. As with the use of Kleene-star expressions, which was addressed earlier, this duplication of results, via multiple state transition strings *uvw*, introduces redundant computation, and fails to describe a precise *declarative* model for superposition. Trimming and block compression, as an approach to modelling the *no time without change* philosophy, only succeeds by casting a wide net, then sifting through the results with *bc* and *trim* (imperatively).

Although the *trim* and *bc* approach is descriptively succinct, a sharper image of superposition can be made by integrating the *time is change* philosophy into the semantics of the superposition operation.

Modified superposition The binary operation $\langle\&\rangle$ has so far been working off the definition used in Fernando (2002) and Fernando (2004), namely

$$(\alpha_1 \dots \alpha_n)\&(\beta_1 \dots \beta_n) = \\ (\alpha_1 \cup \beta_1) \dots (\alpha_n \cup \beta_n) \tag{2.24}$$

This type of *component-wise* union across two strings of equal length has the benefit of not regarding order of computation. The union of the two sets from each string position pair is related in no way to the other union operations. All unions could be easily carried out concurrently, for example. The draw-back is that it relies on the application of *bc*, and without this, it does in fact produce strings with portions exhibiting no change in their sequential fluents. This may be intended behaviour in other applications of set strings, but as a model for temporal event description, it falls down.

The relation between the union operations is in fact very relevant in the case of interval superposition, as one is concerned with avoiding repeated set sequences, that is:

$$(\alpha_p \cup \beta_p) \neq (\alpha_{p+1} \cup \beta_{p+1}) \tag{2.25}$$

for $(1 \leq p \leq n-1)$. Integrating the constraint (2.25) into our description of superposition would cause the preconditions of the operation to grow in size; not only must the arguments of superposition be of the same length, but the resulting superposition must not include adjacent string positions without change.

Superposition of set strings can be carried out by finite-state machines, which can be represented as MSO sentences. The two strings, $\alpha_1 \dots \alpha_n$ and

$\beta_1 \ldots \alpha_n$ are sent as input to a finite-state system. MSO $_{\langle \Sigma, \Gamma, \Delta \rangle}$ is a tuple

$$\langle [n], [m], SA_n, SB_m, \{[\![A_x]\!]\}_{x \in \Sigma}, \{[\![B_x]\!]\}_{x \in \Gamma}, \{[\![U_x]\!]\}_{x \in \Delta} \rangle$$

where A and B are monadic predicates which map their subscript to a position in one of the input strings. U maps its subscript to a position in the output string. Thus, unlike the traditional usage of MSO for the purpose of model matching with a single input, this 7-tuple is used to model binary operations over strings of sets.

The subscript $\langle \Sigma, \Gamma, \Delta \rangle$ represents the alphabets (Σ and Γ) of the input arguments, and the alphabet (Δ) of the output which is usually some set operation between Σ and Γ. In the case of superposition, $\Delta = \Sigma \cup \Gamma$.

$$SA_n := \{\langle p, p+1 \rangle \mid 1 \leq p \leq n-1\}$$
$$SB_n := \{\langle p, p+1 \rangle \mid 1 \leq p \leq m-1\}$$

where $n, m \geq 0$. Now superposition can be defined using the following MSO $_{\langle \Sigma, \Gamma, \Delta \rangle}$-sentence:

$$\begin{aligned}
&(\forall x)(\\
&\quad (((\exists y)(SA(x,y))) \longleftrightarrow \\
&\qquad ((\exists y)(SB(x,y)))) \\
&\quad ((\forall A_{\overline{\alpha}})(A_{\overline{\alpha}}(x) \rightarrow U_{\overline{\alpha}}(x)))\wedge \\
&\quad ((\forall B_{\overline{\beta}})(B_{\overline{\beta}}(x) \rightarrow U_{\overline{\beta}}(x)))\wedge \\
&\quad (((\exists y)(S(x,y))) \rightarrow \\
&\qquad ((\exists U_\mu)(U_\mu(x) \wedge \neg U_\mu(y)))) \\
&)
\end{aligned} \tag{2.26}$$

This modified form of superposition can be described by a function g:

$$g(\alpha, \beta) := f(\emptyset, \alpha, \beta) \tag{2.27}$$
$$f(p, \alpha, \beta) :=$$
$$\begin{cases}
(a \cup c).f(a \cup c, b, d) & \text{if } \alpha = \boxed{\text{a}}\, b \wedge \beta = \boxed{\text{c}}\, d \\
& \quad \wedge a \cup b \neq p \\
\epsilon & \text{otherwise if } \alpha = \beta = \epsilon \\
& \quad \wedge p \neq \emptyset \\
null & \text{otherwise}
\end{cases}$$

The recursive formula (2.27) can be captured by the Prolog predicate *superpose* as follows in Listing 1.

```
superpose(A,B,U) :- sp([],A,B,U).
sp(Pre,[H1|T1],[H2|T2],[U|T]) :-
   union(H1,H2,U),
   U \= Pre,
   sp(H,T1,T2,T).
sp(Pre,[],[],[]) :- Pre \= [].
```

Listing 1: Modified superposition described in Prolog

2.2.1 Classes of Superposition

When we speak of superposition, we may mean superposition of strings $\langle \& \rangle$, superposition of languages $\langle \& \rangle$, superposition of intervals $\langle @ \rangle$, or as will soon be encountered, superposition of string and interval $\langle \hat{@} \rangle$. From at least Fernando (2004) onwards, the definition (2.28) for language superposition is used.

$$L \& L' = \bigcup_{s \in L} \bigcup_{s' \in L'} s \& s' \tag{2.28}$$

where $s\&s'$ is traditional superposition of strings.

The model for superposing intervals (or any class of fluents), is described by *padded superposition* in Fernando (2002), and the author has since reduced this definition (in the case of superposing two intervals) to one avoiding the use of the Kleene star, by superposing languages of the form (2.19).

The aim was to create a minimal, purely declarative description of superposition, and that has yet to be achieved for the superposition of arbitrary n intervals, $n > 2$. The model for superposition described by (2.26) cannot simply be inserted into (2.16) to replace the traditional model of superposition, because (2.16) relies on repetitions in strings to model superposition – but the modified *superposition of strings* operation doesn't allow repetitions.

One could, in the loop, apply traditional superposition until the last iteration, and then apply modified superposition to ensure no repetitions occur, but that would involve maintaining a large sum of "potential solutions" during the execution of the loop, which are discarded by the final application of *modified superposition*. Avoiding this "wide net" approach was the main goal however.

The solution the author came up with for superposition of intervals $\langle @ \rangle$, was the introduction of an additional operation $\langle \hat{@} \rangle$, which is applied between a string and an interval.

Terminology In TimeML, all events are intervals. We may consider the concept *interval*, as in "the interval i", to be interchangeable with the notion of a set string of length 1, where the set at position 1 is a singleton $\{i\}$, i.e.

$$TML\ interval\ i := \boxed{\text{i}}$$

Recalling the 10 unique ways that exist to choose parts of a string for superposition of an interval (Figure 2.1), it should be possible to model superposition of an interval $\boxed{\text{i}}$ over a string of any length quite easily. Unlike traditional superposition of strings, and the modified superposition outlined previously, equal length would not be a precondition among the input here (most strings are greater than the length of an interval, which is 1), nor would the output be the same length as the input. In fact, $s\hat{@}i$ would model strings s', $max(1, |s|) \leq |s'| \leq (|s| + 2)$.

Before beginning, some structures that will prove useful later should be outlined. The meta-character $\bullet$ may be placed at boundaries between sets in a set string to denote the pair of points that will mark the string segment within which the interval is superposed. For example, $\bullet \boxed{\varphi_1} \ldots \boxed{\varphi_n} \bullet$ would mean superposing some interval over each set in the string.

A binary operation $\langle \% \rangle$ between a string and language is defined, such that the string (left) is *segmented* into strings

in the language (right), using the point character to mark the point of segmentation.

$$padded(s) \longleftrightarrow s = \boxed{\ } s' \vee s = s' \boxed{\ } \tag{2.29}$$

$$tailed(s) \longleftrightarrow (s = s'\bullet) \vee \tag{2.30}$$
$$s = s' \boxed{\ }\boxed{\ }$$

$$marked(s) \longleftrightarrow s = s' \bullet s'' \tag{2.31}$$

$$\begin{aligned}
&s\%L \longleftrightarrow \\
&\quad (\\
&\qquad (\neg padded(s) \rightarrow X = \{s \boxed{\ } \bullet\}) \oplus \\
&\qquad (X = \emptyset) \\
&\quad) \wedge \\
&\quad (\\
&\qquad (\neg(marked(s) \vee padded(s)) \rightarrow \\
&\qquad\quad Y = \{\bullet \boxed{\ } s\} \\
&\qquad) \oplus \\
&\qquad (Y = \emptyset) \\
&\quad) \wedge \\
&\quad L = X \cup Y \cup \{s'\alpha \bullet \alpha s'' \mid \\
&\qquad s = s'\alpha s'' \wedge \\
&\qquad |\alpha| \leq 1 \wedge \\
&\qquad \neg tailed(s'\alpha) \wedge \neg marked(\alpha s'') \\
&\quad \}
\end{aligned} \tag{2.32}$$

Next we extend the behaviour of $\langle\%\rangle$, so that it may have either a string or a language on the left hand side. For a language as the left argument, the definition of $\langle\%\rangle$ from (2.32) is used inside a set generator (2.33).

$$L\%L' \longleftrightarrow L' = \bigcup_{s \in L} \{s' \mid s\%L'' \wedge s' \in L''\} \tag{2.33}$$

We define the binary operation $\langle\%^n\rangle$ for $n \geq 1$ applications of $\langle\%\rangle$, i.e. (2.34).

$$\begin{aligned}
&X\%^n L \longleftrightarrow \\
&\quad (n > 1 \wedge X\%L' \wedge L'\%^{n-1}L) \oplus (X\%L)
\end{aligned} \tag{2.34}$$

where X is either a string or a language. In the case of $\langle\%^2\rangle$, the set of possible substrings for superposition is being modelled, each string in the language containing exactly 2 non-adjacent point markers $\bullet$.

To superpose an interval or fluent over a portion of a string, an operation between 2-mark strings, $s = u \bullet v \bullet w$, and intervals i, is defined, which applies component-wise union on all $\alpha \in v$. The operation $\langle\hat{\&}\rangle$ is defined:

$$(u \bullet v \bullet w)\hat{\&}i := u(v\& \boxed{\mathrm{i}}^{|v|})w \tag{2.35}$$

where $\langle\&\rangle$ is interpreted as the traditional superposition operation of equal length strings, disregarding rules regarding repetitions.

Finally, these operations may be used in a model for $\langle\hat{@}\rangle$:

$$s\hat{@}i := \{s'\hat{\&}i \mid s\%^2 L \wedge s' \in L\} \tag{2.36}$$

This may be applied iteratively to superpose arbitrary n intervals, i.e. (2.39).

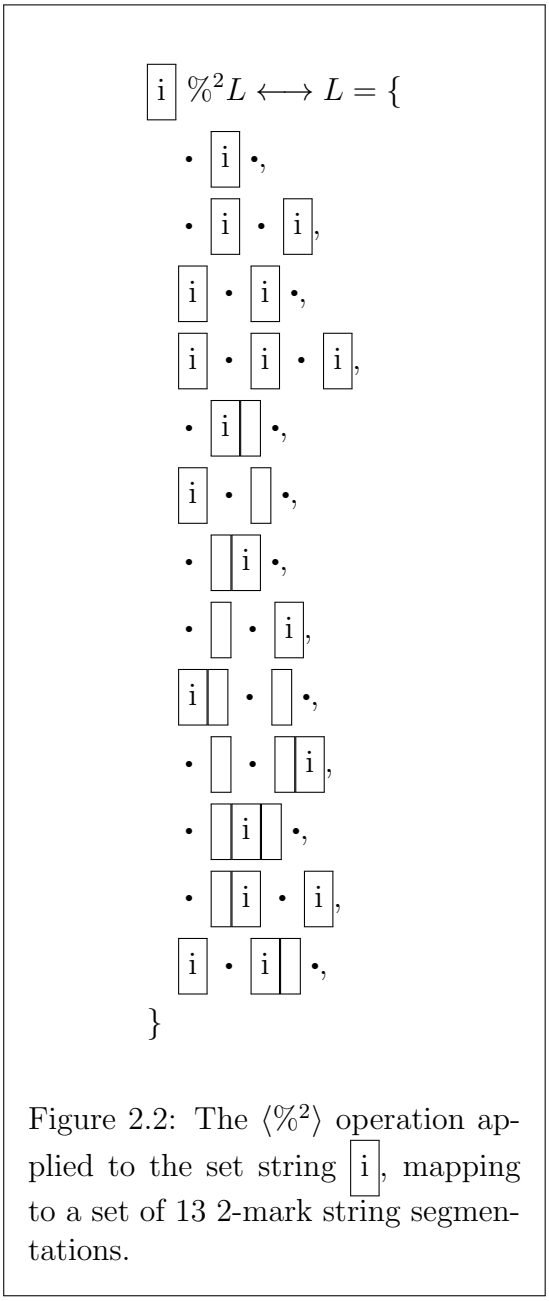

Figure 2.2: The $\langle \%^2 \rangle$ operation applied to the set string i, mapping to a set of 13 2-mark string segmentations.

$$L \hat{@} i := \bigcup_{s \in L} s \hat{@} i \tag{2.37}$$

$$X \hat{@} I := \bigcup_{i \in I} X \hat{@} i \tag{2.38}$$

$$\bigcirc\!\!\!\!\!@_{j=1}^{n} i_j = ((i_1 \ \hat{@} \ i_2) \hat{@} i_3) \ldots \tag{2.39}$$

(2.37) and (2.38) simply show how the $\langle \hat{@} \rangle$ operation is applied to languages. Since $\langle \hat{@} \rangle$ models the same result set as (2.1), except without redundant computation and duplicate results. We can consider them logically equivalent (recalling that an interval may be considered a set string of length 1, containing a singleton with the interval).

$$\hat{@} \models @$$

From here on, we may use the $\langle @ \rangle$ operation synonymously with $\langle \hat{@} \rangle$, as we will always be referring to the behaviour modelled by $\langle \hat{@} \rangle$.

2.2.2 Superposition in TimeML

Events in TimeML documents (TML) are intervals which are often tenuously linked to other intervals. In many cases, a cluster of events $\{i_1, \ldots, i_n\}$ are not related at all. The expressive power of the superposition operation may be applied here to describe such cases (2.43).

$$\bigcirc\!\!\!\!\!@_{i \in \{i_1, \ldots, i_n\}} \ i \tag{2.40}$$

(2.43) is for the case where $\{i_1, \ldots, i_n\}$ are related in no discernible way in the TML document. A lot of the time, two events will be connected in an ambiguous manner rather than an entirely unspecified manner. For example, we may only know that e_1 and e_2 are related to some e_3 by relations $r \in con$. From that, we can tell that the string representation for the document will contain at least one set $\alpha \supset \boxed{e_1, e_2}$. The Allen relation between e_1 and e_2 is ambiguous, as it could be anything in

$$r = \{d, di, o, oi, s, si, f, fi, =\} \quad (2.41)$$
$$= Allen - \{m, mi, <, >\} \quad (2.42)$$

(2.42) is arguably a better notation to use, as it is closer to English in the way it expresses the relation between e_1 and e_2, "r is any Allen relation, but not m, mi, $<$ or $>$".

Suppose, instead, $r = \{m, mi, <, >\}$. $(e_1\ r\ e_2)$ may be combined with basic superposition, through the use of language intersection. Say, we know that, other than the constraint on $(e_1\ r\ e_2)$, $\{e_1, \ldots, e_n\}$ are disconnected events. The language of strings which may represent the temporal network can be described by

$$\left(\bigcirc\!\!\!\!@ \limits_{e \in \{e_1, \ldots, e_n\}} e \right) \cap_{\mathcal{L}} \quad (2.43)$$

$$\left(\boxed{\top}^{a} \boxed{e_1, \neg e_2}\boxed{\neg e_1, \neg e_2}^{b} \boxed{e_2, \neg e_1}\boxed{\top}^{c} \cup_{\mathcal{L}} \right.$$

$$\left. \boxed{\top}^{a} \boxed{e_2, \neg e_1}\boxed{\neg e_1, \neg e_2}^{b} \boxed{e_1, \neg e_2}\boxed{\top}^{c} \right)$$

where $\top$ is the symbol for a fluent that always evaluates to true (a tautology), and $0 \leq a + b + c \leq 2n - 3$. We are now moving away from the interpretation of *language* as a set of strings, to language as a set of *fluents*, where fluents may be simple strings or expressions that evaluate to true or false for a string (each fluent models strings).

$$s \in_{\mathcal{L}} L \longleftrightarrow (\exists \varphi)(\varphi \in L \wedge \varphi \models s) \quad (2.44)$$

$$L \cup_{\mathcal{L}} L' \models \{s \mid s \in_{\mathcal{L}} L \vee s \in_{\mathcal{L}} L'\} \quad (2.45)$$

For example, $\neg e$ models any interval that is not e, whereas e models exactly one interval, e. Intervals can therefore be considered a special case of fluents modelling themselves alone.

The operation $\langle \cap_{\mathcal{L}} \rangle$, as you may have inferred from our description of $\langle \cup_{\mathcal{L}} \rangle$ (2.45), is used to intersect languages of fluents. A language of fluents $\mathcal{L}$ models some language of strings L. $\langle \cap_{\mathcal{L}} \rangle$ doesn't calculate the intersection of languages, but the intersection of languages *modelled* by the fluent language arguments.

$$L \cap_{\mathcal{L}} L' \models \{s \mid s \in_{\mathcal{L}} L \wedge s \in_{\mathcal{L}} L'\} \quad (2.46)$$

We have defined the operation $\langle @ \rangle$ extensively as either an operation between intervals, a string and an interval, or a language and an interval (or language of intervals). However, it would be useful if it could be an arbitrary operation between strings of any size, or languages thereof.

Without the limitation that the right hand side must model intervals (string of length one containing a singleton set), (2.43) could be rewritten as (2.47)

$$\Big(\mathop{@}_{e\in\{e_1,\ldots,e_n\}} e\Big)@_{\mathcal{L}}$$

$$\Big(\boxed{e_1, \neg e_2}\boxed{\neg e_1, \neg e_2}\boxed{e_2, \neg e_1} \cup_{\mathcal{L}}$$

$$\boxed{e_1, \neg e_2}\boxed{\neg e_1, \neg e_2}\boxed{e_2, \neg e_1} \cup_{\mathcal{L}}$$

$$\boxed{e_1, \neg e_2}\boxed{e_2, \neg e_1} \cup_{\mathcal{L}}$$

$$\boxed{e_1, \neg e_2}\boxed{e_2, \neg e_1}\Big) \tag{2.47}$$

(2.47) superposes a language of fluent strings s, $2 \leq |s| \leq 3$, over the language $(@_{e\in\{e_1,\ldots,e_n\}} e)$. This would involve new behaviour for the $\langle @ \rangle$ operation. Until now, $\langle @ \rangle$ has only ever been applied between languages or strings with disjoint alphabets, $\Sigma \cap \Sigma' = \emptyset$. Now it is being applied to languages with intersecting alphabets.

Proposition 1. With disjoint alphabets, $\langle @ \rangle$ may only increase the size of the resultant language.

With intersecting alphabets, $\langle @ \rangle$ must unify the corresponding parts in strings to maintain consistency with definition 1.2.1. Therefore we state Proposition 2

Proposition 2. $\langle @ \rangle$ may increase or decrease the size of the language for arguments with intersecting alphabets.

In (2.47) for example, the language $(@_{e\in\{e_1,\ldots,e_n\}} e)$ is decreased in size, since both e_1 and e_2 occur in every string, and the right hand side of $\langle @_{\mathcal{L}} \rangle$ is anchored at the right and left by either e_1 and e_2. Any string $s \in (@_{e\in\{e_1,\ldots,e_n\}} e)$, where $s \in_{\mathcal{L}} \boxed{\top}^* \boxed{e_1, e_2, \top}\boxed{\top}^*$, will be excluded in the superposition.

Fluent conjunction Before we go further, we should discuss how the basic superposition operation (on which the operation $\langle @_{\mathcal{L}} \rangle$ is built) will work; i.e. the $\langle \&_{\mathcal{L}} \rangle$ operation in (2.48)

$$s@_{\mathcal{L}}s' = \{s''\&_{\mathcal{L}}s' \mid s\%_0^{|s'|+1}L \wedge s'' \in L\} \tag{2.48}$$

Rather than using component-wise union across strings (as traditional superposition $\langle \& \rangle$ does), $\langle \&_{\mathcal{L}} \rangle$, uses an operation – with some parallels to set union – named *fluent conjunction.*

A fluent set in a temporal string, $\boxed{\varphi, \ldots, \varphi_n}$ can be interpreted as the logical conjunction of all the fluents in the set holding true during that time period, $\boxed{\bigwedge_{i=1}^{n} \varphi_i}$.

Like a knowledge base, temporal strings are a dynamic form of knowledge representation (knowledge of events, and relations between events); the knowledge base may expand to incorporate new knowledge (often reducing the size of the set modelled by the temporal string). If a set from a temporal string were represented in Prolog, one might be tempted to make use of an anonymous tail variable to allow for the string to grow, i.e. $[\varphi_1, \ldots, \varphi_n|\_]$. Because we can interpret any set S in a temporal string as a singleton set containing the conjunction of all fluents in S, we can state (2.49).

$$(\forall\alpha)(\forall\beta)(\alpha \in S \wedge \beta \in S) \rightarrow \alpha \wedge \beta \tag{2.49}$$

(2.49) would be unsatisfiable for $S = \boxed{e, \neg e}$, as $X \wedge \neg X$ is a contradiction. *Fluent conjunction* $\langle \sqcap_{\mathcal{L}} \rangle$ is an operation on fluent sets that performs union on the sets, then checks the resultant set for contradictions – returning $\emptyset$ should a contradiction occur, or a singleton containing the result of the union should no contradiction occur (2.50)

$$A \sqcap_{\mathcal{L}} B = \begin{cases} \{A \cup B\} & \text{if } \bigwedge_{\alpha \in A} \bigwedge_{\beta \in B} \alpha \wedge \beta \\ \emptyset & \text{otherwise} \end{cases} \tag{2.50}$$

Unlike traditional $\langle \& \rangle$, which operates with component-wise union, (2.24), and fails for input of unequal length, $\langle \&_{\mathcal{L}} \rangle$ makes use of fluent conjunction $\langle \sqcap_{\mathcal{L}} \rangle$, and fails for unsatisfiable fluent sets, as well as input of unequal length.

$$\left(\boxed{\alpha_1} \ \dots \ \boxed{\alpha_n}\right) \&_{\mathcal{L}} \left(\boxed{\beta_1} \ \dots \ \boxed{\beta_n}\right) = \boxed{\varphi_1} \ \dots \ \boxed{\varphi_n}$$
$$\text{where } \bigwedge_{i=1}^{n} \varphi_i \in (\alpha_i \sqcap_{\mathcal{L}} \beta_i) \tag{2.51}$$

Evidently, (2.51) would fail while superposing two strings of equal length where two corresponding positions in the strings cannot both be considered true, e.g. fluent conjunction of $\boxed{e}$ and $\boxed{\neg e}$.

Splicing and unifying As previously mentioned, a position in a temporal string can be thought of as a knowledge base. Furthermore, a temporal string can be thought of as a knowledge base modelling state transitions in a timeline, where a different knowledge base is considered for each state.

$$\boxed{KB1}\boxed{KB2} \longleftrightarrow KB1 \prec KB2$$

Definition 2.2.1. *Alphabet of a timeline* The alphabet of a timeline is the union of the alphabets of all the knowledge bases which are consistent at some point in the timeline.

The alphabet of a knowledge base is the collection of TBox assertions in the knowledge base. For example, the alphabet of $\boxed{a, \neg b}$ would not include b, as it is part of an ABox, whereas it would include a, as it is a factual assertion whose evaluation to true or false isn't derived from the state of the knowledge base (TBox).

Recalling Proposition 2, $\langle @ \rangle$ must be fitted with some behaviour to handle timeline arguments of intersecting alphabets. We call this behaviour *splicing* – taking two knowledge bases $KB1$ and $KB2$ and producing a third knowledge base KB' that is the combination of $KB1$ and $KB2$, capturing the parts that are shared by $KB1$ and $KB2$, and unifying the parts that are not shared. Inconsistent knowledge bases cannot be spliced. One of the central constraints for consistency across two temporal strings s and s' (time-

lines) is the relative order of the symbols from $\Sigma_s \cap \Sigma_{s'}$ in s and s' respectively. This constraint can, for explanatory purposes, be described by the *bc* and *trim* functions (2.52).

$$bc(trim(\pi_{\Sigma_s \cap \Sigma_{s'}} s)) = bc(trim(\pi_{\Sigma_s \cap \Sigma_{s'}} s')) \tag{2.52}$$

If we were to superpose the temporal strings for *witness A*'s testimony and *witness B*'s (as described in two TML document for example), we may first want to know whether these two sources are dictating consistent stories. A says that Han shot (h), later Greedo shot (g), then Han dodged (d). B says the same, but omits the detail d, and instead includes a detail that smoke appeared (s) between h and g. In short, A says $\boxed{h}\boxed{\ }\boxed{g}\boxed{d}$, and B says $\boxed{h}\boxed{s}\boxed{g}$. Although A and B differ, their testimonies are consistent, as $\boxed{h}\boxed{g}$ is the left and right result when both testimonies are inserted into the test (2.52). If a judge spliced the two testimonies together, the sequence of events (a knowledge base) $\boxed{h}\boxed{s}\boxed{g}\boxed{d}$ would be attained.

Demarcation for Δ To splice two strings, s and s', we first need to segment s and s' into corresponding demarcations based on the occurrences of *common ground* in both strings. In other words, given $\Delta = \Sigma_s \cap \Sigma'_s$ (common ground), we wish to break s and s' respectively into m and m' of equal length, where (2.53) holds for $\langle m, m' \rangle$.

$$\bigwedge_{i=1}^{|m,m'|} bc(trim(\pi_\Delta m_i)) = bc(trim(\pi_\Delta {m'}_i)) \tag{2.53}$$

We can borrow the symbol for demarcation used earlier ($\bullet$) to mark boundaries in m and m' where portions containing fragments of Δ begin and end. A segment α of a temporal string surrounded by demarcation should satisfy (2.54).

$$|bc(\pi_\Delta \alpha)| = 1 \tag{2.54}$$

Before going into the formal definition of Δ-demarcation, we will give an illustrative example.

$$\boxed{a}\boxed{a,b}\boxed{b}\boxed{c}\boxed{d}\boxed{e,f}$$

is $\{a, b, e\}$-demarcated as

$$\epsilon \bullet \boxed{a} \bullet \epsilon \bullet \boxed{a,b} \bullet \epsilon \bullet \boxed{b} \bullet \boxed{c}\boxed{d} \bullet \boxed{e,f} \bullet \tag{2.55}$$

The same string would have been $\{b, d\}$-demarcated as

$$\boxed{a} \bullet \boxed{a,b}\boxed{b} \bullet \boxed{c} \bullet \boxed{d} \bullet \boxed{e,f}$$

Δ-demarcation makes use of a 2-state system ("in Δ-segment" or "not in Δ-segment"), and projection π_Δ to demarcate a temporal string.

Δ-demarcation of a string s is formally described by the function $f(s, \Delta)$ in (2.56). The state for "in Δ-segment"

is described by the g function (2.57). The demarcation process begins "outside" a Δ-segment, even if the first position in the temporal string contains a TBox from Δ – this would result in a Δ-demarcated string with a point-marker at the beginning (e.g. (2.55)).

$$f(\alpha\beta, \Delta) = \begin{cases} \epsilon & \text{if } \alpha\beta = \epsilon \\ \bullet.\alpha.g(\beta, x, \Delta) & \text{otherwise if} \\ & \quad |\alpha| = 1 \wedge \\ & \quad \pi_\Delta \alpha = x' \wedge \\ & \quad x' \neq \emptyset \\ \alpha.f(\beta, \Delta) & \text{otherwise for} \\ & \quad |\alpha| = 1 \end{cases} \tag{2.56}$$

$$g(\alpha\beta, x, \Delta) = \begin{cases} \bullet & \text{if } \alpha\beta = \epsilon \\ \alpha.g(\beta, x, \Delta) & \text{otherwise if} \\ & \quad |\alpha| = 1 \wedge \\ & \quad \pi_\Delta \alpha = x \\ \bullet.\bullet.\alpha.g(\beta, x, \Delta) & \text{otherwise if} \\ & \quad |\alpha| = 1 \wedge \\ & \quad \pi_\Delta \alpha = x' \wedge \\ & \quad x' \neq \emptyset \\ \bullet.\alpha.f(\beta, \Delta) & \text{otherwise for} \\ & \quad |\alpha| = 1 \end{cases} \tag{2.57}$$

Proposition 3. The Δ-demarcation (2.56) of two consistent (2.53) strings s and s', with respective alphabets Σ_s and $\Sigma_{s'}$, $\Sigma_s \cap \Sigma_{s'} = \Delta$, produces demarcations m and m', $|bc(trim(\pi_{\{\bullet\}} m))| = |bc(trim(\pi_{\{\bullet\}} m'))|$.

A Δ-segment of a temporal string (s) is a portion w of s, such that $\pi_\Delta w = \boxed{\varphi}^n$. This means a portion of the string that holds common ground φ with Δ throughout, and φ is constant across w.

Indexing a Δ-demarcation All Δ-demarcations m of string s take the form $w(\bullet w)^*$ for $w \in (2^{\Sigma_s})^*$ (Note that w may be ϵ). The first w in the expression is not a Δ-segment, as it is not surrounded by "$\bullet$". This first segment w has index 1. From the definition of Δ-demarcation (2.56), we know that demarcated Δ-segments are surrounded by non-Δ-segments, and non-Δ-segments are not demarcated from other non-Δ-segments (i.e. their neighbours consist only of Δ-segments). In other words, the point-markers demarcate the string s into segments alternating between Δ-segments and non-Δ-segments. We know that the first w is a non-Δ-segment, therefore we know that all odd indexes are non-Δ-segments, and all even indexes are Δ-segments.

Expressing $w(\bullet w)^*$ in the form of a 3-state finite state automata, and adding an additional dimension of symbols to the edges to build the transducer T (Figure 2.3), a good descriptive model is gained for the process by which an index in the demarcation string is found.

$$\begin{aligned} &\text{for } s = uiv,\ \neg marked(i),\ T(ui) = s' \\ &\quad index(i) = |s'| \end{aligned} \tag{2.58}$$

Disjoint superposition The original basic case of superposition by fluent conjunction (2.48), where the alphabets of the arguments were disjoint, will now be classed as a special case of superposition only applicable to strings sharing no common alphabet symbols. We give

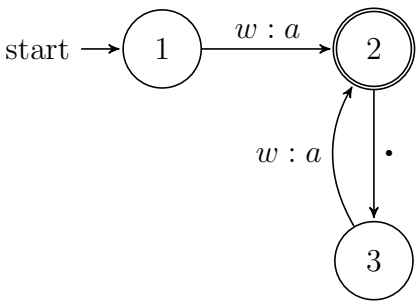

Figure 2.3: The finite state transducer T, outputting a string a^n equal in length to the number of segments in the input demarcated string

this type of superposition operation the symbol $\langle\tilde{@}\rangle$.

Anchored superposition Anchored superposition, in contrast to disjoint superposition, is classed as the superposition of strings with intersecting alphabets, and both strings are anchored to the same segment of a timeline. The way this is done is by the fluents intersecting to the left and right of the strings being superposed. Given string argument s and s', $\Sigma_{left(s)\cap_{\mathcal{L}}left(s')} \neq \emptyset$, and $\Sigma_{right(s)\cap_{\mathcal{L}}right(s')} \neq \emptyset$. The symbol $\langle@_{\Delta}\rangle$ can be used for *anchored superposition*. The reason the subscript Δ is chosen is because the Δ-segments described previously are precisely where this form of superposition is applied.

Anchored superposition is carried out between string segments s and s', whose shared alphabet Δ is a subset of each set at every respective position in s and s'. E.g. $\boxed{x}\boxed{x,y}\boxed{x}$ and $\boxed{x,d}\boxed{x,c}$ for $\Delta = x$. Since the left and right markers of these two strings must align in the combined superposition, the $\langle\%^n\rangle$ operation isn't used at all. Instead positions in either s or s' may be split so that the two strings align in length. At this point, safe superposition $\langle\&\rangle$ (2.27) is applied across the strings. The maximum number of *splits* allowed in a string s is the length of the other string s' with 1 added. The shorter of the two strings must be split until both strings align in length, thus at least $|s| - |s'|$ splits are necessary for the short of the two strings, s'.

Superposing temporal strings s and s' of possibly unequal length, with possible intersections between both alphabets is a process involving many steps which have been described individually extensively at this point. The entire process is put into perspective in the ordered list of steps below:

(a) Finding the intersection Δ of Σ_s and $\Sigma_{s'}$ (where Σ_x denotes the alphabet of timeline string x; Definition 2.2.1)

(b) Determining whether s and s' are consistent, using Δ (2.52)

(c) Creating demarcations m and m' for the string s and s' respectively, based on the occurrences of $\alpha \in \Delta$ in both strings (2.56)

(d) Mapping these demarcations to one another (both segmentations should be of equal length, and neither should begin with a segment containing $\alpha \in \Delta$) (Proposition 3)

(e) Applying

(i) $\langle\tilde{@}\rangle$ between segments at odd positions, and

(ii) $\langle@_\Delta\rangle$ between segments at even positions

and concatenating the results together.

2.2.3 Splicing: A Useful Temporal String Operation

At this point, many new operations have been described which attempt to model not only the superposition of equal length, disjoint strings, but the superposition of variable length strings with segments of common ground. This attempt to model a more robust form of superposition for the purposes of reasoning with TML documents is arguably no longer described properly by the term *superposition*.

It is suggested that the term *splicing* be used to describe the sum of the steps outlined in the previous section. Furthermore, it is noted that splicing, unlike the *interval-on-string superposition* operation from which it derives, is a transitive and commutative relation. Yet it can be used either to superpose basic intervals over strings, or strings over other strings. It could also be used iteratively to splice temporal string *languages* together. Due to the application of the new "demarcation" approach to both superposition (in the $\langle\%^n\rangle$ operation), and splicing (in Δ-demarcation), much of the redundant computation (duplicate results, reliance on larger, non-finite data structures) present in traditional superposition using the *bc* and *trim* functions is addressed. This *universal splicing* operation can be used to build up a knowledge base from temporal relations between intervals.

Since much of the operations we used intermediately to arrive at the construction of *splice* are now replaceable by the splice operation, some symbols should be recycled.

It should be stated exactly what the splicing model consists of at this point: The symbol $\langle@\rangle$ is adopted to describe the universal splicing operation, as its prior usage can be replaced by the splice operation.

$$
\begin{aligned}
&s = \alpha_1 \dots \alpha_n \\
&s' = \beta_1 \dots \beta_n \\
&\Sigma_s = \bigcup_{\alpha \in s} \Sigma_\alpha \\
&\Sigma'_s = \bigcup_{\beta \in s'} \Sigma_\beta \\
&\Delta = \Sigma_s \cap \Sigma'_s \\
&m = f(s, \Delta) = m_1 \dots m_p \\
&m' = f(s', \Delta) = m'_1 \dots m'_p \\
&s@s' = \\
&\quad (m_1 \tilde{@} m'_1).(m_2 @_\Delta m'_2). \\
&\quad (m_3 \tilde{@} m'_3).(m_4 @_\Delta m'_4) \dots
\end{aligned}
$$

Since $\langle@\rangle$ can be used to splice together temporal relations described in string format, we can describe the process by which a TML document is mod-

elled as a string representations (2.59).

$$\bigcirc\!\!\!\!@_{R \in DOC} \; s_R \models \mathcal{F} \tag{2.59}$$

where $\mathcal{F}$ is a language of temporal string representations describing the temporal events and relations in the document, and s_R is an Allen relation in string format, as outlined in Fernando (2013). This language is not only regular, but finite.

2.3 Temporal Taxonomies

During the construction of a string representation of events, the special Allen relation sets *dur* and *con* are of particular importance. Given a TML document, annotating a news item, we'd like to eventually represent the relation between events in a format that is both human-readable and equivalent to MSO, e.g.

$$s = \boxed{wc, m_1}\boxed{wc, m_1, g_1}\boxed{wc, m_1} \ldots \boxed{wc, m_3} \tag{2.60}$$

The string in (2.60) may be thought of as representing a news article relating to the World Cup (wc). The TML markup specifies that several matches, $m_1 \ldots m_3$, occur during the world cup, and several (or no) goals ($g_1 \ldots g_n$) occur in each game. It also specifies the relative order of the matches and the goals. TML may even state that a few matches began and ended at the same time. This is valid TML, and valid TML should has a valid string representation. Reaching that representation isn't so trivial, and the order in which we try to build s will determine how fast we reach a valid representation. Once we apply constraint propagation, we may find that all goals that occur during matches occurring during the world cup, are also event intervals that occur during the world cup. However, to begin our construction of s by building a string encapsulating the relation between wc and $g_1 \ldots g_n$ would be unwise, as we'll later have to greatly modify such a string to integrate the matches contained (immediately) during wc. Taxonomies are acyclic ontologies with only one label possible (*contains* in our case). Thus they model a small fragment of what an ontology can express. We define *temporal taxonomy* to be a tree-like structure depicting the composition of intervals. A non-hierarchical taxonomy is ideal for our purposes, as temporal composition cannot (to our knowledge) be cyclical; a *contains* $b \rightarrow \neg(b$ *contains* $a)$, and the contains relation is transitive.

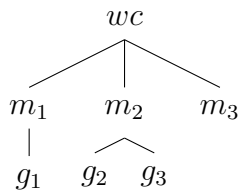

Figure 2.4: The temporal taxonomy for the string in (2.60)

The purpose of this representation

is to reveal in what order we should consider our intervals for addition to a string representation. We begin at the leaves of the tree, stripping away each layer sequentially, until we arrive at the root. Later, we will explain why this is a good approach to building strings from interval networks.

2.3.1 An algorithm for Hierarchical Taxonomies

Terminology A *holds* relation between two intervals is one that is a subset of Allen's special *dur* relation, which was the ambiguous set $\{d, s, f\}$. A *held-by* relation between two intervals is one that is a subset of Allen's special *con* relation, $\{di, si, fi\}$.

$(\forall a)(\forall b) a \; holds \; b \longleftrightarrow b \; held\text{-}by \; a$

Reducing the temporal network Reducing an Allen interval network (ontology) to intransitive taxonomy involves removing all relations (edges) that are neither *holds* nor *held-by* relations. We then remove all nodes that are no longer connected to any other node by a relation. Finally the *held-by* relations between any pair of nodes $\langle \eta_s, \eta_d \rangle$ are replaced with a relation *holds* between the nodes in reverse, $\langle \eta_d, \eta_s \rangle$ (should this *holds* relation already exist in the network, a simple deletion of the *held-by* is carried out). What remains is an intransitive taxonomy of the *holds* relations.

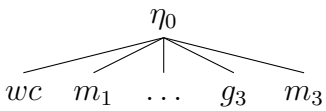

Figure 2.5: The starting point of the algorithm for building up a hierarchical taxonomy when applied to the "World Cup" TML document example

To further reduce this to a transitive non-hierarchical taxonomy, we can apply iteratively an algorithm for building up a taxonomy from an input stream of rules. We begin with all intervals in an alphabet placed as children to a master node (Figure 2.5)

A non-hierarchical taxonomy can be represented in Prolog as a list of triples

$$node : children : descendents$$

where *children* and *descendents* are lists of nodes. The *descendents* list is redundant, but it serves a purpose while updating the taxonomy. The starting state of the taxonomy shown in Figure 2.5 can be depicted in Prolog with the list (2.61)

$$\begin{array}{l}[\\ \quad \eta_0 : [wc, m_1, g_3, m_3|X] : [wc, m_1, g_3, m_3|X], \\ \quad wc : [\,] : [\,], \\ \quad m_1 : [\,] : [\,], \\ \quad g_3 : [\,] : [\,], \\ \quad m_3 : [\,] : [\,]|\_ \\ ] \end{array} \tag{2.61}$$

The process by which the initial taxonomy is built from an alphabet of inter-

vals, and modified by a list of *holds* relations, can be described in Prolog as follows:

```
relset2taxonomy(RelSet,A,Taxo) :-
   initialtaxo(A,Init),
   in2taxonomy(RelSet,Init,Taxo).
in2taxonomy([],X,X).
in2taxonomy([H|T],In,Out) :-
   morphtaxtree(In,Mid,H),
   in2taxonomy(T,Mid,Out).
initialtaxo(Alpha,[H|Nodes]) :-
   H = e0:Alpha:Alpha,
   initaxnodes(Alpha,Nodes).
initaxnodes([],[]).
initaxnodes([H|A],[H:[]:[]|B]) :-
  initaxnodes(A,B).
```

Listing 2: Building a taxonomy from a relation set in Prolog

in2taxonomy simply adds a set of *holds* relations iteratively to the initial taxonomy, by a predicate for updating the taxonomy, titled *morphtaxtree*. *morphtaxtree* can be described as an algorithm (Algorithm 3), though it can also be defined declaratively in Prolog.

Line 2 of Algorithm 3 deletes the node A from the taxonomy, and places the deleted node and related information (children and descendants) into the two variable $ASub$ and $ADes$, respectively. If B is in the descendants of A, then we need not add this information to the taxonomy, so T is returned to its original state (on entry to the procedure), and the story ends there.

Algorithm 3 Adds a relation to a non-hierarchical taxonomy, while maintaining consistency

1: **procedure** MORPH($T, A \rightarrow B$)
2: $\_ : ASub : ADes \leftarrow T.del(A)$
3: **if** $B \in ADes$ **then**
4: $T \leftarrow T \cup \{A : ASub : ADes\}$
5: **else**
6: $\_ : \_ : BDes \leftarrow T.find(B)$
7: $X \leftarrow \{B\} \cup BDes$
8: $Des' \leftarrow ADes \cup X$
9: $Sub' \leftarrow \{B\} \cup (ASub - BDes)$
10: $A' \leftarrow A : Sub' : Des'$
11: $T' \leftarrow \{A'\}$
12: **for** $N : C : D \in T$ **do**
13: $U(T', A, B, X, N : C : D)$
14: **end for**
15: $T \leftarrow T'$
16: **end if**
17: **end procedure**
18: **procedure** U($T, A, B, X, N : C : D$)
19: **if** $A \in D$ **then**
20: **if** $B \in D$ **then**
21: **if** $B \in C$ **then**
22: $Y \leftarrow C - X$
23: $T' \leftarrow T' \cup \{N : Y : D\}$
24: **else**
25: $T' \leftarrow T' \cup \{N : C : D\}$
26: **end if**
27: **else**
28: $T' \leftarrow T' \cup \{N : C : (D \cup X\}$
29: **end if**
30: **else**
31: $T' \leftarrow T' \cup \{N : C : D\}$
32: **end if**
33: **end procedure**

Otherwise, new information is being added, so we continue in the else clause. Line 6 finds the B node in the taxonomy, and places its *descendants* set in the variable $BDes$. B is added to both the children and descendants sets of A at this point (lines 8 and 9). After these sets are updated, we must also update the rest of the taxonomy, wherever A occurs in the descendants, and wherever B occurs as a child or descendent; this is the purpose of the iterative call to U (*update*).

Disregarding knowledge of the interval alphabet, which is used to build the initial taxonomy, this algorithm for building a transitive non-hierarchical taxonomy is *online*, meaning it may begin running before the entire input stream of *holds* relations is passed to it. In fact, passing the alphabet to an *initial taxonomy* in advance, can be avoided by building the *initial taxonomy* gradually, as the alphabet of the interval network becomes known. We simply build the initial taxonomy in advance in this paper for explanatory purposes. Fernando (2016) notes that in many applications of finite state techniques, the alphabet of a machine, grammar, sentence etc. is treated as a dynamic construct, which may be enlarged gradually (in an online algorithm such as Algorithm 3, for example).

The version of Algorithm 3 for a dynamic alphabet (realised gradually via the stream of *holds* rules) would involve checking if the taxonomy contains any such A before line **??**. If it doesn't, simply add $A : [B] : [B]$ to the taxonomy, update η_0 to contain A as both a child and a descendent (removing B from the children of η_0 if possible).

```
morphtaxtree(In,Out,[A,B]) :-
  rmember(A:AChildren:ADesc,In,Y),
  (member(B,ADesc) ->
    In = Out;
    member(B:_:BDesc,Y),
    union(ADesc,[B|BDesc],Z),
    set_diff(AChildren,BDesc,X),
    ChildrenPlusB = [B|X],
    NewA = A:ChildrenPlusB:Z,
    findall(Row,
      updatetax([A,B],Z,Y,Row),
      T),
    Out = [NewA|T]).
updatetax([A,B],BDesc,In,Out) :-
  member(Node:Sub:Desc,In),
  (member(A,Desc) ->
    (member(B,Desc) ->
      (member(B,Sub) ->
        set_diff(Sub,BDesc,New),
        Out = Node:New:Desc;
        Out = Node:Sub:Desc);
      union(Desc,BDesc,NewDesc),
      Out = Node:Sub:NewDesc
    );
    Out = Node:Sub:Desc
  ).
```

Listing 3: Algorithm 3 in Prolog

Listing 3 is the declarative approach to the same process described by Algorithm 3, replacing iteration, by the use of the *member* predicate. The *rmember* predicate is similar to *member*, but also

unifies a third argument with a version of the list (or set), excluding a member (should one exist). There is some significant time advantage to an imperative approach, however, as set opeations like *symmetric set difference*, can be implemented with data structures which permit $O(1)$ operations.

Analysis of the taxonomy building algorithm The main advantage of Algorithm 3 is its *online* properties. It can used as the kernel of an algorithm for building up taxonomies dynamically from a stream of data. To achieve this, each call to $Morph$ takes $O(n)$ worst-case asymptotic time to execute on a taxonomy containing n nodes. This means, the larger the taxonomy grows, the longer it will take to update it with new information. This analysis is based on the single loop featured in Algorithm 3, line 12.

2.3.2 Application of the Temporal Taxonomy

We apply the temporal taxonomy to the process for generating strings from Allen interval networks, by using it to generate heuristics.

Terminology An interval i is embedded in an interval η if η *holds* i. Recalling that *holds* is a transitive relation, we may refer to an *embedding degree* of i as the maximal length chain of *holds* relations leading from a reference point η_0 to i. For example, $\langle \eta_0 \; holds \; \eta_1 \; holds \; \eta_2 \; holds \; i \rangle$, being a maximal length chain from η_0 to i, means that i has an embedding degree of 3 (with reference point η_0).

The embedding degree is the heuristic used to determine in which order we should construct strings for event networks. The most deeply nested intervals (highest embedding degree) are examined, and their respective string or string sets (should ambiguity exist) are computed. The next most embedded are then addressed, building upon the results of the previously computed strings or string sets.

The temporal taxonomy is very useful here, as it models the nested structure of intervals, and the next round according to the heuristic is implicitly described by the tree structure; we deal with the leaves, then we trim the leaves, then we repeat the process, until we reach the root.

Chapter 3

Conclusion

The aim of this paper was to connect TimeML to MSO. It is indeed achievable through the use of the temporal string representation to place parts of a TML document into MSO. The parallels between ISO-TimeML and Allen interval logic are stark, though not fully utilised in the corpora available. At the beginning of this paper, statistical analysis of the corpora was brought forward, and it appeared that all TimeML corpora had a predisposition towards the usage of certain fragments of the standard, but not others. Many of the news items in the corpora were not annotated in a consistent manner – some were annotated so inconsistently that Allen constraint propagation failed (recall that this would mean a very visible logical flaw between a set of three events linked in the text). Others were labelled inconsistently, but not evidently (figuring this out is NP-hard). Some were marked consistently.

TimeML is robust, not all parts of the language have been explored by corpus linguists. It wasn't possible to look at the language as a whole – there are simply too many purposes to consider, and too little data to work with. For the fragments of TimeML which were examinined (events, durations as intervals and temporal relations), success was found in the creation of a new operation that makes the construction and combination of temporal strings straightforward.

It is hoped that the *splice* operation outlined in this paper can be used, in conjunction with the taxonomy-based heuristic outlined, to form temporal string representations from well-structured TimeML documents, which can then be used in formal verification techniques, or to form visualisations of events. These strings are logically equivalent to MSO.

A handful of smaller problems met along the way were also addressed, such as the suggested optimisations and corrections for the Allen constraint propagation algorithm, the in-place approach to superposition using segmentation, and the *online* algorithm for building up a non-hierarchical taxonomy from either a preset or dynamic alphabet.

Acknowledgements

I must credit Dr. Tim Fernando for his continual guidance during the months leading up to the completion of this thesis, and also Dr. Khurshid Ahmad for his constructive criticism during the demonstration period.

Bibliography

Allen, J. F. (1983). Maintaining Knowledge about Temporal Intervals. *Communications of the ACM*, *26*.

Bittar, A., Amsili, P., Denis, P., & Danlos, L. (2011). French TimeBank: An ISO-TimeML Annotated Reference Corpus. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics:shortpapers* (Vol. 49, p. 130-134). Portland,Oregon: Association for Computational Linguistics.

Büchi, J. R. (1962). On a Decision Method in Restricted Second Order Arithmetic. In *Logic, Methodology and Philosophy of Science: Proceedings of the 1960 International Congress* (p. 1-11). Stanford University Press.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). The MIT Press.

Fernando, T. (2002). A Finite-State Approach to Event Semantics. In *Proceedings of the Ninth International Symposium on Temporal Representation and Reasoning (TIME'02)* (Vol. 9). Association for Computer Science.

Fernando, T. (2004). A Finite-State Approach to Events in Natural Language Semantics. *Journal of Logic and Computation*, *14*, 79-92.

Fernando, T. (2012). A Finite-State Temporal Ontology and Event-Intervals. In *Proceedings of the 10th International Workshop on Finite State Methods and Natural Language Processing* (Vol. 10, p. 80-89). Donastia, San Sebastián: Association for Computational Linguistics.

Fernando, T. (2013). Segmenting Temporal Intervals for Tense and Aspect. In *Proceedings of the 13th Meeting on the Mathematical Language* (Vol. 13, p. 30-40). Sofia, Bulgaria: Association for Computer Science.

Fernando, T. (2016). On Regular Languages Over Power Sets. In (Vol. 4, p. 29-56).

James, P., Lee, K., Bunt, H., & Romary, L. (2010). ISO-TimeML: An International Standard for Semantic Annotation..

Pustejovsky, J., Gaizauskas, R., & Katz, G. (2002). TimeML: Robust Specification of Event and Temporal Expressions in Text. *IWCS-5*.

Pustejovsky, J., Hanks, P., Saurí, R., See, A., Gaizauskas, R., Setzer, A., ... Lazo, M. (2003). The timebank corpus..

Modified Allen Constraint Propagation

```
int allenAdd( int i, int r, int j, AllenNet fsa ) {
  for (int x = 0; x < fsa.size() * fsa.size(); x++)
    update[x] = false;
  toDoQ.enqueue(i);
  toDoQ.enqueue(j);
  int lim, n, tr;
  int *ks;
  fsa.setRel(i, r, j);
  do {
    i = toDoQ.dequeue();
    j = toDoQ.dequeue();
    r = fsa.rel(i, j);
    update[i * fsa.size() + j] = false;
    ks = fsa.right(i);
    for (int k = 0; k < fsa.size(); k++) {
      n = fsa.rel(k, j);
      tr = constraints(invertRel(ks[k]), r) & n;
      // Proper subset
      if ((tr & n) == tr && tr != n) {
        if (!update[k * fsa.size() + j]) {
          update[k * fsa.size() + j] = true;
          toDoQ.enqueue(k);
          toDoQ.enqueue(j);
        }
        fsa.setRel(k, tr, j);
      }
    }
    ks = fsa.right(j);
    for (int k = 0; k < fsa.size(); k++) {
      n = fsa.rel(i, k);
      tr = constraints(r, ks[k]) & n;
      // Proper subset
      if ((tr & n) == tr && tr != n) {
        if (!update[i * fsa.size() + k]) {
          update[i * fsa.size() + k] = true;
          toDoQ.enqueue(i);
          toDoQ.enqueue(k);
        }
        fsa.setRel(i, tr, k);
```

```
      }
    }
  } while (!toDoQ.isEmpty());
}
int constraints( int R1, int R2 ) {
  int constr = 0;
  int R2T;
  for (int i = 0; R1 != 0; i++) {
    if (R1 & 1) {
      R2T = R2;
      for (int j = 0; R2T != 0; j++) {
        if (R2T & 1) {
          constr |= TRANS_TABL[i][j];
        }
        R2T >>= 1;
      }
    }
    R1 >>= 1;
  }
  return constr;
}
```

DLD File for TimeML

```
<!ELEMENT TimeML ( #PCDATA | ALINK | CONFIDENCE | EVENT |
        MAKEINSTANCE | SIGNAL | SLINK | TIMEX3 | TLINK )* >
<!ATTLIST TimeML xsi:noNamespaceSchemaLocation CDATA #IMPLIED >
<!ATTLIST TimeML xmlns:xsi CDATA #IMPLIED >
<!ATTLIST TimeML comment CDATA #IMPLIED >

<!ELEMENT EVENT ( #PCDATA ) >
<!ATTLIST EVENT eid ID #REQUIRED >
<!ATTLIST EVENT class ( ASPECTUAL | I_ACTION | I_STATE |
        OCCURRENCE | PERCEPTION | REPORTING | STATE ) #REQUIRED >
<!ATTLIST EVENT stem CDATA #IMPLIED >
<!ATTLIST EVENT comment CDATA #IMPLIED >

<!ELEMENT MAKEINSTANCE EMPTY >
<!ATTLIST MAKEINSTANCE eiid ID #REQUIRED >
<!ATTLIST MAKEINSTANCE eventID IDREF #REQUIRED >
```

```
<!ATTLIST MAKEINSTANCE signalID IDREF #IMPLIED >
<!ATTLIST MAKEINSTANCE pos ( ADJECTIVE | NOUN | VERB |
        PREPOSITION | OTHER | UNKNOWN ) #REQUIRED >
<!ATTLIST MAKEINSTANCE tense ( FUTURE | INFINITIVE |
        NONE | PAST | PASTPART | PRESENT | PRESPART ) #REQUIRED >
<!ATTLIST MAKEINSTANCE aspect ( NONE | PERFECTIVE |
        PERFECTIVE_PROGRESSIVE | PROGRESSIVE ) #REQUIRED >
<!ATTLIST MAKEINSTANCE cardinality CDATA #IMPLIED >
<!ATTLIST MAKEINSTANCE polarity ( POS | NEG ) #REQUIRED >
<!ATTLIST MAKEINSTANCE modality CDATA #IMPLIED >
<!ATTLIST MAKEINSTANCE comment CDATA #IMPLIED >

<!ELEMENT TIMEX3 ( #PCDATA ) >
<!ATTLIST TIMEX3 tid ID #REQUIRED >
<!ATTLIST TIMEX3 type ( DATE | DURATION | SET | TIME ) #REQUIRED >
<!ATTLIST TIMEX3 value NMTOKEN #REQUIRED >
<!ATTLIST TIMEX3 anchorTimeID IDREF #IMPLIED >
<!ATTLIST TIMEX3 beginPoint IDREF #IMPLIED >
<!ATTLIST TIMEX3 endPoint IDREF #IMPLIED >
<!ATTLIST TIMEX3 freq NMTOKEN #IMPLIED >
<!ATTLIST TIMEX3 functionInDocument ( CREATION_TIME |
        EXPIRATION_TIME | MODIFICATION_TIME | PUBLICATION_TIME |
        RELEASE_TIME | RECEPTION_TIME | NONE ) #IMPLIED >
<!ATTLIST TIMEX3 mod ( BEFORE | AFTER | ON_OR_BEFORE |
        ON_OR_AFTER | LESS_THAN | MORE_THAN | EQUAL_OR_LESS |
        EQUAL_OR_MORE | START | MID | END | APPROX )  #IMPLIED >
<!ATTLIST TIMEX3 quant CDATA #IMPLIED >
<!ATTLIST TIMEX3 temporalFunction ( false | true ) #IMPLIED >
<!ATTLIST TIMEX3 valueFromFunction IDREF #IMPLIED >
<!ATTLIST TIMEX3 comment CDATA #IMPLIED >

<!ELEMENT SIGNAL ( #PCDATA ) >
<!ATTLIST SIGNAL sid ID #REQUIRED >
<!ATTLIST SIGNAL comment CDATA #IMPLIED >

<!ELEMENT ALINK EMPTY >
<!ATTLIST ALINK lid ID #REQUIRED >
<!ATTLIST ALINK relType ( CONTINUES | CULMINATES |
        INITIATES | REINITIATES | TERMINATES ) #REQUIRED >
<!ATTLIST ALINK eventInstanceID IDREF #REQUIRED >
```

```
<!ATTLIST ALINK relatedToEventInstance IDREF #REQUIRED >
<!ATTLIST ALINK signalID IDREF #IMPLIED >
<!ATTLIST ALINK syntax CDATA #IMPLIED >
<!ATTLIST ALINK comment CDATA #IMPLIED >

<!ELEMENT SLINK EMPTY >
<!ATTLIST SLINK lid ID #REQUIRED >
<!ATTLIST SLINK relType ( CONDITIONAL | COUNTER_FACTIVE |
        EVIDENTIAL | FACTIVE | MODAL | NEG_EVIDENTIAL ) #REQUIRED >
<!ATTLIST SLINK eventInstanceID NMTOKEN #REQUIRED >
<!ATTLIST SLINK subordinatedEventInstance NMTOKEN #REQUIRED >
<!ATTLIST SLINK signalID NMTOKEN #IMPLIED >
<!ATTLIST SLINK syntax CDATA #IMPLIED >
<!ATTLIST SLINK comment CDATA #IMPLIED >

<!ELEMENT TLINK EMPTY >
<!ATTLIST TLINK lid ID #REQUIRED >
<!ATTLIST TLINK relType ( BEFORE | AFTER | INCLUDES |
        IS_INCLUDED | DURING | DURING_INV | SIMULTANEOUS |
        IAFTER | IBEFORE | IDENTITY | BEGINS | ENDS | BEGUN_BY |
        ENDED_BY ) #REQUIRED >
<!ATTLIST TLINK eventInstanceID IDREF #IMPLIED >
<!ATTLIST TLINK timeID IDREF #IMPLIED >
<!ATTLIST TLINK relatedToEventInstance IDREF #IMPLIED >
<!ATTLIST TLINK relatedToTime IDREF #IMPLIED >
<!ATTLIST TLINK signalID IDREF #IMPLIED >
<!ATTLIST TLINK origin CDATA #IMPLIED >
<!ATTLIST TLINK syntax CDATA #IMPLIED >
<!ATTLIST TLINK comment CDATA #IMPLIED >

<!ELEMENT CONFIDENCE EMPTY >
<!ATTLIST CONFIDENCE tagType ( EVENT |
        MAKEINSTANCE | TIMEX3 | SIGNAL |
        ALINK | SLINK | TLINK ) #REQUIRED >
<!ATTLIST CONFIDENCE tagID IDREF #REQUIRED >
<!ATTLIST CONFIDENCE attributeName CDATA #IMPLIED >
<!ATTLIST CONFIDENCE confidenceValue CDATA #REQUIRED >
<!ATTLIST CONFIDENCE comment CDATA #IMPLIED >
```

French TimeML DLD

```
<!ELEMENT TimeML ( ALINK | SLINK | TEXT | TLINK )* >

<!ELEMENT TEXT ( #PCDATA | EVENT | SIGNAL | TIMEX3 )* >

<!ELEMENT EVENT ( #PCDATA ) >
<!ATTLIST EVENT aspect ( IMPERFECTIVE | PERFECTIVE |
        PERFECTIVE_PROGRESSIVE | PROGRESSIVE |
        PROSPECTIVE ) #IMPLIED >
<!ATTLIST EVENT class ( ASPECTUAL | CAUSE | EVENT_CONTAINER |
        I_ACTION | I_STATE | MODAL | OCCURRENCE | PERCEPTION |
        REPORTING | STATE ) #REQUIRED >
<!ATTLIST EVENT cardinality NMTOKEN #IMPLIED >
<!ATTLIST EVENT eid NMTOKEN #REQUIRED >
<!ATTLIST EVENT eiid ID #REQUIRED >
<!ATTLIST EVENT modality ( NECESSITY | OBLIGATION | PERMISSION
        | POSSIBILITY ) #IMPLIED >
<!ATTLIST EVENT mod ( START | MID | END ) #IMPLIED >
<!ATTLIST EVENT mood ( CONDITIONAL | SUBJUNCTIVE ) #IMPLIED >
<!ATTLIST EVENT polarity ( NEG | POS ) #IMPLIED >
<!ATTLIST EVENT pos ( ADJECTIVE | NOUN | PREPOSITION | VERB ) #REQUIRED >
<!ATTLIST EVENT pred CDATA #REQUIRED >
<!ATTLIST EVENT tense ( FUTURE | IMPERFECT | PAST | PRESENT ) #IMPLIED >
<!ATTLIST EVENT vform ( GERUNDIVE | INFINITIVE | PASTPART |
        PRESPART ) #IMPLIED >

<!ELEMENT SIGNAL ( #PCDATA ) >
<!ATTLIST SIGNAL sid ID #REQUIRED >

<!ELEMENT TIMEX3 ( #PCDATA ) >
<!ATTLIST TIMEX3 anchorTimeID IDREF #IMPLIED >
<!ATTLIST TIMEX3 beginPoint IDREF #IMPLIED >
<!ATTLIST TIMEX3 endPoint IDREF #IMPLIED >
<!ATTLIST TIMEX3 functionInDocument ( CREATION_TIME |
        EXPIRATION_TIME | MODIFICATION_TIME | PUBLICATION_TIME |
        RELEASE_TIME | RECEPTION_TIME | NONE ) #IMPLIED >
<!ATTLIST TIMEX3 freq NMTOKEN #IMPLIED >
<!ATTLIST TIMEX3 mod ( BEFORE | AFTER | ON_OR_BEFORE |
        ON_OR_AFTER | LESS_THAN | MORE_THAN | EQUAL_OR_LESS |
        EQUAL_OR_MORE | START | MID | END | APPROX ) #IMPLIED >
```

```
<!ATTLIST TIMEX3 quant ( EVERY | SOME ) #IMPLIED >
<!ATTLIST TIMEX3 temporalFunction ( true | false ) #IMPLIED >
<!ATTLIST TIMEX3 tid ID #REQUIRED >
<!ATTLIST TIMEX3 type ( DATE | DURATION | SET | TIME ) #REQUIRED >
<!ATTLIST TIMEX3 value CDATA #REQUIRED >
<!ATTLIST TIMEX3 valueFromFunction NMTOKEN #IMPLIED >

<!ELEMENT ALINK EMPTY >
<!ATTLIST ALINK eventInstanceID IDREF #REQUIRED >
<!ATTLIST ALINK lid ID #REQUIRED >
<!ATTLIST ALINK origin NMTOKEN #REQUIRED >
<!ATTLIST ALINK relType ( CONTINUES | CULMINATES |
        INITIATES | REINITIATES | TERMINATES ) #REQUIRED >
<!ATTLIST ALINK relatedToEventInstance IDREF #REQUIRED >
<!ATTLIST ALINK signalID IDREF #IMPLIED >

<!ELEMENT SLINK EMPTY >
<!ATTLIST SLINK eventInstanceID IDREF #REQUIRED >
<!ATTLIST SLINK lid ID #REQUIRED >
<!ATTLIST SLINK origin NMTOKEN #REQUIRED >
<!ATTLIST SLINK relType ( CONDITIONAL | COUNTER_FACTIVE |
        EVIDENTIAL | FACTIVE | MODAL | NEG_EVIDENTIAL |
        NEGATIVE ) #REQUIRED >
<!ATTLIST SLINK subordinatedEventInstance IDREF #REQUIRED >
<!ATTLIST SLINK signalID IDREF #IMPLIED >

<!ELEMENT TLINK EMPTY >
<!ATTLIST TLINK eventInstanceID IDREF #IMPLIED >
<!ATTLIST TLINK lid ID #REQUIRED >
<!ATTLIST TLINK origin NMTOKEN #REQUIRED >
<!ATTLIST TLINK relType ( AFTER | BEFORE | BEGINS | BEGUN_BY |
        DURING | ENDED_BY | ENDS | IAFTER | IBEFORE | IDENTITY |
        INCLUDES | IS_INCLUDED | OVERLAP_AFTER | OVERLAP_BEFORE |
        SIMULTANEOUS | UNKNOWN ) #REQUIRED >
<!ATTLIST TLINK relatedToEventInstance IDREF #IMPLIED >
<!ATTLIST TLINK relatedToTime IDREF #IMPLIED >
<!ATTLIST TLINK signalID IDREF #IMPLIED >
<!ATTLIST TLINK timeID IDREF #IMPLIED >
```

Code

Some of the code used to test the assertions made in this paper are included on a disk accompanying this paper. The code consists of the C++ implementation of Allen's algorithm, as well as a Prolog knowledge base for parsing TimeML, carrying out statistical analysis, and producing intermediate representation of temporal relations.