Strings for Temporal Annotation and Semantic Representation of Events

by

David Woods

A dissertation submitted

in fulfillment of the requirements

for the Degree of

Doctor of Philosophy

University of Dublin, Trinity College

March 2022

Declaration

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

I, the undersigned, agree to deposit this thesis in the University's open access institutional repository or allow the Library to do so on my behalf, subject to Irish Copyright Legislation and Trinity College Library conditions of use and acknowledgement.

I, the undersigned, consent to the examiner retaining a copy of the thesis beyond the examining period, should they so wish (EU GDPR May 2018).

David Woods

March 2022

Abstract

This work describes the use of strings as models for the representation of temporal data—that is, events and times, and their linear ordering and temporal interrelations—to form the basis of a framework for reasoning about that data and using it to aid in the creation or validation of semantic temporal annotation. Some of the relevant motivating literature is examined, in particular Allen (1983)'s interval algebra and relation set and the TimeML (TimeML Working Group, 2005) annotation schema. The finite-state temporality approach to semantics wherein the string framework originated is also detailed, and a breakdown is given of the work done to develop and flesh out the framework, including discussion on the various operations for manipulating and reasoning with the data. In particular, various flavours of a superposition operation allow for collation of the temporal information into compact, timeline or comic strip-like objects, which provide a useful visual reference or signpost for a document's temporal structure. A projection operation allows for the identification of temporal relations between arbitrary events and times which appear in the strings, and also for validating that data is not lost or corrupted from the original sources. Possible treatments of incomplete information are also described, leveraging the relation set associated with Freksa (1992)'s semi-intervals. Applications in annotation and scheduling are discussed, and a proof-of-concept online tool is presented which uses strings as a basis for creating, editing, and removing inconsistencies from documents marked up with TimeML.

Acknowledgements

Although I wrote this thesis alone, a number of people supported me along the way, and made it possible for me to reach the end.

I would like to thank my parents, Margaret and Graham, who never doubted I would do my best; my brother, Fergus, who has often been a much-needed reminder of normal life; Conor and Katie, who inspired me to start; Caitríona and Seán, who regularly found me comforting distractions; Adelais, who encouraged me to make it over the finish line and made sure I remembered life outside of writing; and all the members of DU Trampoline Club, who gave me a reason to stick around. I hope it was worth it!

My appreciation goes also to my supervisors: Tim, for being patient with me even when I was struggling, and Carl, who often managed to reassure me that I wasn't going down completely the wrong path. Thank you both for not giving up on me.

No thanks go to the COVID-19 global pandemic which disrupted my final year. Here's to survival, and here's to the memory of those that did not.

Related Publications

During the course of my studies, I was an author on three papers that were accepted for publication, listed below.

- *Towards Efficient String Processing of Annotated Events* (Woods, Fernando, and Vogel, 2017), describing the use of strings to model temporal data such as could be found in text annotated with ISO-TimeML. Presented at the 13th Joint ISO-ACL Workshop on Interoperable Semantic Annotation in Montpellier, France.

- *Improving String Processing for Temporal Relations* (Woods and Fernando, 2018), discussing refinements to the previously described string-based model, such as varied granularity. Presented at the 14th Joint ISO-ACL Workshop on Interoperable Semantic Annotation, colocated with COLING 2018 in Santa Fé, New Mexico, USA.

- *MSO with tests and reducts* (Fernando, Woods, and Vogel, 2019), discussing differing string granularities in the context of tests within Monadic Second Order logic. Presented at the 14th International Conference on Finite-State Methods and Natural Language Processing in Dresden, Germany.

Contents

Appendices 148

List of Figures

List of Tables

1 Introduction

The ability to reason about time and events is considered an essential aspect in several kinds of intelligent systems. For example, answering questions such as "Which of our students were in receipt of a grant last year?" or "How many conference papers were accepted last semester?" requires the use and interpretation of temporal information. Planning and scheduling systems rely on temporal reasoning in order to organise the data they process. Additionally, as human discourse and narrative often describes events out of chronological ordering, it's critical for natural language processing systems to be able to reason about sentences like "The stock market fell this morning after the controversy last night," so as to understand which event occurred first (the stock market falling, or the controversy), in order for it to have an accurate understanding of the scenario.

In order to perform reasoning about events and other time-related data in a useful way, it is first necessary to create a comprehensive framework for representing that temporal information in knowledge-based systems. Several approaches have been designed over the years, based on a number of conventions and formalisms. The event calculus (Kowalski and Sergot, 1986; Miller and Shanahan, 1999; Mueller, 2008), for example, represents events and their effects through the use of the predicates of a logical language. Allen (1983) used directed graphs to keep track of events and their inter-relations. T-BOX (Verhagen, 2005a) draws semantically-placed boxes to similarly display the relations between events and times, using as its basis the TLINKs of TimeML (TimeML Working Group, 2005), a semantic annotation schema and markup language which has become an international standard (ISO 24617-1:2012, 2012) for the annotation of text with explicit temporal data (Pustejovsky et al., 2010).

However, these frameworks do not offer a straightforward way to adequately and

intuitively visualise a document's temporal structure without sacrificing the ability to perform efficient reasoning over that temporal information, and vice versa. For example, the formality of the event calculus is arguably not intuitive to a layperson—which may impact their ability to use it for the purposes of annotation, the directed graph approach of Allen suffers from a non-semantic layout, TimeML has no native graphical interface, and while T-BOX has an attractive design, it is not intended for use beyond being a visualisation aid—see §2.2 for more details on these. This thesis will explore and describe the use of strings as models to represent temporal information—data concerning times and events—for use in computational systems which deal with knowledge-based reasoning in some way, while also maintaining both a compact visual appeal that is reminiscent of strips of film or panels of a comic.

Using strings—as finite sequences of information $\sigma_1 \cdots \sigma_n$, $n \in \mathbb{N}$, where each σ_i is a symbol representing some data—is not uncommon for visualising the relations between events, evoking notions of Gantt charts and timelines. For example, these strings give a general picture of the relative order of two events X and Y:

<div align="center">

XXX
    YYY
"X occurs before Y occurs"

XXXXXXXX
  YYYYY
"Y occurs during X"

</div>

or see Allen (1983, p. 835, Figure 2), reproduced in Figure 1 in §2, p. 8.

The overarching question this work seeks to answer is *How can strings be used to capture and represent temporal information in a precise, compact, and semantically meaningful way for reasoning and processing, whilst evoking the intuitive metaphor of timelines?* In order to tackle this, events and times must first be made representable as symbols in

a simple and logical manner, so that they may be used as elements in a string. It is important that the strings may be used for semantic reasoning; that is, logical consequences may be inferred—for instance, whether one set of temporal data entails another—allowing for useful deductions to be made. This work is also interested in ensuring that the framework be computationally tractable, so reducing algorithmic complexity and increasing data density are desirable. However, with this in mind, the intuitive presentational form of a timeline should be preserved where possible. The framework should be able to handle the complexities of real narratives and discourse to a satisfactory level, which may or may not contain incomplete or vague data.

Strings offer an excellent basis for this system which is to be created, as they are basic computational entities which are amenable to finite-state methods, such as decidably determining entailment of one string by another—see § 3.1.2—and reading them is intuitively similar to reading a timeline. The framework may have a number of applications in the field of intelligent systems, including among them annotation tooling, for which a proof-of-concept application is presented in § 5. This tool, named START (String Temporal Annotation and Relation Tool) is illustrative of what the string framework can provide to semantic annotation, and it is intended to be seen as a complementary tool for assisting with manual and semi-automatic annotation of temporal data in texts, alongside existant tools like T-BOX, which similarly strives for a semantic and intuitive presentation of temporal information.

The text proceeds as follows: § 2 introduces some of the core material which the current work uses as its foundation, including the interval-based framework for representing events/times and their inter-relations put forward by Allen (1983), as well as TimeML and the TimeBank corpus, and ISO-TimeML, which is TimeML's successor and an inter-

national standard for marking up texts with temporal annotations. §3 goes into detail on the finite-state approach to temporal semantics which licences the string framework, including how strings may be interpreted as finite models of Monadic Second-Order Logic, which leads to an equivalence with regular languages. The mechanics and operations for using strings as a representational tool for temporality are described, as well as how multiple strings can be composed together in order to increase data density and make explicit relations that were previously only implied. Possible applications for the use of strings are discussed, including in creating timeline structures from annotated texts and in scheduling problems which involve temporal constraint satisfaction. §4 discusses how documents marked up with TimeML—such as those in the TimeBank corpus—are used as a source for creating strings, and subsequently some methods for handling the often incomplete or vague information that is extracted from these narratives. §5 presents an online, web-based tool which can be used to demonstrate the utility of strings in a temporal annotation context. §6 explains how the framework can be validated by ensuring that information is not lost or corrupted when using the string representation, which can be proven by leveraging the projection operation.

2 Relevant Literature

In this chapter, the existing literature related to the major topics of the thesis are reviewed and analysed. These works are discussed in detail and some gaps are identified which may be filled through the use of a string-based representation framework for temporal entities and the relations between them.

2.1 Times and Events

The concept of time has fascinated researchers for millenia, and as such, a great deal of work exists on the topic. What follows focuses on the formal study of temporality in logic and language. First is presented the interval-based algebra which pervades throughout the rest of this work through TimeML's use of the same relations, as well as their appearance from superposition—see Table 7, p. 78.

2.1.1 Allen's Interval Algebra

Much of the present work draws its roots in James F. Allen's work *Maintaining Knowledge about Temporal Intervals* (1983), both directly and indirectly via the many systems which have built upon it in the years since it was first published. In this seminal paper, Allen described a framework for the use of temporal intervals as primitives to represent events and time periods, as opposed to points on the real line. There are four key criteria given as being of primary importance in guiding the design of this framework (1983, p. 833), all of which also feature in the system of string-based event representation described in the present work (see § 3.1):

1. The representation should allow for the fact that much temporal information is relative rather than precise.

2. Uncertainty of information should be allowed for, such as when the precise relation between two times is unknown (though constraints on the relation may exist).

3. The granularity of reasoning should be flexible—that is, capable of dealing with years and seconds in the same manner.

4. Reasoning should assume that states will persist unless there is some evidence to the contrary. In other words, *change* is the marker of progression of time.

Part of the motivation for using intervals rather than points is stated as follows:

> "There seems to be a strong intuition that, given an event, we can always 'turn up the magnification' and look at its structure. ... Since the only times we consider will be times of events, it appears that we can always decompose times into subparts. Thus the formal notion of a time point, which would not be decomposable, is not useful." (Allen, 1983, p. 834)

For example, taking just the event *Going-Home* from the sentence "I went home after work last night", it could be conceptually broken down into sub-events that make up parts of the whole: *Leaving-Work*, *Commuting*, and *Arriving-Home*. Each of these could be again further subdivided, repeatedly, as desired. However, if an event is given a specified time, such as "I went home at midnight", this seems to be a little trickier to deal with—'last night' is plainly referring to an interval of time during which the *Going-Home* event occurs, while 'midnight' appears to be an instantaneous moment of time. Nevertheless, the *Going-Home* event can still be subdivided in the same manner, which shows that even times that appear to intuitively points may be thought of as intervals as well.

Allen makes further argument for the use of intervals as primitive, disallowing zero-width time points, with an illuminating example involving a lightbulb being switched on: there must be an interval of time when the light was off, followed by an interval when it was on, but whether these intervals are open or closed presents another issue. If both intervals are open, then there is some point of time between the two when the light is neither on nor off; however, if both are closed, then there is some time point when the

6

light is both on and off. This fault can be resolved by having the intervals be open at one end and closed at the other, although Allen calls this an artificial solution which "emphasizes that a model of time based on points on the real line does not correspond to our intuitive notion of time" (1983, p. 834).

However, this argument that temporal intervals are counter-intuitive if they are open on one end and closed at the other has not been universally accepted—for example, the event calculus (Kowalski and Sergot, 1986; Miller and Shanahan, 1999; Mueller, 2008) describes a predicate $Initiates(e, f, t)$ which states that some event e occurs at a timepoint t, and the temporal proposition f is true after t. In the lightbulb example, this is equivalent to using intervals which are open on the lower end and closed at the higher, which can be interpreted as meaning the light is not on at the initial moment of switching it on, but is afterwards. Similarly, Fernando (2018) describes using an open left border and closed right border as a means for event representation.

Allen goes on to formally present and label the thirteen possible relations which may exist between two temporal intervals, which can be thought of as six invertible relations (before, meets, overlaps, during, starts, finishes) and one symmetric (equals). Figure 1, reproduced from Allen (1983, p. 835, Figure 2) shows these relations along with the symbols typically used to abbreviate them, and examples of some events represented graphically as strings of 'X' and 'Y' demonstrating the relations. These are often referred to as Allen's interval relations, or simply the *Allen relations*.

Relation	Symbol	Symbol for Inverse	Pictoral Example
X *before* Y	<	>	XXX YYY
X *equal* Y	=	=	XXX YYY
X *meets* Y	m	mi	XXXYYY
X *overlaps* Y	o	oi	XXX YYY
X *during* Y	d	di	XXX YYYYYY
X *starts* Y	s	si	XXX YYYYY
X *finishes* Y	f	fi	XXX YYYYY

Figure 1: Allen interval relations (Allen, 1983, p. 835, Figure 2).

These relations form a cornerstone both within this work (see, for instance, Table 7) and elsewhere—they are a fundamental part of the specification of ISO-TimeML (Pustejovsky et al., 2010) (see § 2.2.1), the international standard markup language for temporal annotation, where they are used, as one might expect[1], as the possible relation types between a pair of events tagged elsewhere in a document. Freksa (1992) also describes a larger set of *semi-interval* relations in terms of disjunctions of the Allen relations—see also § 3.1.3 and § 4.1.2.

The framework proposed by Allen uses a directed graph as its basis, in which the nodes represent intervals, and the arcs are labelled according to the relation (or, in the case of uncertainty, relations—see Table 1) between the intervals. It is assumed that complete information about the relations in the network is maintained—that is, there are no nodes without an edge between them, as the transitivities between the various relations are to be

---

[1]Albeit with slightly different labels and omitting *overlaps*—see Figure 7, p. 24.

computed as necessary—for instance, on the addition of a new interval into the network.

To illustrate, given a graph representing some pair of events X and Y such that X is *before* Y (Figure 2 (a)), a third node representing the event Z is added, with the additional information that Y occurs *before* Z (Figure 2 (b)). The relation between X and Z can then be calculated due to the transitivity rules which apply to the Allen relations—in this case, $X < Y$ and $Y < Z$ results in $X < Z$ (Figure 2 (c))—see also Table 1.
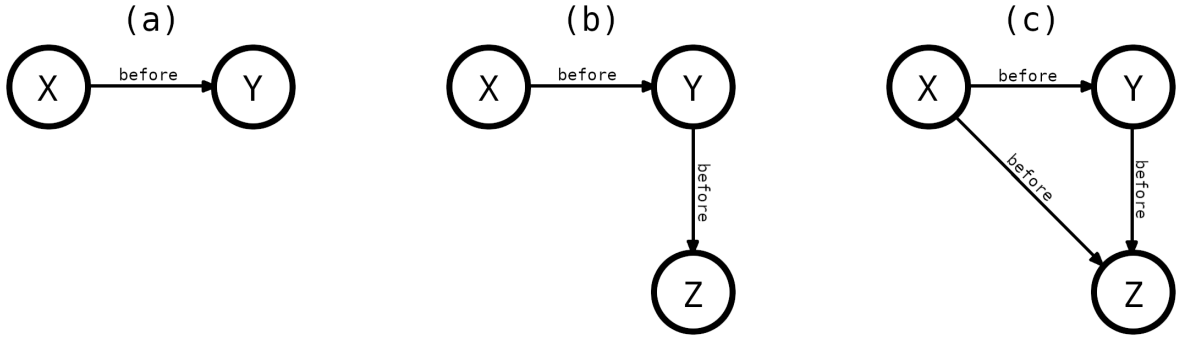
Figure 2: Simple graph network showing the computation of a transitivity.

For N nodes (intervals) there are $\frac{N^2-N}{2}$ edges if all new nodes are connected to all existant nodes when they are created. While in Figure 2 there is one label per edge, there may be as many as $13 \cdot \frac{N^2-N}{2}$ labels per edge if there is no knowledge about any relations. In Figure 3, it can be seen that a pair of single-label edges may result in a multiple-label edge by transitivity: the relations X o Y and Y o Z results in the disjunction of $X < Z$, X o Z, and X m Z. Without further data, there is no way to definitively constrain this disjunction to a single relation. If there is a single label for every edge in the graph, this is known as *temporal closure*—as discourse often naturally features incomplete or vague temporal information, calculating the temporal closure is, in general, a difficult goal to achieve, though it is desirable in order to create a representation that is both complete and consistent (Verhagen, 2005b).
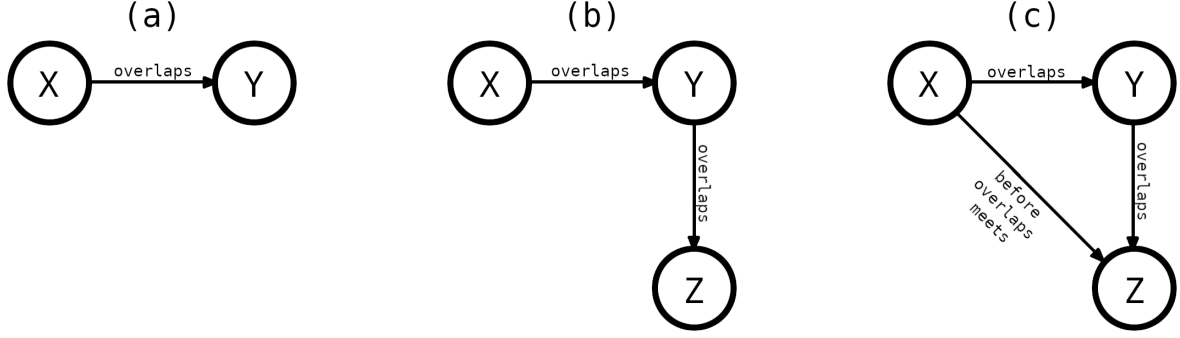
Figure 3: Computation of a transitivity with a multiple-label result.

Allen (1983, p. 836) gives a 12×12^2 transitivity table—a 6×6 fragment of which is reproduced below in Table 1—which provides the relation or relations which may exist between two intervals A and C, given the relations $\bullet$ between A and B, and $\bullet'$ between B and C. Not shown in Table 1 are relation pairs such as A before B and B after C—the disjunction of relations between A and C in this case contains all 13 possibilities (in Allen's table, labelled as 'no info'), indicating that there are no constraints on the relation between A and C. A smaller fragment of this table which shows these transitivities interpreted as superpositions[3] of strings which represent the various Allen relations is given in Woods et al. (2017, p. 130). One advantage of the string-based approach comes from the fact that a single string may be used to represent an arbitrary (finite) number of events, and all of the relations between them, at once—thus transitivities between, for example, four intervals can be represented as easily as three.

---

[2]The equals relation is omitted, since it is reflexive, symmetric, and transitive, meaning for any relation $\bullet$, $((A = B \wedge B \bullet C) \vee (A \bullet B \wedge B = C)) \implies A \bullet C$.

[3]See § 3.1.4 for details of superposition.

$B \bullet' C$ / $A \bullet B$	<	**d**	**o**	**m**	**s**	**f**
before (**<**)	<	< o m d s	<	<	<	< o m d s
during (**d**)	<	d	< o m d s	<	d	d
overlaps (**o**)	<	o d s	< o m	<	o	o d s
meets (**m**)	<	o d s	<	<	m	o d s
starts (**s**)	<	d	< o m	<	s	d
finishes (**f**)	<	d	o d s	m	d	f

Table 1: Fragment of Allen's transitivity table (1983, p. 836, Figure 4).

For the propagation of constraints upon the addition of new information to the network, Allen (1983, p. 835) also details an algorithm using the transitivity table to update all arcs in the network as necessary. While Allen does concede that there are some issues with this algorithm—specifically mentioning the space requirement, which is somewhat high at $O(N^2)$ space for N temporal intervals, and the fact that the algorithm doesn't guarantee consistency in larger than three-node networks[4], and Verhagen (2005b, p. 219) also notes an $O(N^3)$ time complexity—it does give an upper bound of $13 \times \frac{(N-1)(N-2)}{2}$ to the number of modifications that can be made to the network, regardless of the number of constraints added. Additionally, "the average amount of work for each addition is essentially linear (i.e., N additions take $O(N^2)$ time; one addition on average takes $O(N)$ time)" (Allen, 1983, p. 837). This is still quite high, given that it is not unusual for a given text, annotated documents of the TimeBank (Pustejovsky et al., 2003b, 2006) corpus for instance, to feature fifty intervals or more, and some are in the hundreds (Verhagen, 2005b, p. 213).

To tackle the space requirements while maintaining as much inferential power as possible, Allen (1983, p. 838) also describes a method for grouping clusters of intervals which—in regards to the relations between them—are fully computed, termed "reference

---

[4]See § 3.2, p. 77 for how this limit may be improved upon by using strings and superposition to automatically reject inconsistencies.

intervals". This is due to the fact that each interval I_i in the cluster $\{I_1, I_2, ..., I_n\}$ will reference one or more new intervals R_m (where m is an index on the number of reference intervals), and these reference intervals will be connected in the graph, which can be used to find relations between nodes which are not directly connected. Since these reference intervals may be treated as normal intervals, they may themselves be grouped into a cluster, and thus hierarchies of intervals may arise.

Allen's work was and remains highly influential in the field of temporal reasoning and annotation, thanks to an approach which is both intuitive and straight-forward, as well as easy to implement in various applications. One such proponent of interval relations has been TimeML, a markup language designed to "capture the richness of temporal and event related information in language" (Pustejovsky et al., 2005, p. 123)—see §2.2.1.

2.1.2 Tense and Aspect

Reichenbach's (1947) theory of verbal tense and aspect allows for the temporal placement of an event time in relation to a speech time and a reference time. The relative orderings of these three times gives rise to the categorisations of tense and aspect in English and other languages. This framework has been widely accepted and adopted, and has found empirical validation via TimeML (TimeML Working Group, 2005) and the TimeBank corpus (Pustejovsky et al., 2003b) in Derczynski and Gaizauskas (2013), which finds that using Freksa (1992)'s semi-interval relations leads to "tense appropriately constrain[ing] the types of temporal relations that can hold between pairs of events described by verbs".

Reichenbach (1947, p. 72) initially presents six tense/aspect combinations of English as arrangements of three temporal points, the 'point of the event', the 'point of reference', and the 'point of speech', which may coincide. These six are: the Simple Past, Past Perfect, Present, Present Perfect, Simple Future, and Future Perfect. However, it is

noted that the 'Simple Future' may have two possible interpretations in English, and the arrangement originally presented under that name—where the points of speech and reference are equal and precede the point of the event—is later (p. 77) renamed as the 'Posterior Present', while the second interpretation becomes the arrangement associated with the term 'Simple Future'—see Figure 4.

Tense	Example	E, R, S
Simple Past	"I bought the book"	$(E = R) < S$
Past Perfect	"I had bought the book"	$E < R < S$
Simple Present	"I buy the book"	$E = R = S$
Present Perfect	"I have bought the book"	$E < (R = S)$
Simple Future	"I will buy the book"	$S < (R = E)$
Future Perfect	"I will have bought the book"	$S < E < R$

Figure 4: Arrangements of the points E, R, and S, from Reichenbach (1947).

As can be seen from these arrangements, the tense of a sentence being in the Past, Present, or Future tense correlates to the ordering of the reference time R and speech time S: whether the reference time R appears before, equal to, or after the speech time S, respectively. The perfective aspect—called the 'Anterior' tenses in Reichenbach (1947, p. 77)—is then created by placing the event time E before R, where it is equal to R in the non-perfective aspect.

While Reichenbach (1947) (mostly) treats the time of event, speech and reference as temporal points, there is so far no issue with assuming that they may instead be treated as temporal intervals, à la Allen (1983) and TimeML (TimeML Working Group, 2005; Saurí et al., 2006). However, in order to treat the Progressive aspect which English may use, the so-called 'point of the event' must in fact be taken as an interval rather than an

instantaneous point, indicating the durative nature of the Progressive. If the event time was already considered to be an interval, then instead the set of relations must change so as to include the 'during' relation from Allen's set—see Figure 1, p. 8. Thus the Progressive aspect can be conceptualised by the reference time R occurring during the event time E, as in Figure 5.

Tense	Example	E, R, S
Past Progressive	"I was buying the book"	$R\ d\ E,\ R < S$
Present Progressive	"I am buying the book"	$R\ d\ E,\ R = S$
Future Progressive	"I will be buying the book"	$R\ d\ E,\ S < R$

Figure 5: Relations of E, R, and S for the Progressive aspect.

However, if all of E, R, and S are intervals, then Allen's interval relations apply to all pairings. Where before there was an equality relation between two of the points, Allen's equality will apply, but if one point preceded another, then in fact either of the 'before' or 'meets' relations may hold. Additionally, taking the Past Progressive as an example gives $R\ d\ E$ and $R < S$—assuming for the moment that R is before S rather than meeting it, then by the rules of transitivity for Allen relations, the relation between E and S may be any of 'before', 'meets', 'contains', 'finished by', or 'overlaps'. The fact that these disjunctions arise leads Derczynski and Gaizauskas (2013) to use Freksa (1992)'s set of 31 semi-interval relations—a superset of Allen's interval relations—in validating the Reichenbachian framework within the TimeBank (Pustejovsky et al., 2006) corpus. Semi-intervals arise by considering an interval in terms of its beginning and ending, but treating those as intervals in and of themselves, and in this way a set of relations are defined between the beginnings and endings of events. For instance, if the Freksa relation 'older' holds between some pair of events a and b, then the beginning of a, $\alpha(a)$, occurs

before the beginning of b, $\alpha(b)$, while the order in which the events end is left unspecified. Each Freksa relation may also be described in terms of a disjunction of Allen relations—in fact, the 'older' relation corresponds to the same disjunction of 'before', 'meets', 'contains', 'finished by', or 'overlaps' which appear between a verb's event time E and its speech time S in the Past Progressive. The Freksa relations are discussed in more detail in §4.1.2, but they are relevant here due to the disjunctions which come out of treating event, reference, and speech times as events, and leveraging shared reference times may lead to more of these disjunctions arriving when attempting to derive temporal relations between verbal events—see Figure 6.

By using what Reichenbach (1947, p. 74) refers to as the "permanence of the reference point", events in an utterance may be temporally ordered relative to each other. This states that, though events may be in different clauses, the fact that a speaker adjusts the use of tense and aspect for the events relative to each other shows that they must share a reference time. In the following example (1), the two events are in boldface.

(1) "I had **left** her when I **bought** the book."

Since the two events are in the same utterance, they have the same speech time, and by permanence of reference point, their reference times are also identical. Now, given that 'left' is in the Past Perfect, while 'bought' is in the Simple Past, the constraints from Figure 4 can be used to order the two events—using points rather than intervals for the sake of keeping the example simple—as follows in (2):

(2a) $$S_{leave} = S_{buy}, \ R_{leave} = R_{buy}$$
(2b) $$E_{leave} \ < \ R_{leave} \ < \ S_{leave}$$
(2c) $$E_{buy} = R_{buy} \ < \ S_{buy}$$
(2d) $$\therefore \ \ E_{leave} \ < \ E_{buy}$$

This gives that the event time of 'left' E_{leave} occurs before the event time of 'bought' E_{buy}, and thus for any pair of verbal events a and b which share a speech and reference time and the verb denoting a is in the Past Perfect while the verb denoting b is in the Simple Past, we can deduce that a precedes b, as temporal points. While this principle still holds when E, R, and S are treated as intervals instead of points, it does become a little more complex as the size of the possible relation set between a pair of intervals is over four times larger than the possible relation set for points, and as mentioned above, it is possible for disjunctions of relations to arise. Derczynski and Gaizauskas (2013) give a table with all of the combinations of tense {PAST, PRESENT, FUTURE} and aspect[5] {NONE, PROGRESSIVE, PERFECTIVE}—the aspect value of NONE denotes the Simple form of that tense—where each disjunction has been labelled with its associated Freksa relation, displayed in Figure 6.

e1 ↓ e2 →	PAST-NONE	PAST-PROG.	PAST-PERF.	PRESENT-NONE	PRESENT-PROG.
PAST-NONE	*all*	contemporary	succeeds	survived by	survived by
PAST-PROGRESSIVE	contemporary	*contemporary*	survives	older	all
PAST-PERFECTIVE	precedes	survived by	*all*	precedes	survived by
PRESENT-NONE	survives	younger	succeeds	*contemporary*	contemporary
PRESENT-PROGRESSIVE	survives	all	survives	contemporary	*contemporary*
PRESENT-PERFECTIVE	all	all	succeeds	survived by	survived by
FUTURE-NONE	succeeds	younger	after	succeeds	younger
FUTURE-PROGRESSIVE	survives	dies after birth	survives	younger	dies after birth
FUTURE-PERFECTIVE	after	younger	after	younger	younger

e1 ↓ e2 →	PRESENT-PERF.	FUTURE-NONE	FUTURE-PROG.	FUTURE-PERF.
PAST-NONE	all	precedes	survived by	before
PAST-PROGRESSIVE	all	older	born before death	older
PAST-PERFECTIVE	precedes	before	survived by	before
PRESENT-NONE	survives	precedes	older	older
PRESENT-PROGRESSIVE	survives	older	born before death	older
PRESENT-PERFECTIVE	*all*	before	survived by	before
FUTURE-NONE	after	*all*	contemporary	survived by
FUTURE-PROGRESSIVE	survives	contemporary	*contemporary*	survives
FUTURE-PERFECTIVE	after	survived by	survived by	*all*

Figure 6: Tense-aspect pairs and the Freksa (1992) relations they may suggest
(taken from Derczynski and Gaizauskas, 2013, p. 80, Table 5).

---

[5]The PERFECTIVE_PROGRESSIVE aspect is also possible—for example in "I had been walking"—however, it is omitted for simplicity's sake due to featuring in less than 1% of verb events in the TimeBank corpus (Derczynski and Gaizauskas, 2013, p. 77).

Care must be given, though, to note that the rule of a permanent reference time only applies to verbs within the same *temporal context* (Hornstein, 1990; Derczynski and Gaizauskas, 2013)—verbs appearing in reported speech, for instance, are in a separate context to the verbs which introduce it. For example, in (3), the events pointed to by 'put' and 'said' share their speech and reference times, and thus their temporal context is the same. However, 'going' is part of a separate statement, enclosed by the quotation marks, and having a speech time that is equal to the event time of 'said'.

(3) He **put** his head in and **said** "I'm **going** to the shop".

In general, if a pair of events that are to be temporally related are in separate temporal contexts "Reichenbach's framework may not directly apply, and the pair should not be further analysed" (Derczynski and Gaizauskas, 2013, p. 75). However, the temporal context can be difficult to determine automatically from a text, and the TimeML schema doesn't provide an explicit way to annotate it. One way to model a shared context between some pair of events is to look at the proximity of their appearance in the text, assuming events which are more textually distant than an adjacent sentence are unlikely to share the same context, and additionally checking if the events share the same tense— that is, the same speech and reference time. Using this model on the TimeBank 1.2 (Pustejovsky et al., 2006) corpus, Derczynski and Gaizauskas (2013, p. 80) validated the Reichenbachian framework of tense and aspect by extracting pairs of verbal events from the `<TLINK>` tags and using the tense and aspect of the verb pairs to derive one of the disjunctions of relations given in Figure 6. This derived disjunction contained as an element the relation that was actually marked up in the `<TLINK>` tag in 67.8% of cases[6].

---

[6]This excludes the prediction of the 'all' or 'unknown' relation, which is a disjunction of all possible relations. Including this relation is not useful, but when doing so the accuracy was found to be 91.9%.

It is noted that, taking into consideration that the temporal context model is somewhat crude, the amount of inaccuracy using this model (32.2%) is comparable to the inter-annotator disagreement for `<TLINK>` relation type labels (0.23) in the corpus: "The fact that temporal context is derived from models and not explicit gold-standard annotation is also likely a significant source of noise in agreement." (Derczynski and Gaizauskas, 2013, p. 80).

The next section takes a deeper look at the TimeBank corpus and the annotation schema with which it is marked up, TimeML.

2.2   Temporal Annotation

Ideally, humans who use artificially intelligent systems for question-answering, scheduling, or other applications which require some level of reasoning about temporal data won't ever have to consider the annotation which drives it—except, of course, in the case when annotation is itself the goal of using the system. Computer systems have no inherent understanding of natural language, and rely on the samples they have been provided with in order to give a facsimile of the communication abilities that humans have. The better the quality of those samples, and the more information embedded within them, the better the system can interpret human language, and in turn generate interpretable output.

The annotation of a document allows for information about the text to be expressed in such a way as to make the implicit explicit—revealing a layer of semantics which a human may be able to infer, but an artificial agent would not. There are many ways to mark up a text, and many concepts which may warrant marking up, but in terms of temporal data, the most prudent schema to follow is that of TimeML (TimeML Working Group, 2005) and its successor, which has become the ISO (International Organization

for Standardization) standard for temporal annotation: ISO-TimeML (ISO 24617-1:2012, 2012), a part of the ISO Semantic Annotation Framework—see Bunt (2020).

TimeML has found considerable success in terms of widespread adoption as a means of marking up text with temporal information, at least in some part thanks to the release of TimeBank (Pustejovsky et al., 2003b), a corpus of newswire articles[7] which were manually annotated using the TimeML schema. TimeBank frequently appears in discussion about TimeML since, in terms of quality, manual annotation is still seen as the 'gold standard' and superior to automatically generated markup, although machine-created annotation has seen much effort and improvement in recent years (Mani et al., 2006; UzZaman et al., 2013; Reimers et al., 2016).

The availability of appropriate tooling for the creation of human-driven annotation is crucial in ensuring that accurate and consistent results are produced. The first edition of the TimeBank corpus was annotated using the Alembic Workbench tool (Day et al., 1997), which was useful for the non-relational aspects, but impractical for the highly relational nature of the temporal data featured in TimeML annotations (Verhagen, 2005a). A number of other tools have been developed since then which aim to provide visual feedback and assistance to an annotator, including using directed graphs—as in Allen (1983)—and timeline-like depictions, of which the tooling described in §5 falls under the second category.

2.2.1 TimeML and TimeBank

The initial goal for creating the TimeML language came from a desire to improve applications—such as question-answering systems—by means of event recognition, and giving

---

[7]The original version of TimeBank contained 300 articles, while the latest version, TimeBank 1.2, contains 183.

each recognised event an explicit temporal location. This latter point is motivated due to the fact that a large proportion of temporal information in discourse rely on implicit or vague temporal expressions. This relates back to the first principle which Allen (1983) mentioned[8] as influencing the design of the interval algebra framework: that it is not generally intuitive to always think of or refer to time in terms of explicit or precise time points. Instead we tend to use relative expressions, such as the boldface text in the following utterances:

(4) "I didn't go to work **last Monday**."

(5) "I was sick **the week before**."

Speakers will rely on their listeners being able to use context to interpret what these temporal expressions refer to.

TimeML aimed, in part, to give implicit and relative temporal expressions an explicit anchoring, in order to assist with intelligent systems' temporal awareness. Pustejovsky et al. (2005, p. 125, (1a.)) wanted to enable question-answering systems to be able to answer questions such as

(6) "Is Schröder currently German chancellor?"

as capably as a human could after reading an appropriately relevant news article. A number of issues are raised that ought to be addressed within a system that can understand and answer questions similar to this one. Potentially problematic example queries can range from simple questions about the date of a specific event:

(7) "When did the USA first declare independence from the UK?"

to questions about non-unique events:

---

[8]See p. 5.

(8) "How long does it take to drive from Dublin to Cork?"

and questions where the system must perform some level of inference in order to derive the answer, possibly returning information that is not temporal in and of itself, but requires such data to find the correct solution:

(9) "Who was the last president of France?"

Pustejovsky et al. (2005, p. 132) further discusses the kinds of temporal information that might be needed, and how to go about representing it in a useful way. Two tasks are deemed essential: the ability to place events on some timeline, and the ability to determine the relative order of any pair of events. These tasks are termed event *anchoring* and *ordering*, respectively, and these also form a core motivation for the framework described in §3, which uses strings to simultaneously represent the anchoring and ordering of events.

Events in TimeML are "referred to by finite clauses, nonfinite clauses, nominalizations, event-referring nouns, adjectives, and even some kinds of adverbial clauses" (Pustejovsky et al., 2005, p. 133). Systems must also be aware of the possibility of negated and modal events, such as:

(10) "Ireland did **not** *make it* to the World Cup this year."

(11) "The exhibition **might** *create* new opportunities for the museum."

The events (italicised) in these sentences should not be treated as if they actually occurred. Additionally, care should be taken to distinguish separate events in the representation which are referred to together in the text (Pustejovsky et al., 2005, p. 134, (32a.)):

(12) "James *taught 3 times* on Tuesday."

This leads to a distinction between types of events and instances of events, where an instance may appear in the representation in place of a type, *cf.* (35), p. 45.

Times, on the other hand, generally take the form of adverbial or prepositional phrases in English, such as 'next week', 'yesterday', or '10th of August'. For TimeML, these expressions must be normalised in order to anchor events to times on a timeline—converted to some machine-readable form, possibly an integer or real number, depending on the context. It is also important to know the time of utterance (or document creation time, for a text) in order to correctly normalise expressions such as 'today' or 'last Monday', as these terms refer to a time which is relative to some other point, often the time of utterance or document creation time. Some expressions, such as 'recently' cannot be determinately linked to the timeline, due to their inherently vague nature, yet may still be ordered relative to other time points, and thus should still undergo normalisation. The last two kinds of time expressions mentioned in Pustejovsky et al. (2005) are durations and sets of times. Durations, such as 'five hours', may or may not be anchored to times or events (boldface text):

(13) "You'll have it for *a month* from **Monday**."

(14) "She submitted *two weeks* before **the deadline**."

Sets of times are generally used to place recurring events on the timeline:

(15) "He visited *every week*."

(16) "They took the meds *twice a day*."

In order for TimeML to represent the desired ordering and anchoring, a set of relations for times and events is required, and Allen's interval relations are selected as a strong basis for these, as reasoning over them is "well-understood" (Pustejovsky et al., 2005, p. 138). However, it is noted that not all of Allen's relations are equally well represented in texts of the English language. In particular, the *overlaps* relation is "difficult to find

instantiated in natural language text", and thus may be considered unnecessary. In fact, this relation and its inverse are omitted from the final set of temporal relations used in TimeML—see Figure 7. Nevertheless, since these relations may arise through transitivity and it is not immediately problematic to do so, they are reincluded in § 5.

The syntax of TimeML[9] uses the following types of tags to capture the various kinds of information: `<TIMEX3>` for marking up time expressions, `<EVENT>` for events and `<MAKEINSTANCE>` for event instances, `<SIGNAL>` for functional words (such as 'at', 'from', 'when', etc.), and three types of linking tags for relations between the other tags: `<SLINK>` captures subordinating relations involving evidentiality, modality, and factuality; `<ALINK>` captures aspectual links; and `<TLINK>` captures temporal relationships between events and times.

§ 4.1.1 and 5 focus primarily on this last tag type, as it is here that Allen's interval relations are represented, albeit under slightly different nomenclature, with some duplication, and omitting the overlapping relations as mentioned above—see Figure 7. Each of TimeML's tags take a number of attributes which help to flesh out the representation of the information, and for a `<TLINK>` these are: either a `timeID` or `eventInstanceID`, which refers to some temporal expression in the text; either a `relatedToTime` or `relatedToEventInstance`, which will refer to another temporal expression; and a `relType`, which will specify the relation from the first attribute to the second.

---

[9]For the full specification of TimeML (version 1.2.1), see `http://www.timeml.org` (TimeML Working Group, 2005), and annotation guidelines in Saurí et al. (2006).

TLINK	Allen
SIMULTANEOUS	equal $(=)$
IDENTITY	equal $(=)$
BEFORE	before $(<)$
AFTER	after $(>)$
IBEFORE	meets (**m**)
IAFTER	met by (**mi**)
INCLUDES	contains (**di**)
IS_INCLUDED	during (**d**)
DURING	during (**d**)
DURING_INV	contains (**di**)
BEGINS	starts (**s**)
BEGUN_BY	started by (**si**)
ENDS	finshes (**f**)
ENDED_BY	finished by (**fi**)

Figure 7: Possible values of a TLINK's relType attribute and their Allen relation counterparts.

The rationale behind distinguishing INCLUDES/IS_INCLUDED and DURING_INV/DURING is that the former should be used for cases where an event or time is included in another, like in:

(17) "He **arrived** there **last week**."

while the latter should be used specifically for states or events that persist throughout a duration (Pustejovsky et al., 2005, p. 158), such as in:

(18) "They **were Captain** for **two seasons**."

Below in Figure 8 is a simple example of a completed annotation—derived from the annotation guidelines (Saurí et al., 2006)—of the sentence:

(19) "He panicked on Wednesday."

```
He
<EVENT eid="e1" class="OCCURRENCE">
 panicked
</EVENT>
<SIGNAL sid="s1">
 on
</SIGNAL>
<TIMEX3 tid="t1" type="DATE">
 Wednesday
</TIMEX3>
<MAKEINSTANCE eiid="ei1" eventID="e1" pos="VERB"
tense="PAST" aspect="NONE" polarity="POS" />
<TLINK eventInstanceID="ei1" relatedToTime="t1"
signalID="s1" relType="IS_INCLUDED" />
```

Figure 8: Example TimeML annotation for "He panicked on Wednesday."

Since the initial publication of Pustejovsky et al. (2003a)[10], TimeML has received numerous improvements, updates, and tweaks. Although a version of the language has been adopted as an ISO standard (ISO 24617-1:2012) for temporal annotation (Pustejovsky et al., 2010), the most recent publically available version—see `http://www.timeml.org`— is TimeML 1.2.1 (TimeML Working Group, 2005; Saurí et al., 2006), and this is the version which the present work will focus on. The primary reason for this is due to the most recently available edition[11] of the TimeBank corpus—one of the largest available corpora for documents annotated with TimeML—being TimeBank 1.2 (Pustejovsky et al., 2006), wherein the annotation was updated in order to bring the corpus in line with the updated specification, which at the time was TimeML 1.2.1.

---

[10]In this work, the authors do refer to the TimeML language as being "developed in the context of a six-month workshop, TERQAS" during 2002. However, the URL link cited for *Annotation Guideline to TimeML 1.0* (`http://time2002.org`) no longer available points to an available resource as of March 2021. The TERQAS (Time and Event Recognition for Question Answering Systems) workshop was held at MITRE Bedford and Brandeis University (Pustejovsky et al., 2003b, p. 647).

[11]This refers to the English language corpus. There do exist versions of other TimeBank corpora in languages such as French (Bittar et al., 2011) and Hindi (Goel et al., 2020) which use the ISO-TimeML schema.

The TimeBank 1.2 corpus contains 183 texts, featuring news articles from a variety of sources, including broadcast news from ABC, CNN, PRI, and VOA, as well as articles from the Wall Street Journal. The counts for how frequently each tag appears in the corpus are given below in Figure 9, reproduced from Pustejovsky et al. (2006):

Tag	Count
EVENT	7,935
MAKEINSTANCE	7,940
TIMEX3	1,414
SIGNAL	688
ALINK	265
SLINK	2,932
TLINK	6,418
Total	27,592

Figure 9: Tag counts for TimeBank 1.2.

The TempEval-3 (UzZaman et al., 2013) shared task, which aimed to "advance research on temporal information processing", also produced a slightly updated version of the TimeBank 1.2 corpus, which is described as having been "cleaned". This included adjusting all of the files to have consistent formatting, to be XML and TimeML schema compatible, as well as adding some missing events, times, and relations (UzZaman et al., 2013, p. 2).

While this work will focus on TimeML 1.2.1, it is worth outlining some of the most notable alterations which were introduced by ISO-TimeML. One of the largest changes, from a structural point of view, is moving away from in-line annotation, whereby the markup tags are inserted directly into the body of the text. Instead, ISO-TimeML separates the annotation from the main text, using "stand-off" annotation, which increases the interoperability of annotation languages by conforming to the general practice of not modifying the text which is being annotated (Pustejovsky et al., 2010, p. 395). For example, a sentence like in (20) which features an event that appears partway through

will be marked up as something similar to (21) (omitting some attributes) using the older versions of TimeML:

(20) "He walked to the shop."

(21) He <EVENT eid="e1" ... >walked</EVENT> to the shop.

while ISO-TimeML would be something more like in (22) and (23), which are each in separate files:

(22) <seg type="token" xml:id="token1">He</seg>

　　　<seg type="token" xml:id="token2">walked</seg>

　　　<seg type="token" xml:id="token3">to</seg>

　　　<seg type="token" xml:id="token4">the</seg>

　　　<seg type="token" xml:id="token5">shop</seg>

(23) <EVENT xml:id="e1" target="#token2" ... />

Additionally, the <EVENT> tag is now explicitly used to denote an event instance, and as a result, the <MAKEINSTANCE> tag no longer exists. Some of its attributes are shifted to the <EVENT> tag, and <TLINK> tags in ISO-TimeML use eventID and relatedToEvent instead of eventInstanceID and relatedToEventInstance, respectively.

Finally, ISO-TimeML also takes a proposal from (Bunt and Pustejovsky, 2010) and provides a new type of link tag, <MLINK>, which has an "inherent relation type of MEASURE" (Pustejovsky et al., 2010, p. 396), and is used to better reflect the relationships between durative events and stretches of time—for instance, where an event may have been interrupted and resumed during some period, but is referred to as having taken the entire span of time. The new tag uses a measuring function (Bunt, 1985) to interpret the relation, so

that period of time that is measured is equal to the sum of all of the spans of time that make it up, whether contiguous or not. For example, in the sentence:

(24) "It rained for an hour."

it may not have been raining consistently for the full hour—there may have been a period of, say, twenty minutes when it abated—but it is valid to interpret it either as a full, non-stop hour of rain, or as a span of an hour during which it rained. This is what the `<MLINK>` is designed to treat, as in the example (25) below (where *P1H* refers to a period of 1 hour):

(25)
```
<EVENT xml:id="e1" pred="RAIN" />

<TIMEX3 xml:id="t2" type="DURATION" value="P1H" />

<MLINK eventID="e1" relatedToTime="t2" />
```

ISO-TimeML introduces a number of other changes over TimeML, such as the alteration and addition of several tag attributes, with the general aim of increasing interoperability, making it easier for other systems or software that might make use of the represented information. However, neither version of the language is intended to stand truly alone, but rather annotations are created for use in other applications. As such, there is no built-in way to visualise the depicted timeline of times and events, which is an intuitive way[12] to assist in understanding the anchoring and ordering of the temporal entities. As such—like the string-based framework described in § 3—other tools have been created which aim to aid in this arena, either simply for visualising a completed annotation, or as means of assisting an annotator in creating the markup.

---

[12]See § 3.2.1.

2.2.2 Tango and T-BOX

Manual annotation of temporal data in text is not a simple task, requiring a solid under-standing of the annotation schema, a strong ability to identify and keep track of multiple times and events, as well as being skilled in interpreting the often vague temporal data that exists in language (Verhagen, 2005b, pp. 213–214). Accordingly, a number of tools have been designed with the aim of assisting an annotator in making more correct decisions, ei-ther by helping to visualise the temporal structure of the document, or by automatically computing relations or marking inconsistencies as the annotator works. For example, Tango (Pustejovsky et al., 2003c) was developed with the aim of improving the annota-tion of documents with TimeML by allowing users to—quite literally—draw connections between times and events which were displayed graphically, as in Figure 10.
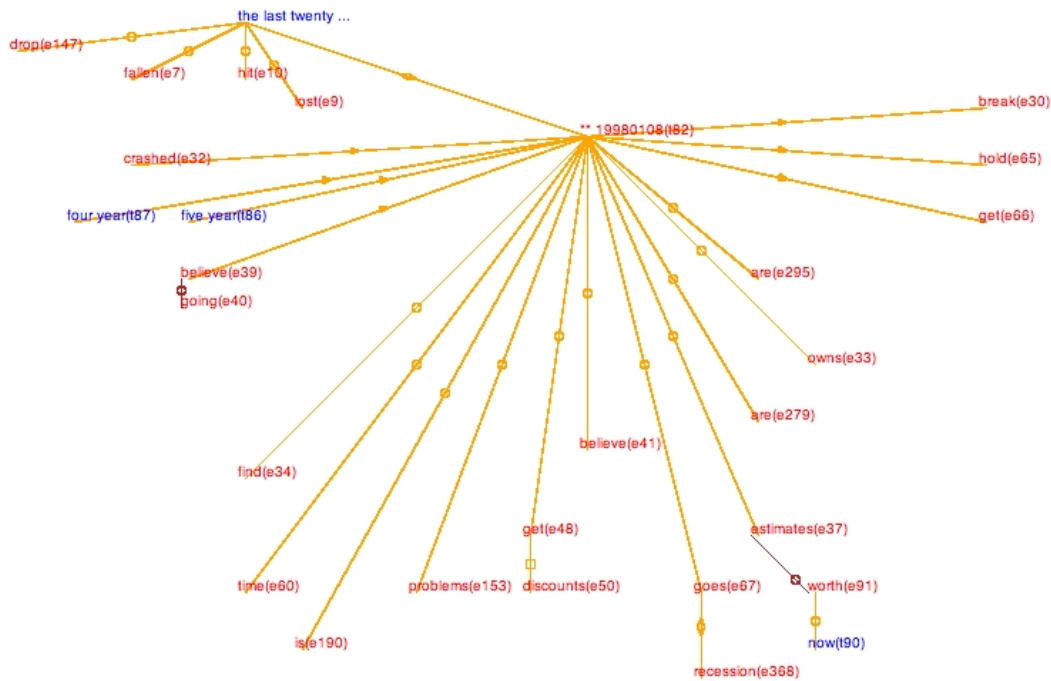
Figure 10: Tango's interface (taken from Verhagen, 2005a, p. 2, Figure 1).

The graph here works on the same principle as those described in § 2.1.1, in that the

nodes represent the events and times which are marked up in the TimeML document, while the arcs are labelled with the relation between the intervals. According to Verhagen (2005a, p. 2), using Tango improved the quality and reliability of the annotations being produced, with a higher number of links being found between the nodes, though he also notes that Tango's main flaw is that it does not allow a user to "quickly capture the temporal structure of the document" (Verhagen, 2005a, p. 2)—that is, interpreting the overall chronology of a text by means of a directed graph is not straightforward for a human user of the technology. Part of the problem, according to Verhagen (2005a), is that the graph labels—representing the `<TLINK>` tags of TimeML—quickly become difficult to read, especially when the document contains a large number of nodes. More troublesome, however, is the fact that there is no inherent semantics to the placement of the nodes; there is no graphical depiction of the temporal ordering, left-to-right or otherwise, that is enforced by the system in a meaningful way.

As a means of addressing some of these issues with the Tango tooling and its predecessor, the Alembic Workbench (Day et al., 1997), Verhagen (2005a) presents the T-BOX framework, intended to be used in a complementary fashion alongside the table-based and graph-based depictions in the existant tools. T-BOX is based around the core idea that "relative placement of two events or times is completely determined by the temporal relations between them" (p. 2). The image shown in Figure 11 represents the same data as in Figure 10, placing each event and time into boxes (called T-BOXes), and using arrows, stacking, and box inclusion to represent the various relations between the temporal expressions that may be derived from `<TLINK>` tags, under the assumption that events and times are intervalic, as described in § 2.1.1.

The rules governing placement of boxes are (Verhagen, 2005a, pp. 3–5):

- An event which occurs *before* another is placed to the left of it, with an arrow leading from the one to the other, or a line ending in a dot if the relation is *meets*.

- Simultaneous events are stacked, one box atop the other. Identical events are placed in the same box, rather than being treated the same as simultaneous ones.

- An event which *contains* or *includes* another gains an extended box with thinner walls, and the included event's box is placed inside this box.

- If an event *starts* or *finishes* another, it is placed inside that event's extended box, touching the left or right edge, respectively.

- Otherwise, none of these configurations may occur.
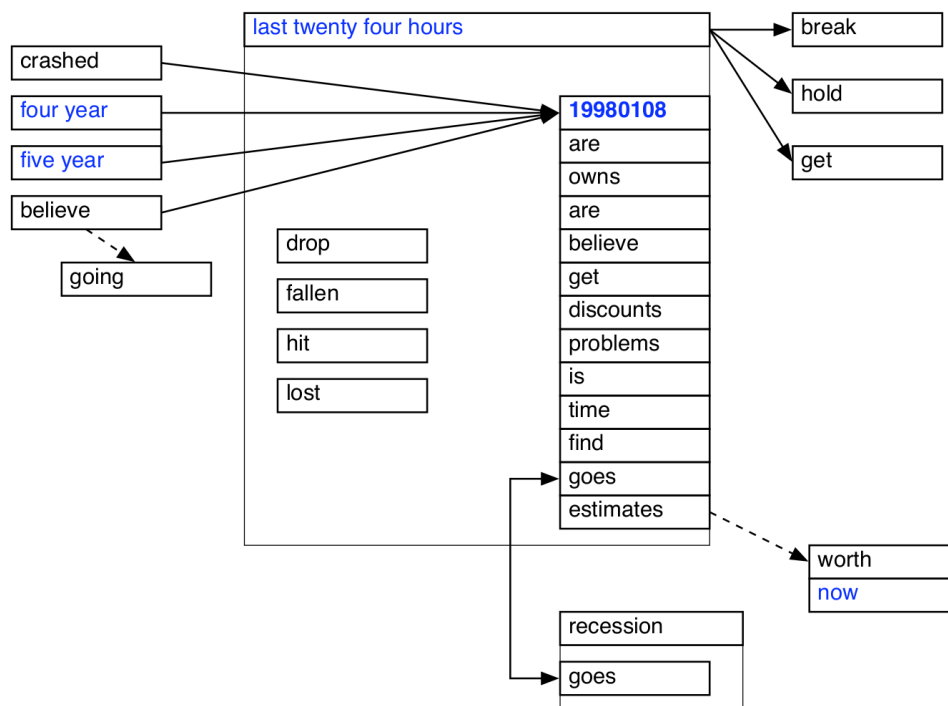
Figure 11: Figure 10's data now drawn using T-BOX (taken from Verhagen, 2005a, p. 3, Figure 2).

It is stressed that the vertical and horizontal positioning of boxes doesn't mean anything in and of itself, and thus despite a focus on semantically arranging and depicting the temporal relations in a TimeML annotation, T-BOX abandons the "timeline metaphor" (Verhagen, 2005a). However, using the string-based framework described in §3.1, many of the principles which guide the T-BOX architecture can be maintained whilst also preserving the ideology of timelines, being an intuitive way to conceptualise sequences of events and times—see §3.2.1.

Another point of interest with T-BOX is that it requires as input a TimeML document which is 'complete', and has already had temporal closure constraints applied to its `<TLINK>` tags, providing all of the inferrable relations ahead of time. It then reduces this to a minimal graph (Verhagen, 2005a, p. 6). This is as opposed to the more typical scenario of trying to compute the temporal closure of a document using a constraint propogation algorithm—such as Allen (1983)'s in §2.1.1, p. 11—or, as in the present work, using the superposition of strings to calculate relations at the same time as depicting them.

One further aspect of the T-BOX framework worth noting is that it can be used to possibly detect some inconsistencies in input data if there is some part of the data which cannot be drawn using the rules described above. For example, if the `<TLINK>` tags give that some event X is *before* some other event Y, that Y is *before* a third event Z, and also that Z is *before* X—an impossibility due to the circular transitivity—the inconsistency should be discovered when attempting to draw X to the left of Y, which is to the left of Z, which should then somehow be drawn to the left of X. However, while 'non-drawability' implies an inconsistency, drawability does not imply consistency (Verhagen, 2005a, p. 12), since similarly to the constraint propogation algorithm in Allen (1983), inconsistencies

may appear in the graph which appear consistent when looking at three intervals, but become problematic in a larger context. This issue can be circumvented using a string which may represent far more than three intervals and their relations at once, and strings which represent information inconsistent with the knowledge base are ejected through superposition—see (95), p. 71—and thus, 'non-superposability' can be seen as similar to the 'non-drawability' of T-BOX.

The next section discusses some of the existing approaches to semantic representation of times and events aside from the semantics associated with the annotation schemas of TimeML and ISO-TimeML, in particular the structures of Discourse Representation Theory.

2.3   Temporal Semantics of DRT

In what have become robust and well-known works since their publication, Kamp (1981, 1988); Kamp and Reyle (1993) presented Discourse Representation Theory (DRT) as a formal framework for semantically representing information derived from discourse—that is, coherent series of sentences or propositions, which may appear in speech or text. The development of DRT aimed to allow model-theoretic semantic representations that go beyond single sentences, which was the standard approach in First-Order Logic (FOL), and to treat phenomena such as anaphora, where an entity named earlier in a discourse may be referred to again later without naming it again. The classic example sentences (Kamp, 1988) involve an indefinite, 'a donkey', appearing as an antecedent, and acting

as a quantifier which binds the pronoun 'it' in the second sentence, as in (26a).

(26a)   "John owns a donkey. He feeds it."

(26b) $\exists x \ (donkey(x) \land own(John, x) \land feed(John, x))$

The sentence in (27a) is problematic in Montague-style approaches, which assume that any indefinite in a sentence should introduce an existential quantifier, but this comes into conflict with the scope of the universal quantifier introduced by 'Every' (Kamp, 1988, p, 91). In (27b), the expected existential has acquired a universal force to capture the natural interpretation of (27a) as claiming that every farmer feeds every donkey they own.

(27a)   "Every farmer who owns a donkey feeds it."

(27b) $\forall x \forall y \ ((farmer(x) \land donkey(y) \land own(x, y)) \implies feed(x, y))$

DRT solves these issues with the notion of discourse referents, which are the set of entities under discussion in a given discourse, corresponding to the individual variables of FOL (Bird et al., 2009, p. 397). Under DRT, indefinites do not introduce existential quantifiers, instead introducing new discourse referents, which are added to a mental representation known as a Discourse Representation Structure (DRS) (Geurts et al., 2020). DRSs contain separately both the discourse referents and the DRS conditions, which describe what is known about the referents. Multiple DRSs can be concatenated where it is appropriate to do so, with the result being a merged DRS containing the union of the discourse referents and DRS conditions from each of in the inputs to the concatenation (Bird et al., 2009, p. 399).

For the example in (26a), the first sentence is processed and the two entities create

two discourse referents, x and y, which appear at the top of the DRS in (28), with the DRS conditions over these referents appearing in the lower box of the DRS.

(28)

x y
John(x)
donkey(y)
owns(x, y)

As the second sentence of (26a) is processed, the DRS is augmented with two further discourse referents, u and v, corresponding with the two pronouns 'He' and 'it'. Resolving the anaphora[13] adds two new conditions, identifying the two pronoun referents with the two existing noun referents in (28). Finally the condition that u feeds v is added, giving (29).

(29)

x y u v
John(x)
donkey(y)
owns(x, y)
$u = x$
$v = y$
feeds(u, v)

There is a direct translation from a DRS to a formula of FOL, where the discourse referents which appear at the outermost level are interpreted as existential quantifiers, and the DRS conditions are interpreted as being conjoined. The translated formula for (29) is in (30),

---

[13]The component which performs the anaphora resolution is assumed separate to DRT, which merely sets constraints on which discourse referents are available as candidates for selection (Bird et al., 2009, p. 399).

which has the same interpretation as (26b) despite its differences.

(30) $\exists x \exists y \exists u \exists v \; (john(x) \wedge donkey(y) \wedge owns(x,y) \wedge u = x \wedge v = y \wedge feeds(u,v))$

Universal quantification is usually handled by embedding two DRSs connected by an implication within another DRS (Kamp, 1988, p. 98), or alternatively by embedding a negated DRS within another embedded negated DRS[14]. Below, (31) represents (27a).

(31)

x
farmer(x)

y		u
donkey(y)	$\implies$	$u = y$
owns(x,y)		feeds(x,y)

DRT is capable of representing events and times as temporal entities which introduce discourse referents and may appear in DRS conditions. For example, including a special 'indexical' discourse refererent *now* for the time of the utterance, as in Kamp (1988, p. 104) and Abzianidze et al. (2017), (32b) represents the sentence in (32a).

(32a)   "The doctor cured the patient."

(32b)

$x \quad y \quad e \quad now$
doctor(x)
patient(y)
cure(e,x,y)
$e \prec now$

The condition $e \prec now$ denotes that the event e precedes the time of the utterance, indi-

---

[14]Due to the equivalence $\forall x \; (P(x)) \iff \neg \exists x \; (\neg P(x))$, stating that if some property $P(x)$ holds for all values of x, then there is no value for x where $P(x)$ does not hold.

cating the past tense of the verbal event 'cure'. Other temporal relations available in DRS conditions usually include $=$ equality, $\subset$ during, and $\bigcirc$ overlaps, though other relations occasionally are used, and in principle any binary relation could be used, including those in Allen (1983)'s set. It is possible in general to extract these relations and the events they refer to in order to build temporal-focused representations, such as the strings described in §3.

The DRS in (32b) represents a Davidsonian approach to event semantics, where an event variable e was added as an argument to the predicate corresponding to the verb (Davidson, 1967; Kamp and Reyle, 1993), and DRSs can also feature discourse referents representing statives as well as eventives (Kamp, 1988, p. 103). However, by introducing an event's semantic roles as conditions to the DRS, a neo-Davidsonian style (Dowty, 1989) can be implemented instead, reducing the verbal predicate's arguments to just the event variable e. This is an approach which Bos and Abzianidze (2019); Bunt (2020) point out is the *de facto* standard approach taken by most (if not all) semantically annotated corpora. Indeed, the Parallel Meaning Bank[15] (PMB) corpus (Abzianidze et al., 2017)—which aims to "provide fine-grained meaning representations for words, sentences and texts" and contains over 8000 DRSs from English sentences—uses this approach in their DRS representations. The DRS in (33) below represents the sentence in (32a) again, this time including such semantic roles as might appear in the PMB, which uses an inventory

---

[15]Available at `https://pmb.let.rug.nl/`.

of roles from the VerbNet resource (Schuler, 2005).

(33)

$x \quad y \quad e \quad now$
$\text{doctor}(x)$
$\text{patient}(y)$
$\text{cure}(e)$
$\text{Agent}(x)$
$\text{Patient}(y)$
$e \prec now$

Automatic parsing of discourse text to DRSs is a non-trivial, multi-faceted task, involving tokenisation, part-of-speech recognition, and syntactic parsing before DRSs can even be constructed. A wide-coverage parser called Boxer was released by Bos (2008), which claimed 95% converage for semantic analysis of newswire texts, although significantly for the present work, it was noted in that release that performance for temporal information was not as strong as its other areas (Bos, 2008, pp. 283–285). Boxer takes as input a Combinatory Categorial Grammar (CCG) (Steedman, 2000; Steedman and Baldridge, 2011) which has itself been parsed following named-entity recognition, and part-of-speech tagging prior to that, using the C&C tools (Curran et al., 2007), and outputs one or more DRSs as formally interpretable semantic representations (Bos, 2008, p. 285). An updated version of Boxer (van Noord et al., 2018) is implemented as part of the Parallel Meaning Bank toolchain (Abzianidze et al., 2017)—although, this version had unfortunately not been made available for general use at the time of writing. However, the recent shared task created by (Abzianidze et al., 2019) shows that there is an interest in developing systems which can perform automatic DRS parsing. The state-of-the-art was improved considerably in this task, with the winning system of Liu et al. (2019) achieving an F1 score of 84.8%, an improvement from the baseline score of 54.3%.

This chapter has attempted to give a general overview of some of the literature which informs the current work. In particular, the interval algebra put forward in Allen (1983), which finds application in the temporal relations of the TimeML (TimeML Working Group, 2005) and its successor ISO-TimeML (Pustejovsky et al., 2010), which has become the international standard for semantic annotation of temporal information, and also the empirical validation of Reichenbach (1947)'s framework of tense and aspect in Derczynski and Gaizauskas (2013), which also brings out Freksa (1992)'s semi-interval relations, and finally the semantic structures of the formalism of Discourse Representation Theory (Kamp, 1981) which aims to reflect the context of an utterance in its model-theoretic interpretation. The next chapter goes into some depth for the approach to temporal semantics known as finite-state temporality, and details the string-based framework it licences which forms the backbone of the work.

3   Finite-State Temporality

Following the intuition that sequences of (possibly overlapping) events and time periods may be conceptualised in a manner akin to a strip of film, or a timeline, finite-state techniques may be applied to temporal semantics in an approach known as *finite-state temporality* (Fernando and Nairn, 2005). In this chapter, strings are demonstrated as a tool of choice in modelling sequences of times and events for use within this approach, justified by their interpretation as finite models of Monadic Second-Order Logic, which leads to an equivalence with regular languages (see § 3.1.2). The advantage of this is that strings can be accepted and parsed by finite-state automata (FSA), which are a well-known formalism and have found significant usage across many fields, including disciplines of Mathematics, Linguistics, and Computer Science (see, for example, Buchner and Funke (1993); Veanes et al. (2012)), as well as in modern software development (for instance, Khourshid (2015) provides a tool for using FSA in the creation of online web applications). Such widespread application of FSA is due to their flexibility and efficiency as a technology, with benefits such as deterministic recognition being linear according to the length of the input, the associated closure properties of regular languages[16], and the ability to compose several automata (Wintner, 2007).

The mechanics of these temporal strings are shown in detail, along with an explanation on the methods of their creation, and a discussion on the granularity of temporal information that should be included within a string—that is, what level of detail should be considered when representing events in this manner. A number of operations are subsequently described for working with these strings, in particular *superposition* for the composition of multiple strings and combining the data found within them. The avail-

---

[16]Seen in the superposition of languages on p. 59.

ability of these manipulations will prove useful when reasoning about event relations, and also in maximising data density, as will be seen in § 4.1 and § 4.3.

A description follows of how strings may be applied in the creation of timelines, representations which contain the temporal information—events, times, and the relations between them—as extracted from an annotated piece of text, which has uses in the areas of, for instance, automatic summary-generation, fact-checking, and question-answering systems. Additionally, the techniques which are laid out in § 3.1.4 may be augmented with additional constraints so that strings can be used to model scheduling restrictions—for instance, that in a given system some event a must occur before some other event b, but may not be occurring while c is occurring—and the superposition of these amounts to constraint satisfaction. A variation of the well-known Zebra Puzzle (also referred to as 'Einstein's Riddle', as it is occasionally attributed to Albert Einstein—see Stangroom (2009, p. 10)) which uses temporal properties in place of spatial constraints is provided, exemplifying this particular usage of strings, as well as a string-based treatment of the train scheduling problem from Durand and Schwer (2008b).

3.1   Strings for Times and Events

A string is a basic computational entity, defined as a finite sequence of symbols selected from some finite alphabet. They are amenable to manipulation using finite-state methods, something lacking in the infinite models of predicate logic, thanks to fixing finite sets of symbols to serve as the alphabets which make up the strings.

Strings as described and used throughout this work are used to represent sequences of events and time periods such that the linear order and inter-relations of these events and times are clearly apparent, while unnecessary repetition of information is avoided. For

example, where $js =$ "John sleeps", $fa =$ "The fire alarm sounds", and $lt =$ "Last Tuesday", the sentence "John slept through the fire alarm last Tuesday" might be represented by the string $\boxed{\ |\ lt\ |\ js, lt\ |\ fa, js, lt\ |\ js, lt\ |\ lt\ |\ }$[17] (a detailed explanation follows in § 3.1.1).

These strings model the concept of inertial worlds, wherein a state will persist unless and until it is altered. The intuition for viewing inertia as a default state seems to go at least as far back as Aristotle ("But neither does time exist without change" in *Physics IV*), and is known by the term *commonsense law of inertia* (Shanahan, 1997, p. 19). This notion is also present in the Event Calculus, which represents the effects of actions on fluents in order to reason about change (Kowalski and Sergot, 1986; Miller and Shanahan, 1999; Mueller, 2008), and in the Fluent Calculus (Thielscher, 1999), which asserts that a state will be unaltered after an event, except for just those conditions which the event changes. Building this inertial world view into strings allows for certain flexibilities, as real duration does not need to be accounted for[18] when, for example, superposing strings in order to determine the relations between the events they mention (see § 3.1.4, p. 58; § 4.3.1). Fernando (2018, p. 44) also directly connects the notion of "No change unless forced" with strings, using it to motivate the concept of actions creating change. Additionally, the built-in inertiality—coupled with the fact that strings are explicit as to whether a particular *fluent* holds or does not hold at any particular moment in time— means that, for string-based representations of events and times, the classic issue of the frame problem[19] is avoided (see McCarthy and Hayes (1969, pp. 30-31)).

---

[17]It is worth noting that, although tense and aspect are often abstracted away from the string models, this is not a necessity, and speech and reference times may be represented in the same way as event times (Fernando, 2016a; Derczynski and Gaizauskas, 2013; Reichenbach, 1947).

[18]Real-time durations may still be represented using strings, but the depiction will not (necessarily) align with the assumption that one string should be longer than another if it features events that have a longer duration.

[19]The frame problem is an issue that can arise in first-order logic representations of the world, whereby specifying the conditions which change as the result of an action is not sufficient to entail that no other conditions have changed.

A fluent is a condition which may change over time—thus a predicate such as $Sleeps(john)$ ("John sleeps") becomes the fluent $Sleeps(john, t)$, where t is the time at which John is sleeping. Following the convention set out and used by McCarthy and Hayes (1969), Van Lambalgen and Hamm (2008), Fernando (2016b) (among others), a fluent may be understood here as naming a temporal proposition—some event, time period, or state which may change (which hereafter will also be referred to as an event, as in Pustejovsky et al. (2005)). Sets of fluents will be encoded as symbols so that any number of them may hold at once, and these symbols will make up the alphabet from which strings are created.

In most cases, simple identifiers suffice for the purposes of labelling fluents as they appear in a string—so, for example, $Sleeps(john, t)$ may be labelled as `"js"`, where the time t being represented by the fluent's position in the string, explained in depth in the following section. This follows TimeML's standard (TimeML Working Group, 2005; Pustejovsky et al., 2010) of using identification labels such as `"e1"` or `"t2"` for events and times. However, in some instances, particularly when drawing inferences (see §4.3) using semantic roles or lexical semantics, it will be useful to use a fuller labelling system, and so a string $\boxed{\ \boxed{js}\ }$ would be rendered instead as $\boxed{\ \boxed{sleeps(john)}\ }$.

3.1.1 Creating Strings

In order to create a string, first it is necessary to fix a finite set $\mathcal{V}$ of symbols which represent the times and events under discussion, such that each $v \in \mathcal{V}$ will be understood as naming a *fluent* as a unary predicate which holds at a particular time. A string $s = \sigma_1 \sigma_2 \cdots \sigma_n$ of subsets σ_i of $\mathcal{V}$ is interpreted as a finite model of n discrete, contiguous moments of time, with $i \in \{1, 2, \ldots, n\}$.

The set $\mathcal{V}$ will be known as a *vocabulary*, and the powerset of $\mathcal{V}$ will serve as a finite

alphabet $\Sigma = 2^{\mathcal{V}}$ of a string $s \in \Sigma^*$. Accordingly, at each position i in the string s, the *component σ_i* will be a (possibly empty) set of the fluents which hold at that position.

The chronology of a string is read from left to right, and thus, each component of the string depicts one of the n moments similar to a snapshot, or a frame of a film reel, and specifies the set of exactly those fluents which hold simultaneously at the ith moment[20]. A string does not (necessarily) give any indication of real-time duration[21], due to the fact that it models an inertial world, and thus if a symbol occurs in several string components, this is not implicative of the fluent associated with that symbol occurring multiple times, nor of the fluent's real-time duration being necessarily different to another fluent whose associated symbol only occurs in a single string component (see also § 3.1.4, p. 57). A fluent $a \in \sigma_i$ is understood to be occurring before another fluent $b \in \sigma_j$ if $i < j$ and $b \notin \sigma_i$; if $a \in \sigma_i$ and $b \in \sigma_i$, then a and b are understood as occurring at the same time.

For convenience of notation, boxes $\boxed{\cdot}$ are used instead of curly braces $\{\cdot\}$ to denote the sets which make up each component σ_i of a string $s = \sigma_1 \sigma_2 \cdots \sigma_n$ (as in Fernando (2004, 2015, 2016b); Woods et al. (2017))—for example, the string $\{\}\{a\}\{a,b\}\{b\}\{\}$ is written as $\boxed{}\boxed{a}\boxed{a,b}\boxed{b}\boxed{}$. This lends to the intuition that the strings may be read as strips of film, or as panels of a comic, with the same narrative-style layout as a timeline[22]. An empty box $\boxed{}$ is drawn for the empty set $\varnothing$: this is a string of length 1, a moment of time during which no fluent $v \in \mathcal{V}$ holds. This should not be confused with the empty string ϵ, which has length 0, and contains no temporal information.

---

[20]Conversely, the set $\mathcal{V} - \sigma_i$ specifies exactly those fluents which *do not* hold at the ith moment. This relates to the notion of logical circumscription (McCarthy, 1980), wherein if a formula is not known to be true, then it must be false.

[21]Although the durations may still be used in certain applications—see § 3.2.2, p. 89.

[22]It is worth pointing out here that, as each box is a set, the order of fluents appearing in it is immaterial— for instance, $\boxed{a,b}$ is exactly equivalent to $\boxed{b,a}$, where in both cases, a is co-occurring with b. This contrasts with $\boxed{a}\boxed{b}$ vs $\boxed{b}\boxed{a}$, where in the first case a is occurring before b, and in the second, the converse is true.

It will be said that for any event a occurring in a string $s = \sigma_1\sigma_2\cdots\sigma_n$, a may not *judder*—that is, if a appears in multiple positions of the string, then those positions are contiguous; there are no gaps between appearances of a in s:

(34)
$$a \in \sigma_i \wedge a \in \sigma_j \wedge i < j \implies \forall k \in [i \mathrel{..} j]\ (a \in \sigma_k)$$

If some string, such as in (35a), features judder in this way, it is said to be invalid, similarly to Pustejovsky et al. (2005, p. 134) distinguishing separate instances of an event which are referred to together—see (12), p. 21—and only allowing instances to appear in a string. Judder can, if necessary, be treated by subindexing an event—for example, a juddering event a becomes separate sub-events $\{a_1, a_2, a_3\}$, as in (35b). Alternatively, similarly to Bunt and Pustejovsky (2010) and ISO-TimeML's treatment of non-contiguous events over a duration[23], it may be possible to interpret judder as one whole event, which includes the pauses, as in (35c), which would then be *block compressed*—see (61), p. 58.

(35a) $\boxed{a}\ \boxed{a}\ \boxed{a}$ $-$ invalid

(35b) $\boxed{a_1}\ \boxed{a_2}\ \boxed{a_3}$ $-$ valid

(35c) $\boxed{a\,|\,a\,|\,a\,|\,a\,|\,a}$ $-$ valid

Sets of strings are *languages*, which allow for grouping of strings based on some property—such as their source—and which introduces a method for handling such cases of ambiguity and non-determinism as will be shown in more detail in §4.3.1 (see also (67), p. 60). For

---

[23]See §2.2.1, p. 27.

example,

(36) $$L = \{\boxed{a\;|\;b\;|\;c}\,,\boxed{a\;|\;c\;|\;b}\}$$

where the language L contains two strings which represent different—albeit related—sequences of events.

Note that a language containing only a single string may be conflated with its sole member, and vice versa. For instance:

(37) $$\boxed{a\;|\;b\;|\;c} \approx \{\boxed{a\;|\;b\;|\;c}\}$$

This is a useful admittance, particularly in regards to superposition and other string operations (see §3.1.4), when it may be necessary to, for example, superpose a string and a language, in which case the string is conflated with a language containing just that string. This allows for superposition to be extended beyond just two strings to any arbitrary number: $s \mathbin{\&_{x}} s' \mathbin{\&_{x}} s'' \mathbin{\&_{x}} \ldots$ (see p. 71).

The next section details how strings are expressions of finite models of Monadic Second-Order Logic, and how they are thus manipulable by finite-state automata.

3.1.2 Strings as MSO Models

Strings of symbols representing events as described in §3.1.1 may be interpreted as finite models of Monadic Second-Order Logic (MSO)[24]. MSO is a fragment of Second-Order Logic that restricts quantification so as to be permitted solely for unary predicates, which is equivalent to quantification over sets—this is due to the fact that a unary predicate may

---

[24]See Libkin (2004, ch. 7) for an in-depth introduction to the facets of MSO.

be effectively described by the set of terms for which that predicate is true. That is, in a given model of MSO, if there exists some property P, then $\llbracket P \rrbracket$ is the set of individuals for which P holds—known as the *interpretation* of P relative to the given model—such that:

(38) $$P(x) \Longleftrightarrow x \in \llbracket P \rrbracket$$

This may be construed for temporal representation by considering each predicate of a model as describing an event, and the terms which make that predicate true are the moments (relative to the model) during which the associated event is occurring.

This is most clearly illustrated by means of an example. The string below in (39) will serve for this purpose, with the positional indices shown underneath the string position with which it corresponds:

(39)

$$\boxed{\;\big|\;a\;\big|\;a,b\;\big|\;a\;\big|\;}$$
$$1\;\;2\quad 3\quad\;\;4\;5$$

This is a string of length 5, which contains two events, a and b, where b occurs *during a* (see table 7). The linear ordering of a and b can be identified by the string positions in which each occurs (Fernando, 2016a, 2018):

(40) $$\llbracket P_a \rrbracket = \{2, 3, 4\} \text{ and } \llbracket P_b \rrbracket = \{3\}$$

where P_a and P_b are unary predicates, and their interpretations are subsets of $\{1, 2, 3, 4, 5\}$, which is the set of all string positions for the string in (39).

Generally, for any string $s = \sigma_1\sigma_2 \cdots \sigma_n$ with length $n \geq 0 \in \mathbb{N}$, the set $[n]$ of string

47

positions[25] is defined as:

$$(41) \qquad\qquad [n] := \{1, 2, \ldots, n\}$$

Since s is restricted to being a finitely bounded string, $[n]$ is also finite, and thus the MSO model described by s must also be finite.

The vocabulary $\mathcal{V}$ is the set of fluents which appear in s, and for each $v \in \mathcal{V}$, the set of positions during which v (as a fluent) occurs $[\![P_v]\!]$ is a subset of $[n]$, and if there exists some $x \in [n]$ such that $x \in [\![P_v]\!]$, then P_v holds at x, as in (38):

$$(42) \qquad\qquad [\![P_v]\!] \subseteq [n]$$

$$(43) \qquad\qquad P_v(x) \Longleftrightarrow x \in [\![P_v]\!]$$

The successor relation which links each string position to the next is also defined:

$$(44) \qquad\qquad S_n := \{(i, i+1) \mid i \in [n-1]\}$$

Now, for each $v \in \mathcal{V}$, the predicate P_v specifies all the string positions in which v occurs:

$$(45) \qquad\qquad [\![P_v]\!] := \{i \in [n] \mid v \in \sigma_i\}$$

If MSO$_{\mathcal{V}}$ is the set of sentences (closed formulas, with no free variables) of MSO whose vocabulary is limited to subsets of $\mathcal{V}$, then an MSO$_{\mathcal{V}}$ model $mod(s)$—which is described

---

[25]If $n = 0$, i.e. $s = \epsilon$, the empty string, then $[n] = \varnothing$, allowing a model to have an empty domain, as in Libkin (2004).

by the string s—is defined by the tuple (Fernando, 2016a):

$$(46) \qquad\qquad mod(s) := \langle [n], S_n, \{[\![P_v]\!] \mid v \in \mathcal{V}\} \rangle$$

Given an arbitrary MSO$_\mathcal{V}$ model M, the string $str(M)$ which describes it can be obtained by inverting (45) to get each set $\sigma_i \in str(M)$, for $i \in [n]$, where $[\![P_v]\!]_M$ is the interpretation of the predicate P_v relative to the model M[26]:

$$(47) \qquad\qquad \sigma_i := \{v \in \mathcal{V} \mid i \in [\![P_v]\!]_M\}$$

There is a fundamental theorem which was independently put forward by Büchi, Elgot, and Trakhtenbrot (Fernando, 2016a, p. 30)[27] which states that sentences φ of MSO capture regular languages—that is, *the MSO-definability of a language is equivalent to its regularity*. This leads to the following definition in Fernando (2018, p. 35) of the set of regular languages over the powerset $2^\mathcal{V}$ of the vocabulary $\mathcal{V}$[28], given by the sentences φ of MSO$_\mathcal{V}$:

$$(48) \qquad\qquad \{s \in (2^\mathcal{V})^* \mid mod(s) \models \varphi\}$$

Thus, languages of temporal strings are regular, and as such are open to manipulation and reasoning using finite-state techniques, due to the equivalence between regular languages and finite automata according to Kleene's theorem (see, for example, Yu (1997, p. 41)). For instance, the entailment—that is, set-theoretic inclusion—of one regular language by

---

[26]In general, for a given string s, it will be convenient to write the interpretation of a property P relative to the MSO$_\mathcal{V}$ model described by s, $[\![P]\!]_{mod(s)}$, as simply $[\![P]\!]_s$.

[27]A proof is given in Libkin (2004, p.124, Theorem 7.21).

[28]The powerset $2^\mathcal{V}$ is used here in place of the usual $\mathcal{V}$, due to the fact that strings may allow any number of fluents $v \in \mathcal{V}$ to hold at once.

another is decidable, which is not the case for First-Order Logic (Trakhtenbrot, 1953; Elgot and Rabin, 1966). This translates to being able to determine whether one string entails another, which is a powerful feature for ascertaining the relations which appear in a particular string, and for reasoning with them (see § 3.1.4, p. 65; § 4.3.1).

The fact that strings can be construed as models of MSO gives rise to a convenient method of comparison between strings, whereby the set of temporal relations (see Table 7, p. 78) found in one string s may be said to be equal to the set of relations in another string s' if s and s' are analogous.

Analogous Strings A string s will be said to be *analogous* to some other string s' if the MSO models corresponding to each string can be said to be, in a sense, isomorphic—that is, s and s' are of equal length, there exists a bijective function f mapping between the vocabulary of s and the vocabulary of s', and for every fluent $v \in \mathcal{V}_s$, its image $f(v) \in \mathcal{V}_{s'}$ appears in only those same string positions in s' as v appears in s:

(49) $mod(s) \cong mod(s') \Longleftrightarrow$

$$\text{length}(s) = \text{length}(s') \land \exists f \ (f : \mathcal{V}_s \leftrightarrow \mathcal{V}_{s'} \land \forall v \in \mathcal{V}_s \ (\llbracket P_v \rrbracket = \llbracket P_{f(v)} \rrbracket))$$

(50) $mod(s) \cong mod(s') \Longleftrightarrow s \sim s'$

For example, in (51), $\boxed{\ \ |a|\ |b|\ }$ is analogous to $\boxed{\ |c|\ |d|\ }$, while in (52) $\boxed{\ |a|\ |b|\ }$ is not analogous to $\boxed{\ |c|c,d|d|\ }$

(51) $\boxed{|a||b|} \sim \boxed{|c||d|}$

(52) $\boxed{|a||b|} \nsim \boxed{|c|c,d|d|}$

This definition of analogy can be lifted to languages L and L' simply by testing if all strings in L are analogous with all strings in L':

$$(53) \qquad L \sim L' := \forall(s \in L)\forall(s' \in L') \ (s \sim s')$$

Determining whether strings are analogous is useful when ascertaining the relations between events appearing in a string. If the relations between events in a string s are known, and another string s' can be shown to be analogous to s, then the relations between the events that appear in s' are also known[29]. By employing this concept in conjunction with that of projection (see p. 65) and a set of reference strings—such as those modelling Allen's relations (see table 7)—it becomes simple to determine which relations appear in a string.

Further, if two strings are shown to be analogous, this can improve the efficiency of any calculations or processing involving these strings: any set of operations applied to one of the strings would produce the same result if applied to the other, so there is no need to apply them to both. This is expanded upon further in §3.2.1, see p. 79.

3.1.3 Granularity: Points and Intervals

The vocabulary from which a string's alphabet is constructed is a set of fluents, or temporal propositions—for example "John sleeps" $Sleeps(john, t)$, where $Sleeps$ is a predicate and $john$ is an individual for which $Sleeps$ holds true for some time t. However, it has not yet been made explicit whether t should be an instantaneous point in time, or some temporal interval which has a non-zero duration. The need for this distinction is presented below.

---

[29]For example, the string on the left hand side of (51) says that event a is *before* event b, and since the string on the right hand side is analogous, THEN event c must be *before* event d.

In order for a string such as in (54) to be valid, where some event a occurs in multiple positions in the string, a must be an interval: a period of time with a beginning and an ending, and a is occurring between these.

(54) | | a | a, b | |

This allows for a to be represented via subdivisions[30] across the components within the string, such that the event a is understood as beginning at its first (leftmost) occurrence, and ending after its last (rightmost)—as long as it appears in a component, the event is occurring at that moment. While a point is instantaneous and therefore indivisible, an interval may be split into an arbitrary number of contiguous subintervals in this way, which allows for the representation of other events co-occurring with a specific subdivision of an interval, as in (54), where a is occurring during two string positions $[\![P_a]\!] = \{2, 3\}$, while b is only occurring during a single string position, $[\![P_b]\!] = \{3\}$. In this example, the real-time interval duration τ of position 3 is equal to that of the event b, which is also the span of time during which a and b are co-occurring. The summed duration τ' of positions 2 and 3 is equal to that of the event a, and so the duration of position 2 (the span of time in which a is occurring, but b has not begun) is equal to $\tau' - \tau$.

If a and b represent points of time, then the string in (54) is invalid—since a point is instantaneous, it may not occur at multiple string positions, which are discrete moments of time. Additionally, it is not possible for one point to have a shorter duration than another, and thus the fact that b occurs at the end of a is lost. Allen (1983) also discusses the logical and physical issues with allowing events to be representable as instantaneous points, but there is also a cognitive issue, discussed in Freksa (1992), where events must

---

[30] An interval may be subdivided theoretically *ad infinitum*, though the constraint of a finite vocabulary prevents this from occurring within a string.

have some non-zero duration in order to be perceivable (Hamblin, 1972).

That is not to say that this information cannot be represented in a string using points, but instead, a translation from events to *event borders* must occur, where event borders are points representing the beginnings and endings of events. Fernando (2018); Fernando and Vogel (2019) uses[31] $l(a)$ as the (open) left border and $r(a)$ as the (closed) right border of some event a—the *granularity* is altered to focus on the event borders (as points) rather than on the events (as intervals). In keeping with the analogy of strings being similar to strips of film, the change of granularity is akin to altering the level of 'camera zoom' to focus on details which were previously considered simply as parts of the whole.

This change does introduce additional complexity to the vocabulary, with the new set $\check{\mathcal{V}}$ constructed from the old $\mathcal{V}$ (Fernando, 2018, p. 37):

$$L = \{ \, l(v) \mid v \in \mathcal{V} \, \}$$

$$R = \{ \, r(v) \mid v \in \mathcal{V} \, \}$$

(55) $\qquad \check{\mathcal{V}} := L \cup R \qquad$ if L, R, and $\mathcal{V}$ are pairwise disjoint.

The cardinality of $\check{\mathcal{V}}$ is thus twice that of $\mathcal{V}$, with two symbols required to represent the same information of a single interval. The translation from a string $s = \sigma_1 \sigma_2 \cdots \sigma_n$ whose symbols are events (intervals) to a string $\check{s} = \check{\sigma}_1 \check{\sigma}_2 \cdots \check{\sigma}_n$ whose symbols are event borders (points) is given as (Fernando, 2018, p. 38):

(56) $\qquad \check{\sigma}_i := \begin{cases} \{r(a) \mid a \in \sigma_i\} & \text{if } i = n \\[2ex] \{l(a) \mid a \in \sigma_{i+1} - \sigma_i\} \cup \{r(a) \mid a \in \sigma_i - \sigma_{i+1}\} & \text{if } i < n \end{cases}$

---

[31] In a fashion similar to the S-words of Durand and Schwer (2008a,b), which also use the beginnings and endings of events as basic.

The symbol $l(a)$ appearing in $\breve{\sigma}_i$ says that the event a is not occurring at position i, but is occurring at position $i + 1$. The symbol $r(a)$ appearing in $\breve{\sigma}_i$ says that a is occurring at position i, but is not occurring at position $i + 1$. Using event borders in this way, the string in (54) would be translated to be drawn as:

(57)
$$\boxed{\;l(a)\;\vrule\;l(b)\;\vrule\;r(a),r(b)\;\vrule\;\;}$$

Points are, in some ways, simpler than intervals—for instance, two points may be related in just three ways ($<$, $=$, and $>$), while a pair of intervals have thirteen possible relations between them (precisely, the set of relations given by Allen's interval algebra (Allen, 1983), see Table 7, p. 78). Additionally, focusing on the borders of events opens up a pathway to representing incomplete information, by omitting one of the beginning or ending of an event from the vocabulary of a string, it is left unspecified, which is often the case in natural language. For example, in (58) below, it is known that the events a and b begin simultaneously, but it is not known when a ends:

(58)
$$\boxed{\;l(a),l(b)\;\vrule\;r(b)\;\vrule\;\;}$$

This situation cannot be simply represented using ordinary intervals, since for any symbol in a string's vocabulary, its appearance or non-appearance in a string component indicates explicitly whether or not the event is occurring at that moment, with no option for 'possibly occurring'.

This work, however, will primarily consider intervals as the basic unit of representation for fluents. When looking at a particular string component, it is more straightforward to tell whether some event a is occurring at that moment if the symbol a appears within

that component, than to have to either check backwards and forwards through the string to see if the event has begun ($l(a)$ appears somewhere to the left of the component of interest) and not yet ended ($r(a)$ does not appear to the left of the component of interest), or to have to perform a translation. This also more closely mirrors the intuition of the analogy with panels of a comic strip or frames of a reel of film, such that all events which are occurring at any particular moment are 'visible' within that panel or frame (string component). There is further a higher level of descriptiveness that can be achieved with thirteen relations between a pair of intervals, as opposed to requiring a set of four points to achieve the same.

Chiefly, however, among reasons to generally consider intervals as primitive is the fact that ISO-TimeML, the international standard for temporal annotation Pustejovsky et al. (2010), considers events and event-like entities to be intervals, and uses relations which are based in Allen (1983)'s set of interval relations. Further, Freksa (1992, p. 201) agrees with Allen from a cognitive perspective that events should not be "represented by points on the real line", and additionally describes the concept of *semi-intervals*—using intervals to represent the beginnings and endings of events—which allows for treating incomplete information similar to using points (as in (58) above), and expanding the number of possible relations to a set of 31 options. Nonetheless, semi-intervals do, again, introduce an additional level of complexity over plain intervals, and are discussed more completely in § 4.1.2.

It is worth noting here that, in general, in this work it is also assumed that any text which will be used as a source for the creation of strings will feature only *finite events*. The fluents which represent these events will therefore hold for a finite amount of time, and thus intervals appearing in strings will be *bounded* (Allen and Ferguson, 1994). That

is, for some event a, there is some time *immediately before a* holds during which a does not hold, and similarly, there is some time *immediately after a* holds during which a does not hold. This fact is represented through the use of bounding empty sets, and so any given string will both begin and end with an empty set, drawn as an empty box $\square$[32]. The framework does not, in fact, require this assumption to be true—compare the bounded (finite) interval a, drawn as $\boxed{\ \boxed{a}\ }$, and the non-bounded (infinite) interval b, drawn as $\boxed{b}$—and indeed, it is convenient to go beyond it when discussing the handling of incomplete data, although this moves away from plain intervals to semi-intervals (see §4.1.2).

The following section describes a number of operations which may be used with interval-based strings, though it should be pointed out that for each operation there is an equivalent which may be used for a string which treats event borders as basic instead of events.

3.1.4 String Operations

A number of operations are available for the manipulation and description of strings which represent sequences of times and events, and these are detailed below. Key among these is *superposition* (basic, asynchronous, and vocabulary-constrained), which may be used to combine the information from multiple strings, but additionally defined are: an operation to find the vocabulary of arbitrary strings; block compression, which removes duplicate string components; reduct, which filters a string to only contain a specified set of events; and projection, whereby a string can be said to contain the same temporal data as another. Many of these operations are also available at the language level, in general by performing the particular desired operation on each string within the language.

---

[32]Equivalently, as a formula of MSO: $\forall v \in \mathcal{V} \ (\neg P_v(1) \wedge \neg P_v(n))$.

Vocabulary: The *vocabulary* $\mathcal{V}$ of a string or language is the set of fluents which appear in it. For an arbitrary string $s = \sigma_1\sigma_2\cdots\sigma_n$, the vocabulary $\mathcal{V}_s$ of s may be determined by taking the union of the string's components:

$$(59) \qquad\qquad \mathcal{V}_s := \sigma_1 \cup \sigma_2 \cup \cdots \cup \sigma_n$$

and the vocabulary of a language is just the union of the vocabularies of the strings it contains:

$$(60) \qquad\qquad \mathcal{V}_L := \bigcup\{\mathcal{V}_s \mid s \in L\}$$

The vocabulary of a string is a key factor in the calculation of many other operations, notably projection (p. 65) and vocabulary-constrained superposition (p. 69).

Block Compression: Since the length of a string $s = \sigma_1\sigma_2\cdots\sigma_n$ does not reflect its real duration (due to the understanding that strings model intertial worlds—see § 3.1 p. 41), it is also not required that the length of time represented by any σ_i is equal to that represented by any σ_j, for $i \neq j$. Similarly, if a fluent symbol $v \in \mathcal{V}$ from the vocabulary appears in both σ_i and σ_{i+1}, this does not imply that the event represented by v has a duration twice as long as if it had only appeared in σ_i. Indeed, the symbol v may appear in any number of consecutive positions in s without affecting the interpretation of the real length of time of the event it represents. Further, if the string features a repeating component, i.e. $\sigma_i = \sigma_{i+1}$ for any $1 \leq i < n$, the interpretation of the string is not affected by the deletion of one of either σ_i or σ_{i+1}. So for example, the interpretation of the string $\boxed{a\,|\,a\,|\,a,b\,|\,b\,|\,b}$ is equal to the interpretation of the string $\boxed{a\,|\,a,b\,|\,b}$. A string featuring

such repetitions is said to contain *stutter*.

As a result, the *block compression* $bc(s)$ of a string s may be introduced, which removes any stutter present in s. This is defined as (Fernando, 2015; Woods et al., 2017):

$$(61) \qquad bc(s) := \begin{cases} s & \text{if } length(s) \leq 1 \\[2ex] bc(\sigma s') & \text{if } s = \sigma \sigma s' \\[2ex] \sigma bc(\sigma' s') & \text{if } s = \sigma \sigma' s' \text{ with } \sigma \neq \sigma' \end{cases}$$

Stutter may also be induced in a string which is *stutterless* (it does not contain stutter) by using the inverse of block compression, which will generate infinitely many strings[33]:

$$(62) \qquad bc^{-1}(bc(s)) := \sigma_1^+ \sigma_2^+ \cdots \sigma_n^+ \quad \text{if } bc(s) = \sigma_1 \sigma_2 \cdots \sigma_n$$

For example:

$$(63) \qquad bc^{-1}(\boxed{a\,|\,c}) = \{\boxed{a\,|\,c}, \boxed{a\,|\,a\,|\,c}, \boxed{a\,|\,c\,|\,c}, \boxed{a\,|\,a\,|\,c\,|\,c}, \ldots\}$$

Since these strings all block compress to the same string, they can be said to be equivalent under block compression. Specifically, strings s and s' are *bc-equivalent* iff $bc(s) = bc(s')$. This ability to generate infinitely many strings which have an equivalent interpretation allows for varying the length of a string as will be required in order to form a useful notion of superposition (see p. 61).

---

[33]If the string s features stutter, then $bc^{-1}(s)$ will not contain any strings with a length shorter than s, including $bc(s)$. To capture all possible bc-equivalent strings, s is block compressed before the inverse is applied.

Superposition: In its most basic form, the *superposition* s & s' of two strings $s = \sigma_1\sigma_2\cdots\sigma_n$ and $s' = \sigma'_1\sigma'_2\cdots\sigma'_n$ of equal length n is simply their component-wise union[34]:

(64)
$$\sigma_1\sigma_2\cdots\sigma_n \ \& \ \sigma'_1\sigma'_2\cdots\sigma'_n := (\sigma_1 \cup \sigma'_1)(\sigma_2 \cup \sigma'_2)\cdots(\sigma_n \cup \sigma'_n)$$

For example:

(65)
$$\boxed{a}\,\boxed{b}\,\boxed{c} \ \& \ \boxed{a}\,\boxed{c}\,\boxed{d} = \boxed{a}\,\boxed{b,c}\,\boxed{c,d}$$

This is easily extended to pairs of languages L & L' by collecting the superpositions of strings of equal lengths in each language:

(66)
$$L \ \& \ L' := \bigcup_{n \geq 0}\{s \ \& \ s' \,|\, s \in L \cap \Sigma^n, s' \in L' \cap \Sigma^n\}$$

The result L & L' of superposing two languages L and L' is also a language, and if L and L' are regular languages (due to strings being finite models of Monadic Second-Order Logic and the theorem due to Büchi, Elgot, and Trakhtenbrot—see § 3.1.2, p. 49), then L & L' is also regular (Fernando, 2004; Woods et al., 2017, p. 126). If L is accepted by the finite automaton $\langle Q, (2^{\mathcal{V}_L})^*, (q \xrightarrow{\sigma} r), q_0, F\rangle$ and L' is accepted by the finite automaton $\langle Q', (2^{\mathcal{V}_{L'}})^*, (q' \xrightarrow{\sigma'} r'), q'_0, F'\rangle$ then L & L' is computed by a finite automaton composed of the automata accepting each L and L': $\langle Q \times Q', (2^{\mathcal{V}_L \cup \mathcal{V}_{L'}})^*, ((q, q') \xrightarrow{(\sigma \cup \sigma')} (r, r')), (q_0, q'_0), F \times F'\rangle$.

Using languages provides more flexibility than strings alone, since non-determinism

---

[34]The vocabulary of the resulting string is, as might be expected, the union of the vocabularies of the original strings: $\mathcal{V}_{s \ \& \ s'} = \mathcal{V}_s \cup \mathcal{V}_{s'}$.

can be accounted for through variations between strings within a language. For example, in (67) below, the result of the superposition accounts for the alternate event sequences in the strings of the first input language.

(67) $\quad\quad\quad \{\boxed{a\,|\,b\,|\,c}, \boxed{a\,|\,c\,|\,b}\} \ \& \ \{\boxed{a\,|\,c\,|\,d}\} = \{\boxed{a\,|\,b,c\,|\,c,d}, \boxed{a\,|\,c\,|\,b,d}\}$

This may reflect a situation where there is uncertainty as to the correct order of events—in this case a language is useful to collect all of the possible alternatives, which can then be still be superposed with other languages.

Asynchronous Superposition: In order to extend this operation further, it is necessary to remove the restriction that only strings of equal length may be superposed together. This is desirable so as to allow arbitrary numbers of events to appear in strings, and to superpose strings which may be of unknown length. For example, the operation in (68) below cannot be calculated, and even if the strings were instead singleton members of languages and those languages were superposed, the result would just be the empty set.

(68) $\quad (\boxed{a\,|\,b} \ \& \ \boxed{c\,|\,d}) \ \& \ (\boxed{a\,|\,b\,|\,c} \ \& \ \boxed{a\,|\,c\,|\,d}) = \boxed{a,c\,|\,b,d} \ \& \ \boxed{a\,|\,b,c\,|\,c,d} = undefined$

To achieve this, bc-equivalence is exploited and the *inverse block compression* operation (see (62), p. 58) is leveraged. Since, by inducing stutter in a string, infinitely many new strings of greater or equal length can be generated which are bc-equivalent to the starting string, it is effectively possible to force a pair of strings to be of equal length.

So, the *asynchronous superposition* $s \ \&_* \ s'$ of two strings s and s' is initially defined

as the language obtained by applying block compression to the results of superposition between the languages which are respectively bc-equivalent to each of s and s'[35]:

(69) $$s \&_* s' := \{bc(s'') \mid s'' \in bc^{-1}(bc(s)) \ \& \ bc^{-1}(bc(s'))\}$$

Now the strings in (68) can be superposed using asynchronous superposition, as in (70) below:

(70) $\boxed{a,c}\boxed{b,d} \&_* \boxed{a}\boxed{b,c}\boxed{c,d} = \{\boxed{a,c}\boxed{a,b,c}\boxed{b,c,d}, \boxed{a,c}\boxed{a,b,d}\boxed{b,c,d},$

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

However, one slightly problematic aspect of this definition is the fact that bc^{-1} maps from a string to an infinite language. While this is not an issue from a theoretical standpoint, since $\&_*$ collects the set of block compressed strings from the superposition of these languages, from a practical and computational standpoint anything infinite is inconvenient.

In order to tackle this back to something finite and to avoid generation of large amounts of redundant information, in Woods et al. (2017, p. 127) an upper bound of $n + n' - 1$ is established for the maximum length of any string produced via asynchronous superposition $s \&_* s'$, where n and n' are the (nonzero) lengths of the strings s and s', respectively. This work additionally introduces the operation pad_k, which will perform inverse block compression on a string, but will only produce strings of a given length $k > 0$:

(71) $$pad_k(bc(s)) := \sigma_1^+ \sigma_2^+ \cdots \sigma_n^+ \cap \Sigma^k \quad \text{if } bc(s) = \sigma_1 \sigma_2 \cdots \sigma_n$$

$$= \{\sigma_1^{k_1} \sigma_2^{k_2} \cdots \sigma_n^{k_n} \mid k_1, \ldots, k_n \geq 1, \sum_{i=1}^{n} k_i = k\}$$

---

[35]Note that $bc(s) = bc(s') \iff s \in bc^{-1}(bc(s'))$

The language produced by padding a string is a proper subset of the language produced by performing inverse block compression on that same string. For example, using the same string as (63):

(72)
$$pad_3(\boxed{a\,|\,c}) = \{\boxed{a\,|\,a\,|\,c}, \boxed{a\,|\,c\,|\,c}\}$$

By using this padding operation in place of the inverse block compression in an updated definition of asynchronous superposition, setting k to be the upper bound derived from the lengths of the input strings, the issue of going beyond finite sets is avoided without losing any of the power of using bc-equivalence:

(73)
$$s \mathrel{\&_*} s' := \{bc(s'') \mid s'' \in pad_{n+n'-1}(s) \mathrel{\&} pad_{n+n'-1}(s')\}$$

It is important to remark here that neither basic superposition $\&$ nor asynchronous superposition $\&_*$ place any importance on the semantic content contained within the strings over which they operate. That is to say, they are entirely syntactical operations, and any meaningful information represented by a given string is liable to be lost once it has been superposed with some other string.

For instance, in (65), the second of the operand strings has the event c appearing in a box to the left of (before) the event d, whereas the result has c and d occurring in the same box together, which states that they were occurring at the same moment.

Similarly, in (70), the first input string has events a and c appearing in the same box, while the second input string has a appearing in a box before c. While this should seem like a contradiction which should not have viable results, the operation instead produces a set of four strings, the second and third of which are invalid according to (34),

which states that a given fluent may not appear in non-contiguous string positions—in $\boxed{a,c \mid a,b,d \mid b,c,d}$ the event c occurs in positions 1 and 3 but not position 2, and in $\boxed{a,c \mid a,b,c \mid a,c,d \mid b,c,d}$ the event b occurs in positions 2 and 4 but not in position 3.

This stands in contrast to the concept of bc-equivalence, where strings have an equal interpretation regardless of how much or how little stutter is present. By using superposition as it is currently presented, information is often, in fact, lost rather than gained when strings are combined. The next two operations, *reduct* and *projection*, aim to assist in the resolution of this issue.

Reduct: It will be useful to be able to alter the vocabulary of a string—in particular, to shrink it—so as to control which events are mentioned. If a string contains, for instance, five events, but only two of these are relevant to the application, there is sense in being able to focus on those two.

As such, for any set A of fluents, the *A-reduct* ρ_A of a string $s = \sigma_1\sigma_2 \cdots \sigma_n$ is defined as the componentwise intersection of s with A (Fernando, 2016a; Woods and Fernando, 2018):

(74) $$\rho_A(\sigma_1\sigma_2\cdots\sigma_n) := (\sigma_1 \cap A)(\sigma_2 \cap A) \cdots (\sigma_n \cap A)$$

The resulting new string has a vocabulary which is equal to or a subset of A, but the remaining fluents are still in the same relative positions to each other as in the original string s:

(75) $$\mathcal{V}_{\rho_A(s)} = \mathcal{V}_s \cap A$$

(76) $$\forall a \in A \; ([\![P_a]\!]_{mod(s)} = [\![P_a]\!]_{mod(\rho_A(s))})$$

For example, with the string $s = \boxed{}\boxed{a,b}\boxed{a,b,c}\boxed{a,c,d}\boxed{a,e}\boxed{e}\boxed{}$ and $A = \{a, d\}$, the A-reduct of s is:

$$(77) \qquad \rho_{\{a,d\}}\left(\boxed{}\boxed{a,b}\boxed{a,b,c}\boxed{a,c,d}\boxed{a,e}\boxed{e}\boxed{}\right) = \boxed{}\boxed{a}\boxed{a}\boxed{a,d}\boxed{a}\boxed{}$$

The resulting string in (77) contains only the events of interest (those mentioned in A), without loss of information. That is, the relative ordering of the events in the result string is the same as that in the input. The result string can additionally be block compressed to derive the simplest representation of the information it contains[36]:

$$(78) \qquad \mathrm{bc}\left(\boxed{}\boxed{a}\boxed{a}\boxed{a,d}\boxed{a}\boxed{}\right) = \boxed{}\boxed{a}\boxed{a,d}\boxed{a}\boxed{}$$

It's worth noting that for any pair of strings s and s' with equal length and disjoint vocabularies, the reduct of the result of basic superposition $s \ \& \ s'$ with respect to each of the strings' vocabularies is equal to the string itself:

$$(79) \qquad \mathcal{V}_s \cap \mathcal{V}_{s'} = \varnothing \implies \rho_{\mathcal{V}_s}(s \ \& \ s') = s \text{ and } \rho_{\mathcal{V}_{s'}}(s \ \& \ s') = s'$$

For example, with $s = \boxed{a}\boxed{b}$ and $s' = \boxed{c}\boxed{d}$, $\mathcal{V}_s = \{a, b\}$ and $\mathcal{V}_{s'} = \{c, d\}$:

$$(80) \qquad s \ \& \ s' = \boxed{a,c}\boxed{b,d}$$

$$\rho_{\{a,b\}}\left(\boxed{a,c}\boxed{b,d}\right) = \boxed{a}\boxed{b}$$

$$\rho_{\{c,d\}}\left(\boxed{a,c}\boxed{b,d}\right) = \boxed{c}\boxed{d}$$

---

[36]In the case of (77) and (78), the event a *contains* the event d, according to Allen's relations (Allen, 1983)—see also Figure 1, p. 8.

This shows how the information contained within s and s' is not lost in the superposition s & s'. For asynchronous superposition—which is built on basic superposition—this also holds true for each string[37] in the result language.

Projection: This process is streamlined through the use of *projection*, where the A-projection π_A of a string s is simply the block compressed reduct of s relative to A:

$$(81) \qquad \qquad \pi_A(s) := \mathrm{bc}(\rho_A(s))$$

It will be said that a string s *projects to* another string s', $s \sqsupseteq s'$, if the $\mathcal{V}_{s'}$-projection of s is equal to s':

$$(82) \qquad \qquad s \sqsupseteq s' \iff \pi_{\mathcal{V}_{s'}}(s) = s'$$

If s projects to s', then all of the information represented within s' is also represented in s—s effectively 'contains' s'. Trivially, any block compressed string $s = \mathrm{bc}(s)$ will project to itself, since the reduct of s with respect to its own vocabulary is s.

Borrowing from the examples in (77) and (78), the temporal data represented by the string on the right hand side of (83) is also contained in the string on the left hand side:

$$(83) \qquad \qquad \boxed{\;|\,a,b\,|\,a,b,c\,|\,a,c,d\,|\,a,e\,|\,e\,|\;} \sqsupseteq \boxed{\;|\,a\,|\,a,d\,|\,a\,|\;}$$

It is worth noting that, if strings s and s' share the same vocabulary, but are not equal, then neither can s project to s' nor s' project to s—this scenario suggests that s and s'

---

[37] True when the string $s'' \in s$ &$_*$ s' has been block compressed after the reduct. For example: $\mathrm{bc}(\rho_{\mathcal{V}_s}(s'')) = s$.

are incompatible, that they describe contradictory sequences of the same events:

$$(84) \qquad \mathcal{V}_s = \mathcal{V}_{s'} \wedge s \neq s' \implies \neg(s \sqsupseteq s' \vee s' \sqsupseteq s)$$

A language L can be said to project to another language L' if every string $s \in L$ projects to every string $s' \in L'$:

$$(85) \qquad L \sqsupseteq L' \iff \forall(s \in L)\forall(s' \in L') \; (s \sqsupseteq s')$$

This notion of projection is particularly useful, allowing for temporal reasoning when used in conjunction with the concept of *analogous* strings (see § 3.1.2, p. 50). Particular events can be simply extracted from larger, more complex strings, and compared against reference to determine the relations between and ordering of the events of interest.

Importantly, projection will also be used to enrich the asynchronous superposition (see p. 62) operation, injecting the currently-lacking 'semantic-ness' by ensuring that all results of superposing a pair of strings project to each of their input strings.

Generate and Test: The predominant issue with asynchronous superposition $\&_*$ as it stands is that it needs not *preserve projections*—that is, data can become lost or 'corrupted' when combining strings. If a string generated by superposition does not project back to both of the strings that were superposed to generate it, then information has effectively been lost, as it has become impossible to return to the original event relation data. Only those result strings in $s \&_* s'$ which do project back can be said to be a valid result of the superposition, in terms of preserving the temporal information.

For example, in (70) (p. 61), none of the result strings contain the information that

is represented in the input strings. This can be verified by testing whether any string in the result set projects to either of the inputs, which they do not:

$\boxed{a,c}\,\boxed{a,b,c}\,\boxed{b,c,d}$	$\not\sqsupseteq$	$\boxed{a,c}\,\boxed{b,d}$
$\boxed{a,c}\,\boxed{a,b,d}\,\boxed{b,c,d}$	$\not\sqsupseteq$	$\boxed{a,c}\,\boxed{b,d}$
$\boxed{a,c}\,\boxed{a,b,c}\,\boxed{a,c,d}\,\boxed{b,c,d}$	$\not\sqsupseteq$	$\boxed{a,c}\,\boxed{b,d}$
$\boxed{a,c}\,\boxed{b,c,d}$	$\not\sqsupseteq$	$\boxed{a,c}\,\boxed{b,d}$

$\boxed{a,c}\,\boxed{a,b,c}\,\boxed{b,c,d}$	$\not\sqsupseteq$	$\boxed{a}\,\boxed{b,c}\,\boxed{c,d}$
$\boxed{a,c}\,\boxed{a,b,d}\,\boxed{b,c,d}$	$\not\sqsupseteq$	$\boxed{a}\,\boxed{b,c}\,\boxed{c,d}$
$\boxed{a,c}\,\boxed{a,b,c}\,\boxed{a,c,d}\,\boxed{b,c,d}$	$\not\sqsupseteq$	$\boxed{a}\,\boxed{b,c}\,\boxed{c,d}$
$\boxed{a,c}\,\boxed{b,c,d}$	$\not\sqsupseteq$	$\boxed{a}\,\boxed{b,c}\,\boxed{c,d}$

Table 2: Failed projections for (70).

In fact, every string in (70) shares the same vocabulary, so by (84), it is not possible for any to project to any other.

In the case where the vocabularies of the two strings are identical, then none of the results will project to the original strings[38]. This implication follows from (84), since the vocabulary of every string $s'' \in s \,\&_* \, s'$ is $\mathcal{V}_{s \cup s'} = \mathcal{V}_s = \mathcal{V}_{s'}$:

(86) $\qquad \mathcal{V}_s \cap \mathcal{V}_{s'} = \mathcal{V}_s = \mathcal{V}_{s'} \implies \forall(s'' \in s \,\&_* \, s') \, \neg(s'' \sqsupseteq s \lor s'' \sqsupseteq s')$

The implication of (86) can be used to avoid unuseful superpositions: since the operation $\&_*$ has the potential to generate a large number of new strings, which can become costly from a computational perspective, it is prudent to test ahead of time whether every generated string will be spurious and unable to project back—as seen in Table 2.

Conversely, if the vocabularies of the input strings are disjoint, then every resulting string will project back, due to (79):

(87) $\qquad \mathcal{V}_s \cap \mathcal{V}_{s'} = \varnothing \implies \forall(s'' \in s \,\&_* \, s') \, (s'' \sqsupseteq s \land s'' \sqsupseteq s')$

---

[38]This assumes that $s \neq s'$. In the case where $s = s'$, there is exactly one string $s'' \in s \,\&_* \, s'$ such that $s = s' = s''$.

This can be seen by returning to the example in (80):

(88) $\boxed{a\,|\,b}$ $\&_*$ $\boxed{c\,|\,d}$ $= \{\boxed{a,c\,|\,a,d\,|\,b,d}, \boxed{a,c\,|\,b,d}, \boxed{a,c\,|\,b,c\,|\,b,d}\}$

$$\boxed{a,c\,|\,a,d\,|\,b,d} \;\sqsupseteq\; \boxed{a\,|\,b} \quad \boxed{a,c\,|\,a,d\,|\,b,d} \;\sqsupseteq\; \boxed{c\,|\,d}$$

$$\boxed{a,c\,|\,b,d} \;\sqsupseteq\; \boxed{a\,|\,b} \quad \boxed{a,c\,|\,b,d} \;\sqsupseteq\; \boxed{c\,|\,d}$$

$$\boxed{a,c\,|\,b,c\,|\,b,d} \;\sqsupseteq\; \boxed{a\,|\,b} \quad \boxed{a,c\,|\,b,c\,|\,b,d} \;\sqsupseteq\; \boxed{c\,|\,d}$$

Table 3: Preserved projections of (88).

However, in general, the vocabularies of the input strings to asynchronous superposition may overlap without being equal. It might be assumed that, in this case, some number greater than 0 but less than all of the resulting strings may project back—however, this assumption does not hold in fact. A counter-example is readily found:

(89) $\boxed{a\,|\,b}$ $\&_*$ $\boxed{b\,|\,c}$ $= \{\boxed{a,b\,|\,a,c\,|\,b,c}, \boxed{a,b\,|\,b,c}, \boxed{a,b\,|\,b\,|\,b,c}\}$

$$\boxed{a,b\,|\,a,c\,|\,b,c} \;\not\sqsupseteq\; \boxed{a\,|\,b} \quad \boxed{a,b\,|\,a,c\,|\,b,c} \;\not\sqsupseteq\; \boxed{b\,|\,c}$$

$$\boxed{a,b\,|\,b,c} \;\not\sqsupseteq\; \boxed{a\,|\,b} \quad \boxed{a,b\,|\,b,c} \;\not\sqsupseteq\; \boxed{b\,|\,c}$$

$$\boxed{a,b\,|\,b\,|\,b,c} \;\not\sqsupseteq\; \boxed{a\,|\,b} \quad \boxed{a,b\,|\,b\,|\,b,c} \;\not\sqsupseteq\; \boxed{b\,|\,c}$$

Table 4: Failed projections for (89).

Thus it is necessary to test whether each result of a superposition is valid when neither (86) nor (87) are true. This approach of generating then testing was initially taken in Woods et al. (2017)[39] to ensure that only valid strings were finally produced. Yet, this is

---

[39]Albeit, the testing algorithm used there was based on matching string positions rather than projections.

not without issue either—consider the following examples:

(90)

$$\boxed{\;\boxed{a}\;\boxed{b}\;}\ \&_*\ \boxed{\;\boxed{b}\;\boxed{c}\;} = \{\ \boxed{\;\boxed{a,b}\;\boxed{b,c}\;}\ ,\ \boxed{\;\boxed{b}\;\boxed{a,b}\;\boxed{b,c}\;}\ ,\ \boxed{\;\boxed{b}\;\boxed{a}\;\boxed{b,c}\;}\ ,$$

$$\boxed{\;\boxed{b}\;\boxed{a}\;\boxed{c}\;\boxed{b,c}\;}\ ,\ \boxed{\;\boxed{b}\;\boxed{a}\;\boxed{c}\;\boxed{b}\;}\ ,\ldots\}$$

(91)

$$\boxed{\;\boxed{a}\;\boxed{b}\;\boxed{c}\;}\ \&_*\ \boxed{\;\boxed{d}\;\boxed{c}\;\boxed{b}\;} = \{\ \boxed{\;\boxed{a,d}\;\boxed{a,c}\;\boxed{b}\;\boxed{c}\;}\ ,\ \boxed{\;\boxed{d}\;\boxed{a,d}\;\boxed{b,d}\;\boxed{b,c}\;}\ ,\ \boxed{\;\boxed{d}\;\boxed{c}\;\boxed{b}\;\boxed{a,b}\;\boxed{b}\;\boxed{c}\;}\ ,$$

$$\boxed{\;\boxed{a}\;\boxed{b}\;\boxed{c}\;}\ \boxed{\;\boxed{d}\;\boxed{c}\;\boxed{b}\;}\ ,\ \boxed{\;\boxed{a}\;\boxed{b}\;\boxed{c,d}\;\boxed{c}\;\boxed{b,c}\;\boxed{b}\;}\ ,\ldots\}$$

The language result of (90) contains 270 strings, and in fact, only one of these will project back to both of the input strings: namely, $\boxed{\;\boxed{a}\;\boxed{b}\;\boxed{c}\;}$. This makes an intuitive sense, since the inputs are 'a before b' – $\boxed{\;\boxed{a}\;\boxed{b}\;}$ and 'b before c' – $\boxed{\;\boxed{b}\;\boxed{c}\;}$, and this result string is the only possibility where the linear ordering of the events a, b, and c is retained.

Using projection to test each of the generated strings in (90) and rejecting those which fail to project back to the inputs will produce the singular correct result, though it is rather inefficient: 269 (over 99%) of the generated results must be discarded in this example. The language result of (91) contains 257 strings, and this time 100% of them must be discarded: not one of the results projects back to both of the inputs. Clearly, the computational effort required to produce so many non-viable strings is entirely wasted, and so a modified approach is required to avoid this issue.

In Woods and Fernando (2018), a new version of superposition is defined which integrates projection-based testing into the generation process. This prevents problematic strings from ever being produced, improving on the efficiency of asynchronous superposition.

Vocabulary-Constrained Superposition: In order to define *vocabulary-constrained superposition* $\&_{vc}$, begin by fixing an infinite set of fluents Θ as well as the set of finite subsets of Θ, $Fin(\Theta)$, such that for any string s, $\mathcal{V}_s \in Fin(\Theta)$ and thus $s \in Fin(\Theta)^*$. Given a pair of finite subsets of Θ, $\Sigma \in Fin(\Theta)$ and $\Sigma' \in Fin(\Theta)$, an operation $\&_{\Sigma,\Sigma'}$: $(Fin(\Theta)^* \times Fin(\Theta)^*) \to 2^{Fin(\Theta)^*}$ is defined, mapping a pair of strings s and s' to a language $s \&_{\Sigma,\Sigma'} s'$ as follows, where ϵ is the empty string (of length 0)[40]:

(92a) $$\epsilon \&_{\Sigma,\Sigma'} \epsilon := \{\epsilon\}$$

(92b) $$\epsilon \&_{\Sigma,\Sigma'} s := \varnothing \quad \text{if } s \neq \epsilon$$

(92c) $$s \&_{\Sigma,\Sigma'} \epsilon := \varnothing \quad \text{if } s \neq \epsilon$$

and with $\sigma \in Fin(\Theta), \sigma' \in Fin(\Theta)$

(92d)
$$\sigma s \&_{\Sigma,\Sigma'} \sigma' s' := \begin{cases} \{(\sigma \cup \sigma')s'' \mid s'' \in L(\sigma, s, \sigma', s', \Sigma, \Sigma')\} & \text{if } \Sigma \cap \sigma' \subseteq \sigma \text{ and } \Sigma' \cap \sigma \subseteq \sigma' \\ \varnothing & \text{otherwise} \end{cases}$$

where

(92e) $\quad L(\sigma, s, \sigma', s', \Sigma, \Sigma') := (\sigma s \&_{\Sigma,\Sigma'} s') \ \cup \ (s \&_{\Sigma,\Sigma'} \sigma' s') \ \cup \ (s \&_{\Sigma,\Sigma'} s')$

If $\Sigma = \Sigma' = \varnothing$, then the condition in the first case of (92d) ($\Sigma \cap \sigma' \subseteq \sigma$ and $\Sigma' \cap \sigma \subseteq \sigma'$) holds vacuously, and $\&_{\Sigma,\Sigma'}$ becomes effectively identical to asynchronous superposition

---

[40]It follows from this definition that any string in $s \&_{\Sigma,\Sigma'} s'$ will have a length less than $n + n'$ where n and n' are the lengths of s and s', respectively, which is the same upper bound found in Woods et al. (2017) (see p. 61).

$\&_*$. Otherwise, this condition can be used to eject those strings which do not project back to both s and s', according to Proposition 1 and Corollary 2 in Woods and Fernando (2018, p. 81), reproduced below.

Proposition 1. *For all $\Sigma \in Fin(\Theta), \Sigma' \in Fin(\Theta)$ and $s \in Fin(\Theta)^*, s' \in Fin(\Theta)^*$, $s \&_{\Sigma,\Sigma'} s'$ selects those strings from asynchronous superposition $s \&_{\varnothing,\varnothing} s'$ which project to both the Σ-projection of s and the Σ'-projection of s':*

$$(93) \qquad s \&_{\Sigma,\Sigma'} s' \;=\; \{s'' \in s \&_{\varnothing,\varnothing} s' \mid s'' \sqsupseteq \pi_\Sigma(s) \wedge s'' \sqsupseteq \pi_{\Sigma'}(s')\}$$

Corollary 2. *For all $s \in Fin(\Theta)^*, s' \in Fin(\Theta)^*$ such that s and s' are stutterless, if $\Sigma = \mathcal{V}_s$ and $\Sigma' = \mathcal{V}_{s'}$, then $s \&_{\Sigma,\Sigma'} s'$ selects those strings from asynchronous superposition $s \&_{\varnothing,\varnothing} s'$ which project to s and s':*

$$(94) \qquad s \&_{\Sigma,\Sigma'} s' = \{s'' \in s \&_{\varnothing,\varnothing} s' \mid s'' \sqsupseteq s \wedge s'' \sqsupseteq s'\}$$

According to Corollary 2, *vocabulary-constrained superposition* $\&_{vc}$ can be used to preserve temporal information under projection during superposition:

$$(95) \qquad s \&_{vc} s' := s \&_{\mathcal{V}_s, \mathcal{V}_{s'}} s'$$

Now, this new form of superposition can be used for the same example as (90), and only the one valid result will be produced:

$$(96) \qquad \boxed{\;\boxed{a}\;\boxed{b}\;} \&_{vc} \boxed{\;\boxed{b}\;\boxed{c}\;} = \{\boxed{\;\boxed{a}\;\boxed{b}\;\boxed{c}\;}\}$$

Note that this result is still a language (a set of strings), and that there may be more than one string in this language, depending on the input strings. Where each input's vocabulary has a cardinality of 2, and the intersection of their vocabularies has cardinality 1, then

the number of result strings from the vocabulary-constrained superposition of the inputs corresponds with the transitivity table in Allen (1983, Fig. 4). For example, (97) shows the string 'a overlaps b' – $\boxed{a \mid a,b \mid b}$ superposed with 'b during c' – $\boxed{c \mid b,c \mid c}$, and according to Allen's transitivity table there should be three results, corresponding to 'a during c' – $\boxed{c \mid a,c \mid c}$, 'a overlaps c' – $\boxed{a \mid a,c \mid c}$, and 'a starts c' – $\boxed{a,c \mid c}$, and in fact this is the result shown by Table 5[41].

(97) $\quad \boxed{a \mid a,b \mid b} \ \&_{vc} \ \boxed{c \mid b,c \mid c} = \{ \boxed{c \mid a,c \mid a,b,c \mid b,c \mid c}, \ \boxed{a \mid a,c \mid a,b,c \mid b,c \mid c},$

$\boxed{a,c \mid a,b,c \mid b,c \mid c} \}$

$$\boxed{c \mid a,c \mid a,b,c \mid b,c \mid c} \quad \sqsupseteq \quad \boxed{c \mid a,c \mid c}$$

$$\boxed{a \mid a,c \mid a,b,c \mid b,c \mid c} \quad \sqsupseteq \quad \boxed{a \mid a,c \mid c}$$

$$\boxed{a,c \mid a,b,c \mid b,c \mid c} \quad \sqsupseteq \quad \boxed{a,c \mid c}$$

Table 5: Projections of (97) matching Allen's transitivities.

In Woods and Fernando (2018, p. 82) some simple benchmark tests were run, comparing the time (in milliseconds[42]) taken to compute the superpositions of a number of pairs of strings, using each of asynchronous superposition (generating then testing) and vocabulary-constrained superposition (testing while generating). These figures indicate a notable increase in the efficiency of time to calculate the correct results in using vocabulary-constrained superposition. Each of the strings in Table 7, p. 78 is superposed with itself and each of the others (e.g. *before* with *before*, *before* with *after*, *before* with *meets*, ..., *finished by* with *started by*, *finished by* with *finished by*), while varying the vocabularies of the operand strings as follows: first, both strings had the same vocabulary $\{a,b\}$; second, the strings shared one fluent in common, $\{a,b\}$ and $\{b,c\}$; finally, the strings had disjoint vocabularies, $\{a,b\}$ and $\{c,d\}$. A fragment of these tests is shown in Table 6.

---

[41]The set of Allen Relations can be seen represented as strings in Table 7, p. 78.

[42]The mean time of 1001 runs is given. The testing environment was Node.js v10.0.0 (64-bit) on Ubuntu 16.04 using an Intel i7-6700 CPU with 16GB of memory.

	$\triangle = *$	$\triangle = vc$	Decrease in time
$\boxed{\,\boxed{a}\ \boxed{b}\,}\ \&_{\triangle}\ \boxed{\,\boxed{b}\ \boxed{a}\,}$	0.3207ms	0.0180ms	94.39%
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$\boxed{\,\boxed{a}\ \boxed{b}\,}\ \&_{\triangle}\ \boxed{\,\boxed{b}\ \boxed{c}\,}$	0.3207ms	0.0659ms	79.45%
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$\boxed{\,\boxed{a}\ \boxed{b}\,}\ \&_{\triangle}\ \boxed{\,\boxed{c}\ \boxed{d}\,}$	22.4016ms	5.3616ms	76.07%
$\vdots$	$\vdots$	$\vdots$	$\vdots$

Table 6: Fragment of speed comparison between $\&_*$ and $\&_{vc}$.

The mean percentage decrease for all pairs with identical vocabularies using vocabulary-constrained superposition was 74.28%. With one shared fluent, the mean decrease was 34.11%, and with no fluents in common, it was 64.51%.

The main operations for string-based finite-state temporality have now been defined: block compression, reduct, projection, and (vocabulary-constrained) superposition. These operations may be leveraged in a number of ways in order to create useful applications.

3.2   Applications

Having discussed how to create and manipulate strings which represent temporal data—the linear order and inter-relations of a set of events—the following will explore some of the possible applications of this technology, including the ability to create timelines from annotated texts, to verify that the narrative structure of an annotated text is internally consistent with regards to the relations it depicts between its events, and to infer information not explicitly stated in an annotated text based on the event relations that can be extracted. These abilities can be used as part of tooling for the automatic aiding of creation of temporal annotation in new texts, for automated generation of summaries of a text, or fact-checking via corroboration of event sequences between sources.

Additionally, strings can be used as a tool for other, related applications which deal with sequential data through the use of external constraints. An example of such an application is given via a solution to a variant of the well-known Zebra Puzzle which models temporal constraints (that is, scheduling constraints) rather than spatial ones.

3.2.1 Timelines from Texts

Strings, as entities which are comprised of sequential components representing temporal data, have an intuitive comparison to a traditionally linear view of time. That is, that events which have not yet occurred are ahead of us, and events which occurred in the past are behind us—although not all languages or cultures perceive time in this way, it is common cross-linguistically to use some spatial reference points when discussing temporality (even absent vision, see Bottini et al. (2015))—Lakoff and Johnson (2008, pp. 42–43), for example, discusses the metaphor of "Time is a moving object", where the future is perceived as moving towards us, and Mitchell (1980, p. 542) argues that "we literally cannot 'tell time' without the mediation of space". Often, time is visualised as a line which travels along one of the axes in a three-dimensional space—from left to right, bottom to top, backward to forward, or the reverse of any of these—with events appearing as points or spans along the line, although according to Rosenberg and Grafton (2013, p. 14), the particular modern definition of a *timeline* as a "single axis and a regular, measured distribution of dates ... is not even 250 years old".

Regardless of the spatial orientation or directionality, this perception of time as coming from somewhere and going to somewhere else maps well to a sequential representation. The core concept being that, if two events exist at different moments in time, then they can be put in some kind of spatial ordering corresponding to the temporal ordering. The strings described in § 3.1 are used to model sequences of events in such a way that they can be read in a manner similar to a series of snapshots or film reel, or like the panels of a comic: each 'image' or moment of time features all of the events which are occurring at that time (relative to some fixed vocabulary of events).

An example of timelines in use is via Gantt charts, also known as harmonograms, which are a kind of bar chart that—like strings—display events over time. They are often used as a visual aid to show project schedules or similar temporal data (Kumar, 2005). The vertical axis shows the events mentioned in the chart, and the horizontal axis represents time intervals, with the width of the bars showing the duration of each event, and the beginnings and endings also illustrated by the bars' horizontal placement. See,

for example, Figure 12 below.
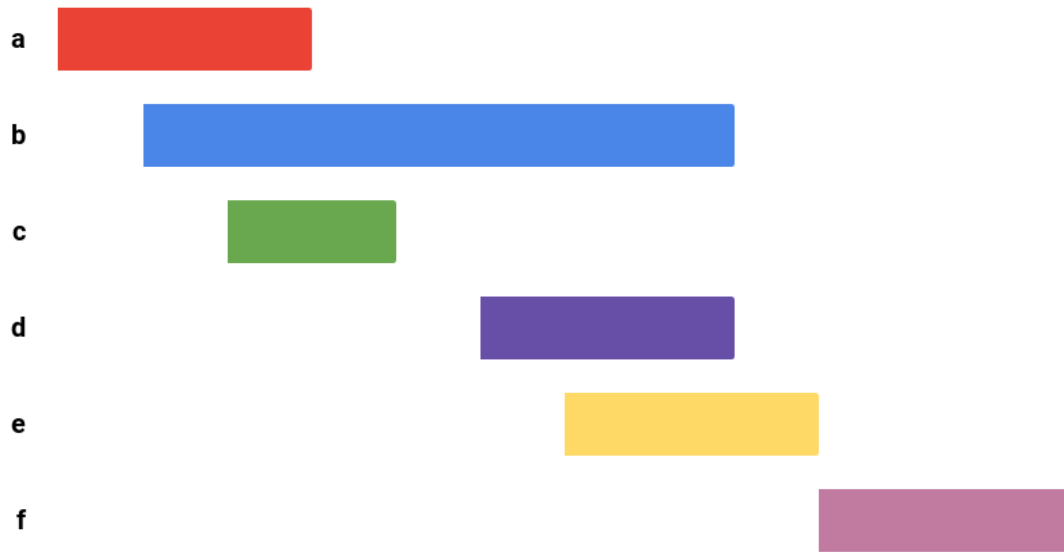
Figure 12: A Gantt chart featuring six events.

A chart like this is a useful visual display tool, though it can become a little unwieldy with large numbers of events, and there have been claims that they should be left behind in the field of project management (Maylor, 2001). Compare the same temporal data in Figure 12 shown in a single string in (98):

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

No matter how many events may be of interest, if all beginnings, endings, and durations are known—that is, the *temporal closure* has been calculated—such as they could be displayed in a Gantt chart, then they can be represented in a single string, with reducts and projections available in order to focus an analysis on a subset of events if desired, demonstrating a compact method of representing the timeline. However, in general, natural language texts do not provide enough precise temporal detail to determine all of this data. Discourse is typically somewhat vague and relies heavily upon the use of context to determine and precisely temporally locate events. As a result, it's often not possible to immediately derive the temporal closure just from a text, even an annotated text such as

one of the documents of the TimeBank (Pustejovsky et al., 2006) corpus, which is marked up to show the events, times, and temporal relations found within—see also §2.2.1. With that said, even if a single, unified timeline cannot be constructed due to ambiguity, the information in one of these annotated texts may still be extracted and used to build strings, which can provide a visual picture of the content of the document, which may reveal insights not obvious when looking at the text alone.

A string, like a timeline, can be thought of as a simplistic narrative, depicting a world by the events which take place in it. A language of strings, then, is a set of alternate timelines, of parallel worlds. Each string in a language represents a different possible world, describing alternate sequences of the events which took place, and each of which might be considered equally probably to be *veridical*—the actually true world, the correct sequence of events which occurred in reality—without further data and constraints. This allows strings as a framework to capture the ambiguity that very often arises from interpreting the temporal information that is derived from a natural language text. For example, given the information that some event a meets some other event b, and b is during some third event c, Allen's transitivities—see Table 1, p. 11—dictate that the relation between a and c can be one of overlaps, during, or starts. These first two relations a meets b and b during c may be represented by the strings $\boxed{\ \boxed{a}\boxed{b}\ }$ and $\boxed{\ \boxed{c}\boxed{b,c}\boxed{c}\ }$, respectively— see Table 7, p. 78—and in fact, the three possible relations between a and c are found by projection relative to $\{a, c\}$ over the result found from superposing thsee two strings, as in (99) below:

(99a) $\quad \boxed{\ \boxed{a}\boxed{b}\ } \ \&_w \ \boxed{\ \boxed{c}\boxed{b,c}\boxed{c}\ } = \{\ \boxed{\ \boxed{c}\boxed{a,c}\boxed{b,c}\boxed{c}\ },\ \boxed{\ \boxed{a}\boxed{a,c}\boxed{b,c}\boxed{c}\ },\ \boxed{\ \boxed{a,c}\boxed{b,c}\boxed{c}\ }\}$

(99b) Under projection to $\{a, c\} = \{\ \boxed{\ \boxed{c}\boxed{a,c}\boxed{c}\ },\ \boxed{\ \boxed{a}\boxed{a,c}\boxed{c}\ },\ \boxed{\ \boxed{a,c}\boxed{c}\ }\}$

In (99a), the language result of the superposition contains three strings, or three parallel worlds depicting the relational and ordering information that is known about the events a, b, and c, with (99b) showing the language under projection, which gives three strings depicting the three possible relations between a and c—see Table 7. Unless some further data becomes available constraining these possibilities, there is no way to tell which of the

three results of (99a) is the correct timeline, and so the language as a whole is considered valid. It is important to be wary, though, as too many of these parallel worlds can become unwieldy, just as Gantt charts can become difficult to keep track of if too large a number of events are depicted. One way to avoid an explosion in the number of timelines to keep track of is to avoid superpositions where the inputs do not share some vocabulary, or where the number of strings in the result would exceed some predetermined limit. Another possibility is to alter the granularity of the string, using semi-intervals (Freksa, 1992) rather than plain intervals, although this does have its own tradeoffs—see § 4.1.2.

Conversely to considering a whole language as valid, occasionally an annotated text may feature inconsistencies, in that relations may be indicated which are impossible for one reason or another, whether that comes from human error on the part of a manual annotator, a poor machine annotation, or simply a text whose narrative is inconsistent. Whatever the source, it is important to be congnisant of these potential issues, as they are likely to lead to at least partially incorrect conclusions being drawn about the timeline of the narrative. Using vocabulary-constrained superposition, these inconsistencies can be discovered as the result of superposing a pair of strings which represent incompatible temporal data will be an empty set. For example:

$$(100) \qquad\qquad \boxed{\;\boxed{a}\;\boxed{b}\;}\;\&_{vc}\;\boxed{\;\boxed{b}\boxed{a,b}\boxed{b}\;} = \varnothing$$

This will always be the case, no matter how many events may appear in either string, and thus the relations between intervals appearing in a string effectively become constraints, which the string models: each string in (100) represents a different constraint, which are incompatible with each other—see also § 3.2.2. When doing superposition of languages, if all of the strings from the first language are incompatible with all of the strings from the second, the result will also be an empty language—although, if some of the strings are compatible, then those superpositions are returned, as in (101) below, where the only strings which are mutually consistent are the second string of the first language and the

first string of the second:

(101) $\{\,\boxed{\;\boxed{a}\;\boxed{b}\;\boxed{c}\;}\,,\,\boxed{\;\boxed{a\;b}\;\boxed{c}\;}\,\}\;\&_{vc}\;\{\,\boxed{\;\boxed{d\;a}\;\boxed{b}\;}\,,\,\boxed{\;\boxed{b,d}\;\boxed{a}\;}\,\}\;=\;\{\,\boxed{\;\boxed{d}\;\boxed{a}\;\boxed{b}\;\boxed{c}\;}\,\}$

A point of interest is that the thirteen interval relations given by the interval algebra of Allen (1983) fall out of the vocabulary-constrained superposition of a pair of strings which each feature a single and different finitely-bounded event[43]:

(102) $$\mathcal{AR} := \{<,>,m,mi,o,oi,d,di,s,si,f,fi,=\}$$

(103) $$\boxed{\;\boxed{v}\;}\;\&_{vc}\;\boxed{\;\boxed{v'}\;}\;=\;\{\mathcal{S}_\bullet(v,v')\mid \bullet\in\mathcal{AR}\}$$

Each string $\mathcal{S}_\bullet(v,v')$ of the result set features one relation $v\bullet v'$, as shown in Table 7, reproduced from Woods and Fernando (2018, p. 79, Table 1).

$\bullet$	$v\bullet v'$	$\mathcal{S}_\bullet(v,v')$	$\bullet^{-1}$	$v\,\bullet^{-1}\,v'$	$\mathcal{S}_{\bullet^{-1}}(v,v')$
$<$	v before v'	$\boxed{\boxed{v}\;\boxed{v'}}$	$>$	v after v'	$\boxed{\boxed{v'}\;\boxed{v}}$
m	v meets v'	$\boxed{\boxed{v}\boxed{v'}}$	mi	v met by v'	$\boxed{\boxed{v'}\boxed{v}}$
o	v overlaps v'	$\boxed{\boxed{v}\boxed{v,v'}\boxed{v'}}$	oi	v overlapped by v'	$\boxed{\boxed{v'}\boxed{v',v}\boxed{v}}$
d	v during v'	$\boxed{\boxed{v'}\boxed{v,v'}\boxed{v'}}$	di	v contains v'	$\boxed{\boxed{v}\boxed{v',v}\boxed{v}}$
s	v starts v'	$\boxed{\boxed{v,v'}\boxed{v'}}$	si	v started by v'	$\boxed{\boxed{v',v}\boxed{v}}$
f	v finishes v'	$\boxed{\boxed{v'}\boxed{v,v'}}$	fi	v finished by v'	$\boxed{\boxed{v}\boxed{v',v}}$
$=$	v equals v'	$\boxed{\boxed{v,v'}}$			

Table 7: Allen interval relations in strings.

It can be said that for some string s, that s *entails* one of the Allen relations if and only if s projects to the string in Table 7 corresponding to that relation:

(104) $$s\models a\bullet b\Longleftrightarrow s\sqsupseteq\mathcal{S}_\bullet(a,b)$$

---

[43]The superposition of more than two unconstrained intervals in this manner creates a rapidly expanding number of strings. Three intervals gives 409 strings, and by six intervals it has already exceeded three hundred million strings—see Woods et al. (2017, p. 129) and the full sequence in `https://oeis.org/A055203` (Schwer, 2000). This is obviously an excessive amount of information to process, and so generally superposition is to be avoided where there are no constraints between the intervals in one string and the other—that is, where there is no shared vocabulary between the strings to be superposed.

Additionally, using the notion of analogous strings (see §3.1.2, p. 50), any block compressed string which has a vocabulary of cardinality 2 can be compared to the strings in Table 7. If such a string $s \sim \mathcal{S}_\bullet(v, v')$ for some $\bullet \in \mathcal{AR}$, then the events appearing in s can also be said to be related by $\bullet$. For example, $\boxed{\,|\,c\,|\,d,c\,|\,}\, \sim \mathcal{S}_{\mathrm{fi}}(v, v')$, and thus the relation between the events c and d is 'c finished by d'.

This is easily extended beyond strings featuring just two fluents. The relations between the events appearing in the string $s = \boxed{\,|\,a\,|\,|\,b\,|\,|\,c\,|\,}$ can be determined by taking its block compressed reduct relative to the subsets of the vocabulary which have cardinality 2— in this case, a is before b, b is before c, and a is before c. Once these relations have been calculated, the relations between the events in another string $s' = \boxed{\,|\,d\,|\,|\,e\,|\,|\,f\,|\,}$ are immediately available on analogy $s \sim s'$.

While this is a relatively simple example, it can be extended for strings featuring any arbitrary number of events, and perhaps more usefully it can be used to shortcut superpositions and other string operations. For instance, given the pair of strings $s = \boxed{\,|\,a\,|\,a,b\,|\,b\,|\,}$ and $s' = \boxed{\,|\,c\,|\,b,c\,|\,c\,|\,}$, the vocabulary-constrained superposition is calculated as in (97):

$$(105) \qquad s \,\&_{vc}\, s' = \{ \boxed{\,|\,c\,|\,a,c\,|\,a,b,c\,|\,b,c\,|\,c\,|\,},\ \boxed{\,|\,a\,|\,a,c\,|\,a,b,c\,|\,b,c\,|\,c\,|\,},$$
$$\boxed{\,|\,a,c\,|\,a,b,c\,|\,b,c\,|\,c\,|\,} \}$$

Now, given two more strings $t = \boxed{\,|\,x\,|\,x,y\,|\,y\,|\,}$ and $t' = \boxed{\,|\,z\,|\,y,z\,|\,z\,|\,}$ such that $s \sim t$ and $s' \sim t'$, the generated results of superposing t and t' will also be analogous to the results in (105), and so there is no need to calculate $t \,\&_{vc}\, t'$. Since the strings are analogous, there is a bijective mapping between the vocabularies $f : (\mathcal{V}_s \cup \mathcal{V}_{s'}) \leftrightarrow (\mathcal{V}_t \cup \mathcal{V}_{t'})$, and applying this function f to the results in (105) gives the same result as calculating the

superposition $t \,\&_x\, t'$:

(106) $\quad f(s \,\&_x\, s') = \{\boxed{\;\boxed{z}\,\boxed{x,z}\,\boxed{x,y,z}\,\boxed{y,z}\,\boxed{z}\;},\; \boxed{\;\boxed{x}\,\boxed{x,z}\,\boxed{x,y,z}\,\boxed{y,z}\,\boxed{z}\;},$

$\qquad\qquad\qquad \boxed{\;\boxed{x,z}\,\boxed{x,y,z}\,\boxed{y,z}\,\boxed{z}\;}\}$

(107) $\qquad\qquad\quad = \; t \,\&_x\, t'$

Again, this is a small example, but with larger numbers of events and more complex strings, leveraging the power of analogous strings has the potential to massively reduce the computational cost to calculate superpositions.

By extracting the relations from annotated text as strings and combining them using superposition, a timeline can be built up, which can assist an annotator or other reader in visualising the overall temporal structure of the text. It allows them to check for consistency and also provides a basis for interpreting the narrative in terms of the events that it describes.

3.2.2 Constraints and Scheduling (Zebra Puzzle)

Scheduling as a general concept—whereby tasks or events are allocated an ordering according to some set of rules or constraints—is a multi-faceted problem that has been a subject of research for many years (Manne, 1960; Applegate and Cook, 1991; Pinedo and Hadavi, 1992; Gong et al., 2018). Available resources must be taken into account, and often it is desirable to find the most efficient way to order the events which are to appear in the schedule so as to minimise the amount of time required for all relevant events to finish. Here it will be shown how the strings described in § 3.1 can be applied to some scheduling and scheduling-like tasks.

The Zebra Puzzle, also sometimes known as Einstein's Riddle is a logic problem, involving solving a set of clues in order to assign a number of properties to a set of individuals. In the original puzzle, there are five houses in a street, each of a different colour, and in each lives a person of different nationality, who drinks a different beverage, smokes a different brand of cigarettes, and owns a different pet. The puzzle provides

clues as to which house contains which set of all of the properties, except that the person who owns a zebra as a pet is not specified, and must be deduced by arranging all of the other elements into their correct houses so that the zebra's home can be determined by process of elimination. This may seem to be a very distinct problem from scheduling, but in fact, there are a number of parallels. Both problems can be viewed as constraint satisfaction problems, and although the Zebra Puzzle concerns spatial constraints, it is not a large stretch to model the street as a sequence of houses, and thus be able to use strings to represent the clues which can be superposed to solve the riddle. If a 'house' is conceptualised as a set which contains as elements all of the properties associated with that house—for example, $\{red, english, zebra, coffee, kools\}$—and the street is a sequence of these sets, then the street can be depicted as a string. The left and right spatial relations are thought of in the same way as the previous and subsequent boxes in a string.

To make this connection clearer, below is presented a variant of the puzzle using clues pertaining to temporal relations instead of spatial ones. The clues to the puzzle are as follows in Table 8:

There are five weekdays.

The foggy day is mild.

I am tired on the warm day.

There is a traffic jam on the overcast day.

It is cold on the day with little traffic.

It is overcast the day after it snows.

I'm sad the day that I'm reading.

It rains the day I have printing to do.

The traffic is average in the middle of the week.

It's freezing at the beginning of the week.

The stapling is done the day before or the day after I'm happy.

Printing happens the day before or the day after I'm angry.

There is a lot of traffic the day that filing happens.

Shredding happens the day it's hot.

It's freezing the day before or the day after the weather is clear.

Table 8: Temporal Zebra puzzle clues in English.

Using these clues, it should be possible to answer these questions:

- What day is there no traffic?

- What day am I curious?

The puzzle makes three assumptions: first, that the values as presented are *discrete* rather than continuous—'freezing' and 'cold', for example, are similar concepts (indeed, if the weather is freezing, then it is also cold), but for the purposes of this puzzle they are treated as being separate and unrelated; second, that each value lasts for only one day— if it rains on one of the days, it will not rain on any other; and third, each day only has one value per attribute—for example, only one task can be performed on any given day. Effectively, recalling that since the strings model inertial worlds, each day contains event-like statives which are treated as having a duration of the entire day. The particular values that appear may make it seem a little absurd—it is rather unlikely that a person

would be doing something like printing for a full day, and then immediately begin stapling for another full day, and so on—however, they should suffice for the sake of the example.

For each of the given clues, a constraint can be constructed from one or more strings. Additional constraints can be formed to represent the external assumptions, and by superposing these constraints together, a solution to the puzzle can be found. The strings corresponding to each clue are given in Table 9.

For most of the clues, the constraint is formed as a set of a few strings: when event a and b occur on the same day, they will appear in the same box, but it's unknown whether they occur at the beginning of the week ($\boxed{a, b}\,$), the end of the week ($\,\boxed{a, b}$), or somewhere in the middle ($\,\boxed{a, b}\,$), and so all of the possibilities appear together. It's also worth noting that strings use the Allen relation 'meets' where the clue states "the day before", rather than the 'before' relation. This is due to the fact that one day meets the next, and there is no time between them. The clues which give an event specific position might also have been equivalently written by specifying that the particular value appeared in the same box as one of the day names—for instance $\boxed{mon, freezing}$. However, the five day names which appear in the first clue of Table 9 are not actually required for the puzzle: a string of five empty boxes suffices. The names are included here purely for convenience of reading.

- $\{$ | mon | tue | wed | thu | fri | $\}$

- $\{$ | | fog, mild | , | | fog, mild | | , | fog, mild | | $\}$

- $\{$ | | tired, warm | , | | tired, warm | | , | tired, warm | | $\}$

- $\{$ | | jammed, overcast | , | | jammed, overcast | | , | jammed, overcast | | $\}$

- $\{$ | | cold, little | , | | cold, little | | , | cold, little | | $\}$

- $\{$ | | snow | overcast | , | | snow | overcast | | , | snow | overcast | | $\}$

- $\{$ | | sad, reading | , | | sad, reading | | , | sad, reading | | $\}$

- $\{$ | | rain, printing | , | | rain, printing | | , | rain, printing | | $\}$

- $\{$ | | | average | | | $\}$

- $\{$ | freezing | | | | | $\}$

- $\{$ | | stapling | happy | , | | stapling | happy | | , | stapling | happy | | , | | happy | stapling | , | | happy | stapling | | , | happy | stapling | | $\}$

- $\{$ | | printing | angry | , | | printing | angry | | , | printing | angry | | , | | angry | printing | , | | angry | printing | | , | angry | printing | | $\}$

- $\{$ | | lots, filing | , | | lots, filing | | , | lots, filing | | $\}$

- $\{$ | | shredding, hot | , | | shredding, hot | | , | shredding, hot | | $\}$

- $\{$ | | freezing | clear | , | | freezing | clear | | , | freezing | clear | | , | | clear | freezing | , | | clear | freezing | | , | clear | freezing | | $\}$

Table 9: Temporal Zebra puzzle clues as strings.

Formally, there are five Attributes (Weather, Temperature, Traffic, Tasks, Mood), each of which is a set of Values. The vocabulary $\mathcal{V}$ of the puzzle is the union of the Attributes:

- Weather $= \{rain, clear, fog, snow, overcast\}$

- Temperature $= \{freezing, cold, mild, warm, hot\}$

84

- Traffic $= \{none, little, average, lots, jammed\}$

- Tasks $= \{printing, stapling, reading, filing, shredding\}$

- Mood $= \{happy, angry, sad, tired, curious\}$

The external constraints are formalised as follows:

"Each Value only lasts for one day."

(108) $\forall v \in \mathcal{V} \ (x \in [\![P_v]\!] \implies \forall v' \in \mathcal{V} \ (v' \neq v \Rightarrow x \notin [\![P_{v'}]\!]))$

"Each day only contains one Value for each Attribute."

(109) $\forall A \in Attributes \ (x \in [\![P_v]\!] \land v \in A \implies \forall v' \in A \ (v' \neq v \Rightarrow x \notin [\![P_{v'}]\!]))$

One further constraint is needed due to the nature of superposition allowing for the result of superposing two strings to be longer than either of the input strings:

"There are only 5 days."

(110) $\forall v \in \mathcal{V} \ (x \in [\![P_v]\!] \implies 1 \leq x \leq 5)$

Although these constraints are not encoded as strings[44], since they are modelled using MSO formulas, they are easily implemented through hand-written filters which apply the constraints to the results of any superposition.

Now, superposing all of the languages in Table 9 and taking into account the constraints in (108) to (110), a single result string is generated (each Value is displayed here abbreviated to its first two letters):

(111) | ra, fr, pr, ha | cl, co, li, st, an | fo, mi, av, re, sa | sn, wa, lo, fi, ti | ov, ho, ja, sh |

Finally, superposing this string with languages representing the questions that were asked, a string containing the full "week schedule" is obtained ((112a)), and by taking the reduct

---

[44]In fact, they could be encoded as strings by allowing negated fluents to appear in the vocabulary: normally a fluent v is implicitly understood to be not occurring by its non-appearance in a string component, but an explicit $\neg v$ fluent appearing would disallow v to occur in the same component. These explicitly negated fluents are not currently part of the framework, but could be introduced in the future—see also § 4.2.1, p. 113.

of this string relative to $\{none, curious\}$ the solution to the puzzle can be found ((112b), and superposed with a string of weekday names for ease of reading in (112c)):

- $\{ \boxed{\;\boxed{none}\;}, \boxed{\;\boxed{none}\;\;}, \boxed{\boxed{none}\;} \}$

- $\{ \boxed{\;\boxed{curious}\;}, \boxed{\;\boxed{curious}\;\;}, \boxed{\boxed{curious}\;} \}$

(112a) $\quad s = \boxed{ra, fr, pr, ha, \underline{no}} \; \boxed{cl, co, li, st, an} \; \boxed{fo, mi, av, re, sa} \; \boxed{sn, wa, lo, fi, ti} \; \boxed{ov, ho, ja, sh, \underline{cu}}$

(112b) $\qquad\qquad\qquad \rho_{\{none, curious\}}(s) = \boxed{none} \; \boxed{\;} \; \boxed{\;} \; \boxed{\;} \; \boxed{curious}$

(112c) $\quad \rho_{\{none, curious\}}(s) \; \& \; \boxed{mon} \; \boxed{tue} \; \boxed{wed} \; \boxed{thu} \; \boxed{fri} = \boxed{mon, \underline{none}} \; \boxed{tue} \; \boxed{wed} \; \boxed{thu} \; \boxed{fri, \underline{curious}}$

Thus the answer is that "there is no traffic on the first day of the week", and "I am curious on the last day of the week". This result is reproduced in Table 10 below for the sake of completeness.

	Mon	Tue	Wed	Thu	Fri
Weather	*rain*	*clear*	*fog*	*snow*	*overcast*
Temperature	*freezing*	*cold*	*mild*	*warm*	*hot*
Traffic	*none*	*little*	*average*	*lots*	*jammed*
Task	*printing*	*stapling*	*reading*	*filing*	*shredding*
Mood	*happy*	*angry*	*sad*	*tired*	*curious*

Table 10: Solution to Temporal Zebra puzzle as in (112a).

While the Attributes and Values for this variation of the puzzle are relatively meaningless, the use of strings to represent the constraints given in the clues maps well to using strings to model constraints for scheduling problems such as the job-shop problem (Manne, 1960; Applegate and Cook, 1991), wherein a finite number of resources (agents) are available to complete a set of tasks, and an agent can only complete a single task at a time, and the goal is to create a schedule that minimises the maximum of their completion times given that each task also has a specified order that it must be completed in—for instance, task A may only begin after task B is completed, and must be completed before task C. This is very similar to the constraints described in the variant of the Zebra Puzzle above,

with the additional factor of seeking a string which encapsulates all of the constraints and also has the shortest duration. Another similar problem is the trains example in Durand and Schwer (2008b), reproduced below in Table 11, which features six trains arriving and leaving a busy station. The task is to determine the minimum number of platforms needed, given that a train is not allowed to arrive at the same platform while another train is still there.

There are 6 trains: $\{A, B, C, D, E, F\}$.

A train may not arrive on a platform if another train has not left that platform.

- A, B, and E reach the station at the same time.

- A leaves before B.

- A leaves after or at the same time as C, but before the arrival of D.

- D and F arrive at the same time as B is leaving.

- E and D leave at the same time.

Table 11: Constraints of the Trains example from Durand and Schwer (2008b, p. 3283).

Durand and Schwer (2008b, p. 3298) use a formalism they call S-languages developed based on the concept of S-arrangements (Schwer, 2002), which uses arrangements of subsets of elements with repetitions in a manner which is similar in some ways to the string framework described in § 3.1, using sequences of elements to represent event-like entities, generally described as either instantaneous points, or in terms of their beginning and end points.

The constraints in Table 11 can be represented through the languages in Table 12 below, where each 'fluent' symbol appearing in a string component represents that train being currently at the station. Therefore, if multiple symbols appear in a box, then each of those trains requires a separate platform at the same time.

- { $|\ |A,B,E|A,B|\ |$, $|\ |A,B,E|E|\ |$, $|\ |A,B,E|A,B|B|\ |$,

  $|\ |A,B,E|A,E|A|\ |$, $|\ |A,B,E|A,E|E|\ |$, $|\ |A,B,E|A,B|A|\ |$,

  $|\ |A,B,E|B|\ |$, $|\ |A,B,E|A,E|\ |$, $|\ |A,B,E|A|\ |$,

  $|\ |A,B,E|B,E|B|\ |$, $|\ |A,B,E|\ |$, $|\ |A,B,E|B,E|\ |$, $|\ |A,B,E|B,E|E|\ |$ }

- { $|\ |A|\ |B|\ |$, $|\ |A|B|\ |$, $|\ |A|A,B|B|\ |$, $|\ |A,B|B|\ |$, $|\ |B|A,B|B|\ |$ }

- { $|\ |A,C|A|D|\ |$, $|\ |C|\ |A|D|\ |$, $|\ |C|\ |A|\ |D|\ |$, $|\ |C|A,C|\ |D|\ |$, $|\ |C|\ |A|D|\ |$,

  $|\ |C|A,C|A|\ |D|\ |$, $|\ |A,C|\ |D|\ |$, $|\ |A,C|A|\ |D|\ |$, $|\ |A|A,C|A|\ |D|\ |$,

  $|\ |C|\ |A|\ |D|\ |$, $|\ |A|A,C|\ |D|\ |$, $|\ |A,C|D|\ |$, $|\ |A|A,C|A|D|\ |$,

  $|\ |C|A,C|A|D|\ |$, $|\ |A|A,C|D|\ |$, $|\ |C|A,C|D|\ |$ }

- { $|\ |B|B,D,F|D,F|D|\ |$, $|\ |B|B,D,F|D,F|F|\ |$, $|\ |B|B,D,F|D,F|\ |$ }

- { $|\ |E|D,E|\ |$, $|\ |D|E,D|\ |$, $|\ |D,E|\ |$ }

Table 12: Train scheduling problem constraints as languages of strings.

Superposing these constraints together produces a language of 48 strings, each of which represents a different possible way of satisfying all the scheduling constraints together. In order to determine the minimum number of resources required—in this problem, the number of platforms in the train station—it is simply a matter of finding the string(s) whose largest component is smaller than the largest component of all other strings. In this case, since B and E arrive together, and D and F arrive before either of them have finished leaving, all 48 strings feature a component $\boxed{B,D,E,F}$—for instance, $|\ |C|A,B,E|B,D,E,F|D,E,F|\ |$—and thus the minimum number of platforms is the cardinality of that component: four.

Additionally, despite strings not explicitly denoting the durations of events, it is possible to find the timeline which is the least durative by finding the shortest string(s) which uses the maximum possible number of resources. In the case of the trains example, this is $|\ |A,B,C,E|B,D,E,F|D,E,F|\ |$. To justify this, consider two events a and b which

88

have durations τ_a and τ_b, respectively, such that the a lasts longer than b, $\tau_a > \tau_b$. In this scenario, four relations are impossible, namely a equals b, a during b, a starts b, and a finishes b—all of these imply that b has the same or shorter duration than a. Of the remaining nine, the duration of the string representing the relations are as follows in Table 13[45]:
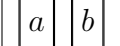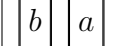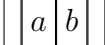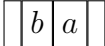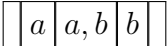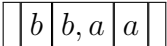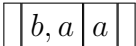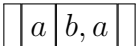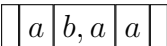
$\bullet$	$\mathcal{S}_\bullet$	$\tau_{\mathcal{S}_\bullet}$
a before b	$\|\|\,a\,\|\ \|\,b\,\|\|$	$\tau_{\mathcal{S}_\bullet} > \tau_a + \tau_b$
a after b	$\|\|\,b\,\|\ \|\,a\,\|\|$	$\tau_{\mathcal{S}_\bullet} > \tau_a + \tau_b$
a meets b	$\|\|\,a\,\|\,b\,\|\|$	$\tau_{\mathcal{S}_\bullet} = \tau_a + \tau_b$
a met by b	$\|\|\,b\,\|\,a\,\|\|$	$\tau_{\mathcal{S}_\bullet} = \tau_a + \tau_b$
a overlaps b	$\|\|\,a\,\|\,a,b\,\|\,b\,\|\|$	$\tau_a + \tau_b > \tau_{\mathcal{S}_\bullet} > \tau_a$
a overlapped by b	$\|\|\,b\,\|\,b,a\,\|\,a\,\|\|$	$\tau_a + \tau_b > \tau_{\mathcal{S}_\bullet} > \tau_a$
a started by b	$\|\|\,b,a\,\|\,a\,\|\|$	$\tau_{\mathcal{S}_\bullet} = \tau_a$
a finished by b	$\|\|\,a\,\|\,b,a\,\|\|$	$\tau_{\mathcal{S}_\bullet} = \tau_a$
a contains b	$\|\|\,a\,\|\,b,a\,\|\,a\,\|\|$	$\tau_{\mathcal{S}_\bullet} = \tau_a$

Table 13: Durations of relations between a and b.

As can be seen, a string cannot have a duration shorter than it's longest lasting event, but it may have a duration longer than the sum of all its events. The strings which maximise for component size (overlaps, overlapped by, started by, finished by, contains) all have a duration shorter than those which do not, of which the two shorter strings (meets, met by) have durations shorter than the two longer (before, after). The shortest strings of the maximising five (started by, finished by) have durations equal to the duration of the longest event, a. The relations overlaps, overlapped by, and contains all contain a component of the same maximum size, and are the same length, so in the scenario where multiple strings are found to be equally the shortest maximising for component size, a further step would be required to determine the types of relations found in the string— a string with more contains relations should be chosen as being less durative than one with more overlaps or overlapped by relations. Finally, if instead a and b have equal

---

[45]Disregarding the durations of the empty bounding boxes, which have effectively infinite duration.

durations, then only one relation is possible between them—a equals b, $\boxed{\;\boxed{a,b}\;}$—which also maximises for component size, and has the shortest string length of all the Allen relations.

This chapter has described a framework which uses strings as sequences of sets of temporal entities—times and events—as part of an approach to temporal semantics known as finite-state temporality. Strings of this type are interpretable as finite models of Monadic Second-Order logic, and so due to an equivalence with regular languages, this allows for strings to be recognised by finite-state automata and systems which use them. Several operations are described which can be used to put these strings to work—in particular, the superposition operation allows for strings and sets of strings to be combined, such that all of the temporal data contained therein is incorporated into new strings, and the projection operation allows for determining the linear ordering and relation between a specified subset of times and events in a string.

Some applications of these operations were discussed, including the ability to derive timelines from annotated narratives using Allen's interval relations along with superposition and projection between strings. These timelines can be used to provide insight into the overall temporal structure of the text, as well as check for inconsistencies. Additionally, strings can be used as a tool for some kinds of temporal constraint satisfaction, such as in scheduling problems, as the superposition operation can be used to combine strings which model temporal constraints, producing sets of one or more strings in which all the constraints hold successfully, or an empty set if the constraints cannot hold simultaneously.

The next chapter will describe the procedures by which the use of these strings can be integrated with systems which produce semantic annotation for temporal data—primarily focusing on the documents of the TimeBank corpus, a dataset of newswire texts marked up with TimeML—and additionally how other resources can be used to augment the work that strings can do, such as being able to make inferences about what relations must be added to a knowledge base in order to be able to draw new conclusions.

4 Methods

This chapter details the use of strings and various string operations, described in § 3, as an augmentative tool for temporal annotation. Using TimeML-annotated documents as a source, strings can be derived from the temporal links, and used to bring out other, implicit relations. However, due to the nature of discourse, the temporal data is often vague or incomplete, so some methods are presented for dealing with this.

4.1 Extracting Strings from Annotated Text

In order to extract temporal strings like those described in § 3.1 from a text, the text must first be marked up using the TimeML (TimeML Working Group, 2005) annotation schema, as this will provide the starting point which indicates which temporal entities are relevant in the text[46]. These annotated times and events, along with the given relations between them, will directly lead to the creation of the initial set of strings, which will serve as a 'seed' for the generation of all other possible strings.

Despite TimeML's successor schema, ISO-TimeML (Pustejovsky et al., 2010), being adopted by the International Organization for Standardization as the standard (ISO 24617-1:2012, 2012) for semantically annotating temporal data in a text, this work will only focus on the older TimeML version 1.2.1 schema (TimeML Working Group, 2005; Saurí et al., 2006). The primary source of data is the TimeBank corpus (Pustejovsky et al., 2006) which is one of the largest available collections of documents which are annotated with any version of TimeML. At the time of writing, the latest release of the corpus is version 1.2, which uses the TimeML version 1.2.1 schema. This corpus is generally seen as a gold standard for any experiments relating to TimeML due to both its size and quality, as it contains 183 documents which were each manually annotated in multiple phases: initially, a pre-processing step wherein five annotators worked on the corpus, regularly meeting to discuss their decisions and improve inter-annotator agreement; the last phase described involved four annotators who "intimately familiar with the latest specifi-

---

[46]Although, if used in an interactive environment such as the tool presented in § 5, it is possible to start instead from plain text.

cation" (TimeML Working Group, 2005), which at the time was version 1.2.1, as already mentioned. The documents are amalgamated from a variety of newswire sources, and contain just over 61,000 non-punctuation tokens. The inter-annotator agreement scores (as the average of precision and recall) for the various tag types are given—for a subset of the corpus—below in Figure 13, reproduced from TimeML Working Group (2005), and it is noted that the low score for the `<TLINK>` tag is due to the large number of event and/or time pairs that may be related to each other, which averages approximately $\frac{(7940+1414)^2}{183} \approx 2613$ pairs per document.

Tag	Exact agreement	Partial agreement
TIMEX3	0.83	0.96
SIGNAL	0.77	0.77
EVENT	0.78	0.81
ALINK	0.81	-
SLINK	0.85	-
TLINK	0.55	-

Figure 13: Inter-annotator agreement for tags in TimeBank 1.2
(from TimeML Working Group, 2005).

The agreement scores are also given for some of the tag attributes, using again the average of precision and recall, and also the kappa scores, reproduced in Figure 14. TimeML Working Group (2005) does note, though, that due to the size of the subset of documents for which inter-annotator agreement was tested, some of these values are not hugely significant, in particular "the inter-annotator agreement numbers for ALINKs as well as the polarity feature for events are not reliable."

Tag and attribute	$\frac{precision+recall}{2}$	Kappa
TIMEX3.type	1.00	1.00
TIMEX3.value	0.90	0.89
TIMEX3.temporalFunction	0.95	0.87
TIMEX3.mod	0.95	0.73
EVENT.class	0.77	0.67
MAKEINSTANCE.pos	0.99	0.96
MAKEINSTANCE.tense	0.96	0.93
MAKEINSTANCE.aspect	1.00	1.00
MAKEINSTANCE.polarity	1.00	1.00
MAKEINSTANCE.modality	1.00	1.00
ALINK.relType	0.80	0.63
SLINK.relType	0.98	0.96
TLINK.relType	0.77	0.71

Figure 14: Inter-annotator agreement for attributes in TimeBank 1.2
(from TimeML Working Group, 2005).

Of most significance to the present work is the favourable score for the `<TLINK>` tags,
from which strings can be derived.

4.1.1 Linking the TLINKs

TimeML primarily uses `<TLINK>` tags to annotate relations between marked up events
and times. These serve as the basis for initial string creation, as they provide information
about not only the two intervals that are being related, but also the relation between them.
The set of available relations is specific to TimeML—given along with each relation's
count and proportion of the TimeBank 1.2 corpus in Table 14—but has its roots in the
set of Allen Relations—see the translation between the sets in Figure 7, p. 24. Since
Allen's set of relations are what will be used to create strings, the `<ALINK>` and `<SLINK>`
tags are ignored for the time being, as are the `<SIGNAL>` tags. The intervals which are
being related will be some pairing of event instance IDs and/or time IDs, which point to

<MAKEINSTANCE> and <TIMEX3> tags, respectively, found elsewhere in the document.

Relation	Count	Proportion
BEFORE	1408	21.94%
AFTER	897	13.98%
IBEFORE	34	0.53%
IAFTER	39	0.61%
DURING	302	4.71%
DURING_INV	1	0.02%
IS_INCLUDED	1357	21.14%
INCLUDES	582	9.07%
BEGINS	61	0.95%
BEGUN_BY	70	1.09%
ENDS	76	1.18%
ENDED_BY	177	2.76%
SIMULTANEOUS	671	10.45%
IDENTITY	743	11.58%
Total	6418	100%

Table 14: The counts and proportions of each TimeML relation in the TimeBank 1.2 corpus.

The decision is made to fold the <MAKEINSTANCE> tags into the <EVENT> tags which they reference for three reasons: first, this is one of the changes made in ISO-TimeML (Pustejovsky et al., 2010), which considers the <EVENT> tags as explicitly event instances; second, there are 7,940 <MAKEINSTANCE> tags in the corpus, and 7,935 <EVENT> tags, meaning there are just 5 instances which point at an event already pointed at by another instance—this is just 0.06% of the instances, so merging them should not have a significant adverse impact; finally, for simplicity's sake, as it makes it easier to track the text in the <EVENT> tag along with all of the attributes from the <MAKEINSTANCE> tag, like the part of speech, tense, and aspect. While mapping the instances onto the events, it is also necessary to update the <TLINK> tags, so that any eventInstanceID and relatedToEventInstance attributes are converted to eventID and relatedToEvent attributes, and also updating

their values as appropriate.

After folding the tags for events together, a translation is built, mapping from a `<TLINK>` tag to string. The `relType` attribute is extracted from the tag, and converted to an Allen relation by looking up its corresponding relation in Figure 7, p. 24. Then, extracting the IDs for the pair of intervals to be related from the `<TLINK>` tag's `eventID` or `timeID` (as $E1$) and `relatedToEvent` or `relatedToTime` (as $E2$) attributes, a string can be constructed using Table 7, p. 78, replacing a and b in that table with $E1$ and $E2$.

For a minimal example, assume the sentence in (113a) is marked up as in (113b), omitting any attributes not directly relevant to this example.

(113a) "John ate dinner, and then washed the dishes."

(113b) `John <EVENT eid="e1">ate</EVENT> dinner,`

`and then <EVENT eid="e2">washed</EVENT> the dishes.`

`<TLINK eventID="e1" relatedToEvent="e2" relType="BEFORE" />`

(113c) `BEFORE` $\equiv$ $before \in \mathcal{AR}$, $E1 = $ `e1`, $E2 = $ `e2`

(113d) $\boxed{\boxed{e1}\boxed{e2}}$

The relation and two event IDs are extracted from the `<TLINK>` tag in (113c) (where $\mathcal{AR}$ is the set of Allen's interval relations, as in (102), p. 78), which are then arranged according to Table 7, p. 78 to produce the string in (113d).

After mapping from the `<TLINK>` tags to strings, the next step is to start building towards a timeline for the document, by using the vocabulary-constrained superposition technique described in §3.1.4, p. 71. This operation will combine the temporal data in pairs of strings, producing a set of output strings which are constrained by the inputs. If the result of a superposition is the empty set, then the input strings were inconsistent with one another—that is, the relations between the fluents in one of the input strings contradicted a relation between those same fluents in the other input. For example, if through some oversight during the annotation process, three events are related by `<TLINK>` tags as follows:

(114a) `<TLINK eventID="e1" relatedToEvent="e2" relType="BEFORE" />`

(114b) `<TLINK eventID="e2" relatedToEvent="e3" relType="BEFORE" />`

(114c) `<TLINK eventID="e1" relatedToEvent="e3" relType="IBEFORE" />`

which produces the following three strings, respectively:

(115a) | | $e1$ | | $e2$ | |

(115b) | | $e2$ | | $e3$ | |

(115c) | | $e1$ | $e3$ | |

Attempting to superpose (115a) and (115b) produces a language containing one string—| | $e1$ | | $e2$ | | $e3$ | |—however, when adding the third input string (115c) to the superposition will result in the empty set. The reason why should be clear from the transitivities of Allen's relations—see Table 1, p. 11—as $e1 < e2$ and $e2 < e3$ gives that $e1 < e3$ should be found, which is the result of superposing the first two strings, but this is contradicted by the $e1me3$ relation in (115c). By ejecting inconsistencies like this, it is ensured that whatever representation is portrayed to a user is at the very least consistent.

In general, combining the `<TLINK>` tags by superposing their string-translations allows for connections to be built between events and times that were not already explicitly related by the annotation. If every time and event in the document is or becomes related to every other time and event, then the *temporal closure* has been calculated for the document—see § 2.1.1. For an annotator manually marking up a text with the TimeML schema to give enough detail for the temporal closure to be calculated unassisted would be incredibly difficult, as for N events and/or times in a document there would need to be $\frac{N^2-N}{2}$ unique `<TLINK>` tags in order to label the relation between each pair, which increases rapidly: at $N = 10$ there are 45 relations, at $N = 50$ there are 1225 relations— while some documents of the TimeBank corpus have many less and several have more, the average document has $\frac{7940+1414}{183} \approx 51$ events and/or times. This quickly becomes unwieldy for a human to annotate, especially given that discourse is often vague in the sense that there may not be a clear or even possible way to determine some relations with any kind

of precision. For instance, given the following sentence (116), it is clear that both of the leaving events occur after the crash event, but it is not obvious how to label the relation between the two leaving events[47].

(116) "After the crash, the police left and the paramedics left."

As such, it should not generally be expected that the temporal closure will already have been computed for any given document—however, if it is computed or is computable, then superposing the strings derived from the document's `<TLINK>` tags will ultimately produce a language containing a single string which models the timeline or temporal structure of the document: its vocabulary will be the set of all the events and times in the document[48], and it will specify the linear ordering and inter-relations of all of these times and events within the context of the document's narrative.

As exemplified in (116), it should be noted that even with the most diligent of annotators, it is still possible that the temporal closure is not immediately computable, as several combinations of the `<TLINK>` relations produce a disjunction of relations which are subsets of the set of Allen Relations according to the transitivities in Table 1, p. 11. In these cases, the superposition of the strings derived from the `<TLINK>` tags will produce a language with a number of strings, each representing one possibility from the disjunction of relations. Recall that superposing a pair of languages with each other produces a new language which contains the results of superposing every string from the first language with every string in the second language—this can quickly lead to an explosion in the number of strings that are generated if the relations between the events do not tightly constrain the disjunctions which arise. Take, for example, a document which has tagged within it its document creation time $t0$ and three events $e1, e2, e3$, but the only relations given by the `<TLINK>` tags is that all three of the events occur before the document creation time, and there are no other constraints given on the relations between the three. In

---

[47] Although, there have been studies which link the order that events appear in the text of a narrative with their temporal ordering, showing that in this example there would be a tendency to assume the police left ahead of the paramedics since they were mentioned first (Ohtsuka and Brewer, 1992).

[48] At least, all of those which were included in at least one TLINK relation. If a time or event was tagged in the text but was not marked up as being related to any other time or event, then there is no way to constrain its relation, and it is effectively an isolated temporal entity, disconnected from the narrative.

this case, vocabulary-constrained superposition will produce a language with 409 strings in it, and if there were four events instead of three, this would already be producing 23,917 strings[49], each of which is possibly the true interpretation of the sequence according to all that is known about the document. Obviously this is unwieldy, and in fact for this example superposing the strings like this is no better than guessing the correct linear ordering of the events at random, since superposition is just giving every single possibility. Derczynski (2016) notes that there needs to be a careful balance between being exact in terms of presenting possible relations, and not letting this set of possible relations grow too large. Assuming for the moment a non-interactive environment where there is no new information forthcoming that can constrain the relations between the events further, there needs to be a way to treat situations like this where the data that can be derived from an annotated text contains some amount of incompleteness, and the temporal closure cannot be computed from just the given knowledge.

4.1.2 Handling Incomplete Data

When dealing with any temporal information, there is often a lack of specificity that means the temporal closure—where all intervals in the knowledge base are (consistently) related by exactly one of the set of Allen Relations—for the given document's set of intervals cannot be calculated. For example, take the sentences in (117):

(117) "Aideen was singing until dinner was served. Brian called Aideen after she had sung."

Assume that for (117), we have an annotation available which provides two relations between the three events[50], with $e1$ = 'singing', $e2$ = 'served', and $e3$ = 'called'. The relations are that $e1$ meets $e2$, and $e3$ is after $e1$, which are translated to strings as described in the previous section and superposed in (118), producing a language with five

---

[49]This number increases dramatically quickly with additional intervals, according to the sequence in Schwer (2000).

[50]In fact, this would be annotated as having a fourth event, 'sung', which has the `IDENTITY` relation with 'singing'. The example has been simplified to omit this, but it does not ultimately impact the superposition calculation given in (118).

strings in it:

$$(118) \qquad \boxed{\;\boxed{e1}\,\boxed{e2}\;} \;\&_{vc}\; \boxed{\;\boxed{e1}\;\boxed{e3}\;} \;=\; \{\; \boxed{\;\boxed{e1}\,\boxed{e2}\,\boxed{e2,e3}\,\boxed{e2}\;}\,,\; \boxed{\;\boxed{e1}\,\boxed{e2}\,\boxed{e2,e3}\;}\,,$$

$$\boxed{\;\boxed{e1}\,\boxed{e2}\;\boxed{e3}\;}\,,\; \boxed{\;\boxed{e1}\,\boxed{e2}\,\boxed{e2,e3}\,\boxed{e3}\;}\,,\; \boxed{\;\boxed{e1}\,\boxed{e2}\,\boxed{e3}\;}\;\}$$

The result of the superposition in (118) contains five strings which are each equally valid interpretations of the sequence of the three events, assuming there is no further data with which to constrain the disjunction of strings. While in all five, due to the nature of vocabulary-constrained superposition preserving projections—see (95), p. 71—the relation between $e1$ and $e2$ is always 'meets', and the relation between $e3$ and $e1$ is always 'after', the relation between $e2$ and $e3$ is a disjunction of options. This is seen more clearly by taking the block compressed reduct of each string s in the result of (118) relative to the set $\{e2, e3\}$, as in Table 15 below:

$\mathrm{bc}(\rho_{\{e2,e3\}}(s))$	Relation
$\boxed{\;\boxed{e2}\,\boxed{e3,e2}\;}$	$e2$ finished by $e3$
$\boxed{\;\boxed{e2}\,\boxed{e3,e2}\,\boxed{e2}\;}$	$e2$ contains $e3$
$\boxed{\;\boxed{e2}\,\boxed{e3}\;}$	$e2$ meets $e3$
$\boxed{\;\boxed{e2}\;\boxed{e3}\;}$	$e2$ before $e3$
$\boxed{\;\boxed{e2}\,\boxed{e2,e3}\,\boxed{e3}\;}$	$e2$ overlaps $e3$

Table 15: Relations arising from the superposition in (118).

One thing that can be seen in Table 15 is that in each string, the leftmost non-empty component always contains just $e2$, implying that $e2$ has begun before $e3$ has begun, although it is not clear which of the two events ends first. In fact, this is exactly the case, and Freksa (1992) uses these kind of disjunctions that can arise as partial justification for their set of 31 semi-interval relations, which are a superset of the Allen Relations. To use semi-intervals, their must be a change in granularity so as to consider the beginnings and endings of intervals as primitive, rather than intervals themselves. In contrast to, for example, Durand and Schwer (2008a) or Fernando (2018), the beginnings and endings of intervals are themselves intervals, rather than instantaneous points. Since they are

intervals, they can also be decomposed into beginnings and endings, depending on how much 'zoom' is desired. The major advantage of using beginnings and endings as primitive is that it allows for underspecification of an interval, so that either the beginning or the ending may be omitted from consideration if necessary. This perfectly captures the scenario in Table 15, where the beginning of $e2$ precedes the beginning of $e3$, but the order of their endings is unclear—this corresponds to the Freksa relation labelled 'older', and thus $e2$ is older than $e3$.

According to Freksa (1992, p. 202) there is a "cognitive awkwardness" in that, in the case of incomplete information with Allen intervals, the representation becomes more complex the less one knows (as is the case with the example in Table 15), since that knowledge is represented as a disjunction of what *might be* true. He states that, from a cognitive standpoint, the knowledge would be preferably represented "more directly and in such a way that less knowledge corresponds to a simpler representation than more knowledge does" (Freksa, 1992, p. 202). To address this, Freksa describes the 13 Allen relations in terms of constraints between semi-intervals, as well as 18 more semi-interval relations for situations when it is not (necessarily) known how an event's beginning and ending relate to those of another event (Freksa, 1992, p. 219).

The set of Freksa (1992)'s relations is given in Table 16, showing the constraints which each is associated with, where $l(a)$ and $r(a)$ are the beginning and ending of some interval a, and similarly $l(b)$ and $r(b)$ are the beginning and ending of some interval b. Allen's relations, which are a subset of Freksa's are omitted from the table, as is the 'unknown' relation, which does not impose any constraints on the relation of the intervals nor their beginnings and endings.

The relations are given again in Table 17, this time in terms of disjunctions of Allen (1983)'s relations, again excluding Allen's relations and the 'unknown' relation, which is a disjunction of all 13 Allen Relations.

It's worth noting here that Freksa (1992)'s primary motivation in the deisgn of these relations was to appeal to the concept of what they called "conceptual neighbourhoods", whereby a pair of interval relations are concpetual neighbours if "they can be directly

transformed into one another by continuously deforming (i.e. shortening, lengthening, moving) the events (in a topological sense)" (Freksa, 1992, p. 206). Thus all of the disjunctions of Allen's relations contain only relations which are conceptual neighbours in this way—for example, it's a shorter cognitive leap to adjust from one event being 'before' another to the events having the 'meets' relation, than it is to go from 'before' to 'after'. A subset of these concpetual neighbourhoods can be found as the disjunctions which appear in the transitivity table of Allen (1983, p. 836)—see Table 1, p. 11.

Freksa label	Constraints
older	$l(a) < l(b)$
younger	$l(a) > l(b)$
head to head	$l(a) = l(b)$
tail to tail	$r(a) = r(b)$
survives	$r(a) > r(b)$
survived by	$r(a) < r(b)$
precedes	$r(a) \leq l(b)$
succeeds	$l(a) \geq r(b)$
born before death	$l(a) < r(b)$
died after birth	$r(a) > l(b)$
contemporary	$l(a) < r(b),\ r(a) > l(b)$
older survived by	$l(a) < l(b),\ r(a) < r(b)$
younger survives	$l(a) > l(b),\ r(a) > r(b)$
older contemporary	$l(a) < l(b),\ r(a) > l(b)$
younger contemporary	$l(a) > l(b),\ l(a) < r(b)$
survived by contemporary	$r(a) > l(b),\ r(a) < r(b)$
surviving contemporary	$l(a) < r(b),\ r(a) > r(b)$

Table 16: Freksa (1992)'s relations described in terms of the constraints they impose on the beginnings and endings of a pair of intervals.

Freksa label	Disjunction of Allen's relations
older	before, meets, finished by, contains, overlaps
younger	after, met by, finishes, during, overlapped by
head to head	starts, started by, equals
tail to tail	finishes, finished by, equals
survives	after, met by, started by, contains, overlapped by
survived by	before, meets, starts, during, overlaps
precedes	before, meets
succeeds	after, met by
contemporary	starts, started by, finishes, finished by, during, contains, overlaps, overlapped by, equals
born before death	before, meets, starts, started by, finishes, finished by, during, contains, overlaps, overlapped by, equals
died after birth	starts, started by, finishes, finished by, during, contains, overlaps, overlapped by, equals, after, met by
older survived by	before, meets, overlaps
younger survives	after, met by, overlapped by
older contemporary	finished by, contains, overlaps
younger contemporary	finishes, during, overlapped by
survived by contemporary	during, starts, overlaps
surviving contemporary	contains, started by, overlapped by

Table 17: Freksa (1992)'s relations described in terms of disjunctions of Allen (1983)'s relations.

In order to take advantage of these relations in strings, first there must be a translation from intervals to semi-intervals, which is described here. First, two predicates are defined:

(119) $\qquad \alpha_v(x) \; := \; \exists y \; (x < y \; \wedge \; y \in [\![P_v]\!] \; \wedge \; \forall z \; (z < y \Longrightarrow z \notin [\![P_v]\!]))$

(120) $\qquad \omega_v(x) \; := \; \exists y \; (y < x \; \wedge \; y \in [\![P_v]\!] \; \wedge \; \forall z \; (y < z \Longrightarrow z \notin [\![P_v]\!]))$

(119) and (120) define (for some fluent $v \in \mathcal{V}$) the appearance of $\alpha(v)$ within a box[51] as representing a negation of the fluent v conjoined with a formula stating that v will be true in some subsequent box; similarly, $\omega(v)$ represents a negation of the fluent v conjoined with a formula stating that v was true in some previous box.

A string s may be translated to one using semi-intervals by placing $\alpha(v)$ in every box preceding one in which a fluent v appears, and $\omega(v)$ in every box succeeding it, for each $v \in \mathcal{V}_s$ (Woods and Fernando, 2018).

For example, a string $s = \boxed{}\,\boxed{a}\,\boxed{b}\,\boxed{}$ becomes $semi(s)$ in (121), in which each of the fluents originally appearing in the string s have also been removed. It's worth highlighting that this representation is possible since, under the semi-interval interpretation, beginnings and endings of intervals are themselves treated as intervals, which is why $\alpha(b)$ and $\omega(a)$ are able to appear in multiple components of the string:

(121)
$$semi(s) = \boxed{\alpha(a), \alpha(b)}\,\boxed{\alpha(b)}\,\boxed{\omega(a)}\,\boxed{\omega(a), \omega(b)}$$

Allowing non-atomic formulas such as these inside the string components does pose a risk of trivialising the work done by superposition, so care must be taken here (Woods and Fernando, 2018). It should be noticed that strings which use semi-intervals do not (necessarily) feature empty boxes at each end. This is due to the fact that a beginning or ending is only bounded on one side each—if $\alpha(a)$ is true at some moment, stating that the interval a has not yet begun, then it must also be true at every moment before that, since otherwise would imply a having occurred already; similarly for $\omega(a)$ holding at some moment and every moment after it. $\alpha(a)$ may be thought of as 'pre-a', and $\omega(a)$ as 'post-a' (Woods and Fernando, 2018).

This mechanism allows for partially known information to be represented using strings. For example, the string $\boxed{\alpha(a), \alpha(b)}\,\boxed{}$ represents the knowledge that the events labelled a and b both begin at the same moment, without stating anything about when they each finish—they may end simultaneously, a may finish before b, or b may finish before a. Which of these states is true is unknown without further data. Going back to the

---
[51]Note that it is convenient to write $\alpha(v)$ for α_v when using the box-notation.

example in Table 15, translating these five strings to use semi-intervals in Table 18 makes the relationship amongst them clearer.

String	Relation
$\boxed{\alpha(e2),\alpha(e3)}\ \boxed{\alpha(e3)}\ \boxed{\ }\ \boxed{\omega(e2),\omega(e3)}$	$e2$ finished by $e3$
$\boxed{\alpha(e2),\alpha(e3)}\ \boxed{\alpha(e3)}\ \boxed{\ }\ \boxed{\omega(e3)}\ \boxed{\omega(e2),\omega(e3)}$	$e2$ contains $e3$
$\boxed{\alpha(e2),\alpha(e3)}\ \boxed{\alpha(e3)}\ \boxed{\omega(e2)}\ \boxed{\omega(e2),\omega(e3)}$	$e2$ meets $e3$
$\boxed{\alpha(e2),\alpha(e3)}\ \boxed{\alpha(e3)}\ \boxed{\alpha(e3),\omega(e2)}\ \boxed{\omega(e2)}\ \boxed{\omega(e2),\omega(e3)}$	$e2$ before $e3$
$\boxed{\alpha(e2),\alpha(e3)}\ \boxed{\alpha(e3)}\ \boxed{\ }\ \boxed{\omega(e2)}\ \boxed{\omega(e2),\omega(e3)}$	$e2$ overlaps $e3$

Table 18: Translating strings corresponding to the Freksa 'older' relation from Table 15 to use semi-intervals.

Leaning on this example a final time, the strings in Table 18 are subjected to block compressed reduct relative to the set $\{\alpha(e2),\alpha(e3)\}$ in Table 19.

$\mathrm{bc}(\rho_{\{\alpha(e2),\alpha(e3)\}}(s))$	Relation
$\boxed{\alpha(e2),\alpha(e3)}\ \boxed{\alpha(e3)}\ \boxed{\ }$	$e2$ finished by $e3$
$\boxed{\alpha(e2),\alpha(e3)}\ \boxed{\alpha(e3)}\ \boxed{\ }$	$e2$ contains $e3$
$\boxed{\alpha(e2),\alpha(e3)}\ \boxed{\alpha(e3)}\ \boxed{\ }$	$e2$ meets $e3$
$\boxed{\alpha(e2),\alpha(e3)}\ \boxed{\alpha(e3)}\ \boxed{\ }$	$e2$ before $e3$
$\boxed{\alpha(e2),\alpha(e3)}\ \boxed{\alpha(e3)}\ \boxed{\ }$	$e2$ overlaps $e3$

Table 19: Block compressed reduct on Table 18.

Each of the five strings in Table 18 projects (using a block compressed reduct) to the string $\boxed{\alpha(e2),\alpha(e3)}\ \boxed{\alpha(e3)}\ \boxed{\ }$, meaning the relation $e2$ is older than $e3$ can be represented using just this one string, instead of a disjunction of five. This does raise the question, however, of whether the tradeoffs are worth it: a great reduction in the cardinality of the timeline set can be achieved, but at the cost of using a more complex vocabulary and reducing the precision of the known information by underspecification.

Out of the 18 non-Allen[52] Freksa relations, only 11 can be described using a single string without further complicating the vocabulary which may appear within a component (box) of a string. Since all of these new relations are equivalent to disjunctions

---

[52]The 13 Allen relations can be represented as single strings using beginnings and endings without underspecification.

of Allen relations, it follows that it may be acceptable to include disjunctions of semi-intervals inside a string component, such as $\boxed{\alpha(a) \vee \alpha(b)}$, which is interpreted as one might expect: either $\alpha(a)$ appears in the component, or $\alpha(b)$ does, or they both do. This allowance admits a further five Freksa relations to be described as single strings. Some relations may be represented in several ways, including the 'unknown' relation, which may trivially be described using a single string since it encompasses a disjunction of all 13 Allen relations, and is formed by a simple disjunction of all possible semi-inteval symbols: $\boxed{\alpha(a) \vee \alpha(b) \vee \omega(a) \vee \omega(b) \vee \epsilon}$, or even more simply, with a string consisting of just a single empty box: $\boxed{}$.

Table 20 below shows the 18 Freksa relations and the string which they will project to for some interval events a and b. The disjunctions of Allen Relations each Freksa relations corresponds to are given again for convenience, using mnemonic labels[53].

Freksa	Allen	string
unknown	b, bi, m, mi, s, si, f, fi, d, di, o, oi, e	$\boxed{}$
older	b, m, o, di, fi	$\boxed{\alpha(a), \alpha(b)}\,\boxed{\alpha(b)}\,\boxed{}$
younger	bi, mi, oi, d, f	$\boxed{\alpha(a), \alpha(b)}\,\boxed{\alpha(a)}\,\boxed{}$
head to head	s, si, e	$\boxed{\alpha(a), \alpha(b)}\,\boxed{}$
tail to tail	f, fi, e	$\boxed{}\,\boxed{\omega(a), \omega(b)}$
survived by	b, m, s, d, o	$\boxed{}\,\boxed{\omega(a)}\,\boxed{\omega(a), \omega(b)}$
survives	bi, mi, si, di, oi	$\boxed{}\,\boxed{\omega(b)}\,\boxed{\omega(a), \omega(b)}$
born before death	b, m, s, si, f, fi, d, di, o, oi, e	$\boxed{\alpha(a)}\,\boxed{}\,\boxed{\omega(b)}$
died after birth	bi, mi, s, si, f, fi, d, di, o, oi, e	$\boxed{\alpha(b)}\,\boxed{}\,\boxed{\omega(a)}$
precedes	b, m	$\boxed{\alpha(b) \vee \omega(a)}$
succeeds	bi, mi	$\boxed{\alpha(a) \vee \omega(b)}$
contemporary	s, si, f, fi, d, di, o, oi, e	$\boxed{\alpha(a) \vee \alpha(b)}\,\boxed{}\,\boxed{\omega(a) \vee \omega(b)}$
older contemporary	o, fi, di	$\boxed{\alpha(a), \alpha(b)}\,\boxed{\alpha(b)}\,\boxed{}\,\boxed{\omega(a) \vee \omega(b)}$
younger contemporary	oi, f, d	$\boxed{\alpha(a), \alpha(b)}\,\boxed{\alpha(a)}\,\boxed{}\,\boxed{\omega(a) \vee \omega(b)}$
surviving contemporary	di, si, oi	$\boxed{\alpha(a)}\,\boxed{}\,\boxed{\omega(b)}\,\boxed{\omega(a), \omega(b)}$
survived by contemporary	d, s, o	$\boxed{\alpha(b)}\,\boxed{}\,\boxed{\omega(a)}\,\boxed{\omega(a), \omega(b)}$
older and survived by	b, m, o	...
younger and survives	bi, mi, oi	...

Table 20: The Freksa relations and the strings they project to.

[53]Using b, bi, and e for $<$, $>$, and $=$, respectively.

The last two relations in Table 20, shown again in Table 21, cannot be represented using a single string, and instead must use conjunctions of pairs of strings. What this means is that for the other relations, it can be determined whether some string of semi-intervals s entails that relation if and only if s projects to the string given in Table 20 for that relation—see also (104), p. 78—for these last two, s will only entail these relations iff it projects to both of the strings given in the conjunction. This may be a small complication, but it is important to address.

Freksa	Allen	strings		
older and survived by	b, m, o	$\boxed{\alpha(a),\alpha(b)\,|\,\alpha(b)\,|\,}$ $\wedge$	$\boxed{\,|\,\omega(a)\,|\,\omega(a),\omega(b)}$	
younger and survives	bi, mi, oi	$\boxed{\alpha(a),\alpha(b)\,|\,\alpha(a)\,|\,}$ $\wedge$	$\boxed{\,|\,\omega(b)\,|\,\omega(a),\omega(b)}$	

Table 21: The remaining Freksa relations and the strings they project to.

Now there is a way of representing (almost) all of the disjunctions that may arise from superpositions of strings representing relations between times and events derived from `<TLINK>` tags as single strings. This presents a useful way to tackle the often large sizes of the languages produced when trying to string together the `<TLINK>`s to form a picture of an annotated document's temporal structure in a timeline-like manner. However, there is a notable tradeoff for using semi-intervals instead of intervals, which is that in the process of reducing the cardinality of the output languages, the vocabulary for the strings have become much more complex, with two symbols for every interval where there was only one previously, and including non-atomic formulas such as the disjunctions within string components also adds further complexity to superposition computation. Additionally, the strings which represent some of the Freksa relations are not in all cases necessarily intuitive—for instance, while $\boxed{\alpha(a),\alpha(b)\,|\,}$ shows neatly that both intervals a and b begin at the same moment for the 'head to head' relation, it's not immediately obvious why $\boxed{\alpha(b) \vee \omega(a)}$ should correspond with 'precedes'. The additional complexity that arises for the benefit of reducing the amount of data to process threatens the tenet of Freksa (1992, p. 202) that the representation should not become more complex the less is known.

As such, it would be useful to consider an alternate approach to treating situations where extracting and combining strings from the `<TLINK>` tags of a TimeML document

leads to unwieldy numbers of generated strings due to loose constraints from the relation types on the times and events in the narrative. This new approach may be comortably used alongside the semi-interval way of handling if so desired, as shall be shown.

In this treatment, rather than blindly performing vocabulary-constrained superposition between all of the strings which are extracted from the `<TLINK>` tags, which can lead to an explosion of generated strings due to a lack of constraints, only certain pairs of strings will be superposed. Where vocabulary-constrained superposition will produce a language that may contain vastly many strings, a new operation is defined which may only produce at most some fixed number of strings as output. This new operation will avoid superposition if it doesn't make sense to superpose the given strings: when the input strings are the same, the result can only be the same string again; if the strings are not the same, but they have an identical vocabulary, then the strings must contradict each other in terms of the relations they specify among their vocabulary; if the strings don't share any vocabulary, then they cannot constrain each other further through superposition, so no information would be gained; if the size of the language produced by vocabulary-constrained superposition between the languages would be larger than desired. In these last two cases, the input strings are returned so that their data is not lost, and they can potentially be superposed with other strings instead. If none of these situations holds, then the result is just the vocabulary-constrained superposition of the two strings.

First, set some limit $k \in \mathbb{N}$ which will be the maximum size of the language allowed to be produced. A good default choice for this limit is 12, on the basis that in the case where two intervals are unconstrained relative to each other, they will lead to at least 13 strings being produced by their superposition, which do not add any useful information to the knowledge base. Then, for two strings s and s' the 'sensible' superposition $s \mathbin{\hat{\&}} s'$

is defined as follows:

(122)
$$ s \mathbin{\hat{\&}} s' := \begin{cases} \{s\} & \text{if } s = s' \\[2mm] \varnothing & \text{if } s \neq s' \wedge \mathcal{V}_s = \mathcal{V}_{s'} \\[2mm] \{s, s'\} & \text{if } \mathcal{V}_s \cap \mathcal{V}_{s'} = \varnothing \text{ or } \#(s \mathbin{\&_{\scriptscriptstyle W}} s') > k \\[2mm] s \mathbin{\&_{\scriptscriptstyle W}} s' & \text{otherwise} \end{cases} $$

The examples in (123) demonstrate each of the cases in (122), assuming the limit is set as 12.

(123a) $\boxed{\;\boxed{a}\;\boxed{b}\;} \mathbin{\hat{\&}} \boxed{\;\boxed{a}\;\boxed{b}\;} = \{\boxed{\;\boxed{a}\;\boxed{b}\;}\}$

(123b) $\boxed{\;\boxed{a}\;\boxed{b}\;} \mathbin{\hat{\&}} \boxed{\;\boxed{b}\;\boxed{a}\;} = \varnothing$

(123c) $\boxed{\;\boxed{a}\;\boxed{b}\;} \mathbin{\hat{\&}} \boxed{\;\boxed{d}\;\boxed{c,d}\;\boxed{d}\;} = \{\boxed{\;\boxed{a}\;\boxed{b}\;},\; \boxed{\;\boxed{d}\;\boxed{c,d}\;\boxed{d}\;}\}$

(123d) $\boxed{\;\boxed{a}\;\boxed{b}\;} \mathbin{\hat{\&}} \boxed{\;\boxed{a}\;\boxed{c}\;} = \{\boxed{\;\boxed{a}\;\boxed{b}\;},\; \boxed{\;\boxed{a}\;\boxed{c}\;}\}$

(123e) $\boxed{\;\boxed{a}\;\boxed{b}\;} \mathbin{\hat{\&}} \boxed{\;\boxed{c}\;\boxed{b,c}\;\boxed{c}\;} = \{\boxed{\;\boxed{a,c}\;\boxed{b,c}\;\boxed{c}\;},\; \boxed{\;\boxed{a}\;\boxed{a,c}\;\boxed{b,c}\;\boxed{c}\;},\; \boxed{\;\boxed{c}\;\boxed{a,c}\;\boxed{b,c}\;\boxed{c}\;}\}$

In order to utilise this choosier form of superposition, one alteration will be made to the process of extracting strings from the `<TLINK>` tags of the TimeBank documents: now, the result of extraction will be a singleton language containing the string, rather than just the string itself. Since the result of superposition of either strings or languages is a language, this change just makes the process a little smoother.

Next, attempt to superpose all of the languages together by iterating through the list, and attempting to perform the sensible superposition for each pair of languages. If the superposition was successful, remove that pair of languages from the list, and add the result of the superposition and start attempting to superpose the languages in the updated list. If there were no successful superpositions after iterating through all the languages, then return the whole list. Since the operation is recursive, this will result in all of the languages being superposed together correctly. The operation is given in pseudo-code below.

```
superpose_languages(A, B, limit):
  return superpose_sensible(a, b, limit) for a in A, for b in B

superpose_all_languages(list, limit):
  for i=0, i++, i < len(list):
    for j=i+1, j++, j < len(list)
      sp = superpose_languages(list[i], list[j], limit)
      if sp = {}:
        # inconsistent
      elif sp = {list[i], list[j]}:
        # superposition avoided
      else:
        # add the result of superposition and
        # remove the languages that were used
        newList = sp + (list - {list[i], list[j]})
        return superposition_all_languages(newList)
  return list
```

Figure 15: Pseudo-code for superposing a list of languages of strings.

The result of this procedure will be a list of languages which were not superposed together—or if all of the languages were superposed, then the list will contain just one language. Each language will contain at most k strings, where k was the pre-determined limit in (122).

To demonstrate the utility of being choosy with when to combine temporal information, an example document is taken from the TimeBank corpus, with a doc ID of wsj_1073. The <TLINK>s of this document are given below—noting that they have been altered so that the <EVENT> and <MAKEINSTANCE> were merged.

```
<TLINK relType="SIMULTANEOUS" eventID="e9" relatedToEvent="e30" />

<TLINK relType="BEFORE" eventID="e4" relatedToTime="t0"/>

<TLINK relType="BEFORE" eventID="e2" relatedToTime="t0"/>

<TLINK relType="BEFORE" eventID="e9" relatedToTime="t0"/>

<TLINK relType="DURING" eventID="e30" relatedToTime="t31"/>
```

Figure 16: TLINKs from the wsj_1073 document of TimeBank 1.2.

The strings derived from these tags are in (124):

(124) $\{[\ [e9, e30]\ ],\ [\ [e4]\ [t0]\ ],\ [\ [e2]\ [t0]\ ],\ [\ [e9]\ [t0]\ ],\ [\ [t31]\ [e30, t31]\ [t31]\ ]\}$

Using the normal procedure, these strings are superposed together, and the result is a language containing 7,449 strings, each of which has a vocabulary of $\{e9, e30, e4, t0, e2, t31\}$. Using the sensible superposition along with the procedure in Figure 15, the result is a list of three languages: two of which contain a single string, and the third of which contains five strings:

(125) $\{[\ [t31]\ [e30, e9, t31]\ [t31]\ [t0]\ ],\ [\ [t31]\ [e30, e9, t31]\ [t31]\ [t0, t31]\ [t31]\ ],$

$[\ [t31]\ [e30, e9, t31]\ [t31]\ [t0]\ ],\ [\ [t31]\ [e30, e9, t31]\ [t31]\ [t0, t31]\ [t0]\ ],$

$[\ [t31]\ [e30, e9, t31]\ [t31]\ [t0, t31]\ ]\},\ \{[\ [e4]\ [t0]\ ]\},\ \{[\ [e2]\ [t0]\ ]\}$

These seven strings are a lot easier as a human to interpret than the over 7000 from the normal procedure. When one of the languages contains more than one string, as in this case, it is possible to translate these to use semi-intervals, and try to further reduce the cardinality that way, although this may be a case of user preference. The sensible superposition operation is the default option built into the tool presented in §5, although it is possible to remove the limit if desired.

4.2 Strings From Other Sources

The automatic parsing of the Discourse Representation Structures (DRSs) of Discourse Representation Theory (DRT) (Kamp, 1981) from plain text is a field which has recently seen some new approaches (Abzianidze et al., 2017; van Noord et al., 2018), particularly during a recent shared task due to Abzianidze et al. (2019) which aimed to take sentences of English as input and produce DRSs—in an evaluation-friendly clausal form, rather than the typical human-friendly boxed format—as output. The relative success of this task shows that there is an interest in developing systems which can perform automatic DRS parsing. The state-of-the-art was improved considerably, with the winning system

110

of Liu et al. (2019) achieving an F1 score of 84.8%, an improvement from the baseline score of 54.3%. It is possible to leverage the temporal information that features in a DRS to create strings of times and events such as are described in § 3.1. While, in general, the temporal relations found using this approach are less specific than those found in TimeML as most parsing tools for DRSs do not appear to utilise Allen's interval algebra, the advantage of using DRSs is that elements such as event participants can be picked out automatically.

Boxer (Bos, 2008) is one readily available piece of tooling which claims coverage as wide as 95% for semantic analysis of newswire texts, the same domain as the TimeBank corpus (Pustejovsky et al., 2006). The Boxer software is part of the C&C toolchain (Curran et al., 2007) for parsing a text into one or more DRSs, using a neo-Davidsonian representation for events and leveraging the set of thematic roles provided in VerbNet (Schuler, 2005), though it is noted in Bos (2008, p. 277) that the temporal aspects of the tool are not quite as strong as some of the others, and that "some measure and time expressions are correctly analysed, others aren't." A newer version of the tool is used as part of the Parallel Meaning Bank (Abzianidze et al., 2017) toolchain, although it has not been made widely available for use in research at the time of writing. Another widely available tool for automatic DRS parsing exists as part of the popular Python-based natural language processing toolkit, NLTK (Bird et al., 2009), although it is to be noted that the DRSs produced by this tool do not give much focus to their treatment of events.

4.2.1 Parallel Meaning Bank

The Parallel Meaning Bank (PMB) is an online corpus (`https://pmb.let.rug.nl`) with a total of over 12,000 DRSs for parallel structures in English, German, Dutch, and Italian. The corpus comprises sentences along with clausal form DRSs for providing fine-grained meaning representations for texts, as well as word senses and thematic roles from WordNet (Fellbaum, 2010) and VerbNet (Schuler, 2005) respectively.

Under the assumption that the version of Boxer that is used in the PMB would become

available at some point, here are some examples of extracting strings from example entries from the corpus. The idea being that, using a high-quality DRS parser to automatically determine semantic information from an unannotated text, then extracting the temporal information from these DRSs to derive strings from could present an opportunity to create a gateway towards automatic temporal annotation: since it is already possible to translate from strings to the `<TLINK>` tags of TimeML (TimeML Working Group, 2005)—utilised in the annotation tool described in §5—generating strings from another source could at the very least allow for creating a basic level of semantically informed automatic annotation of a document's temporal relations.

The goal, therefore, is to create a mapping from a text T to some set of strings S which will be used to represent the linear ordering and inter-relations of the temporal entities in T. As intermediate steps, T is parsed for semantic information which is represented as one or more DRSs. A projection from these DRSs is created, such that the resultant, reduced DRSs only contain temporal information, and S is the set of strings derived from these temporalised DRSs. The original DRSs are retained in this process so that there is a link back from the representations which have lost all non-temporal information.

Once a DRS representation is parsed from the text, the next step is to eliminate any non-temporal information from the produced DRSs. It is assumed that all temporal information is accurate. A basic approach for how to choose which information to remove is to select all discourse referents which either do not appear in a Time condition or related conditions, such as Duration, or are not explicit time expressions such as *now*, or *09:00*, and then to remove any conditions which featured a now-removed referent. Any now-empty DRSs are also removed. (126) below takes an example from Abzianidze et al. (2019, p. 8) and shows the text and the originally given DRS, though using the construction that is found in the Parallel Meaning Bank (v2.2.0, ID 99/2308). The conditions in 126b which will be retained after projection to (126c), the temporalised DRS, are shown in bold.

After removing all of the non-temporal information, a single DRS is left, which in this example corresponds to an instance of not being afraid occurring in the present. This

DRS has no information about *who* isn't afraid, or *what* they aren't afraid of, it just represents the *when*.

(126a)   Tom isn't afraid of anything.

(126b)

$$
\boxed{\begin{array}{l} x_1 \\ \hline \texttt{male.n.02}(x_1) \\ \texttt{Name}(x_1,\texttt{tom}) \\ \neg \boxed{\begin{array}{l} s_1 \;\; x_2 \;\; t_1 \\ \hline \texttt{time.n.08}(t_1) \\ t_1 \;=\; \texttt{now} \\ \texttt{afraid.a.01}(s_1) \\ \texttt{Time}(s_1,t_1) \\ \texttt{Stimulus}(s_1,x_2) \\ \texttt{Experiencer}(s_1,x_1) \\ \texttt{entity.n.01}(x_2) \end{array}} \end{array}}
$$

(126c)

$$
\neg \boxed{\begin{array}{l} s_1 \;\; t_1 \\ \hline \texttt{time.n.08}(t_1) \\ t_1 \;=\; \texttt{now} \\ \texttt{afraid.a.01}(s_1) \\ \texttt{Time}(s_1,\; t_1) \end{array}}
$$

(126d) $\boxed{\neg s_1, t_1, now}$
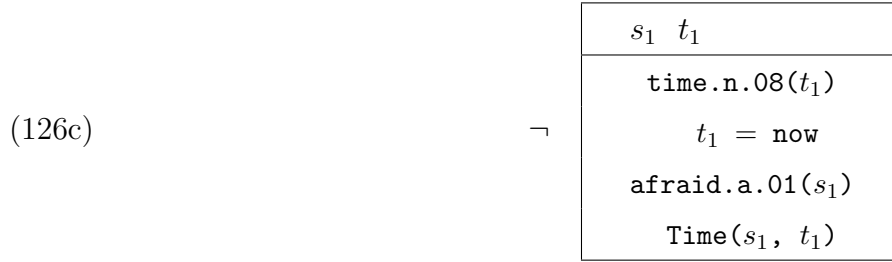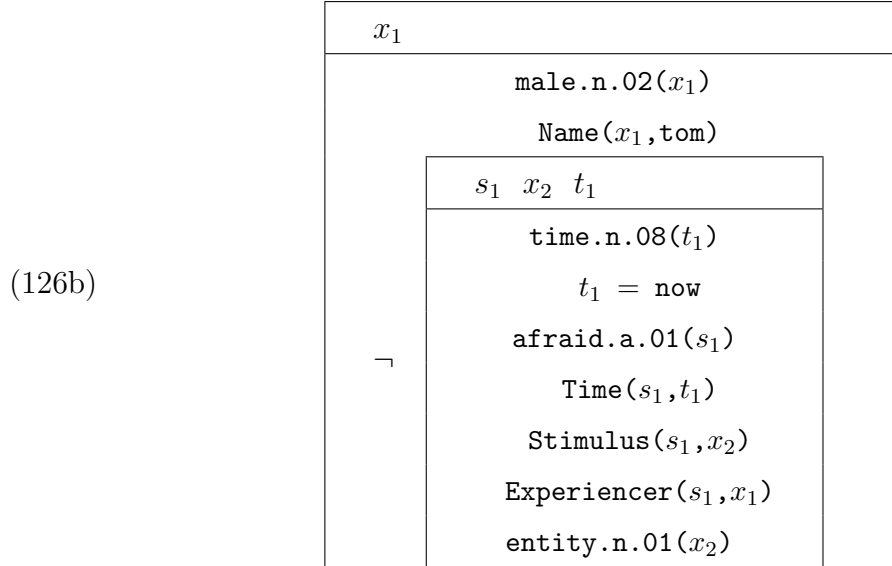
From this reduced DRS in (126c), a string can be formed, as in (126d). Fluents corresponding with the discourse referents are placed into string components (boxes) in this string, and are ordered based on any temporal relations which are mentioned in the DRS's conditions, similar to ordering fluents in strings using Allen's relations. In this example, (126c) contains $t_1 = \texttt{now}$, which indicates that any temporal entities which occur at timepoint t_1 are concurrent with the "current" moment[54], and as a result the symbols appear in a single component of the created string in (126d). This is the time at which the utterance was created, or the time of speech in Reichenbach (1947)'s framework—see § 2.1.2. Other timepoints may be referenced, such as dates, durations, relative times—for example *yesterday*—clock times, and so on, each of which must be ordered correctly. In addition to the = equals relation, there might appear a ≺ precedence, ⊂ during, or ◯ overlaps

---

[54]Note that there are other uses of *now*, but the distinctions are dropped here for simplicity.

relation, which would correspond to different orderings of the discourse referents.

Other conditions, such as negation, are to an extent also accounted for—for example, in (126d): since there is (currently) no mechanism for a negated string[55], the scope of the negation has to be narrowed to one of the fluents which appear in the string. An assumption is made that an explicit time expression such as `now` should not be negated to create a `not-now` fluent. Similarly, since there is a condition marking the referent t_1 as a time, it is also not selected to be negated. As a result, the remaining referent s_1, which is associated with the event expression, becomes a negated fluent instead[56]. Typically, a fluent appearing in a string is negated by its absence from a string component, that is, a fluent which appears in the vocabulary of a string is not occurring unless it explicitly *is* occurring. However, event-like statives are treated in TimeML, and may occur as fluents in a string, so it is not too large a step to allow negated fluents into strings.

One further utility of using the DRSs of the Parallel Meaning Bank is the ability to leverage the resources of VerbNet (Schuler, 2005), which supplies the semantic roles that permit the neo-Davidsonian event semantics in a DRS, and WordNet (Fellbaum, 2010), which is used to specify the particular sense of a verb or other word. The version of Boxer used in the PMB includes this information in its output, which may be utilised in linking events that are not given an explicit temporal relation, as events which share semantic participants are likely to be linked in some way.

4.3   Reasoning with Strings

Strings of events may be used to reason about the temporal information they contain: to infer a linear ordering and new relations that were not explicitly stated in the source as existing between the times and events modelled within the strings. There are a number methods which can be employed depending on the specific results that are desired.

Superposition of strings creates new strings which feature all of the constraints of

---

[55] Although this could be explored in future work, and would potentially reduce the discomfort of altering the scope of the negation in this manner.

[56] This has the implicit effect of shifting the sentence-level negation in "isn't afraid" to something more like "is not-afraid"—a subtle shift in this example, but nonetheless the necessity to make any alteration to the original meaning is not ideal, and should be addressed in future development of the DRS-string extraction.

their 'parent' strings, and new relations can be derived by taking the projections of these strings in relation to an intersection of the parent vocabularies. Additionally, residuals are employed to determine what relations may need to hold in order for other inferences to be made.

4.3.1 Superposition and Projection

Given some set of strings, whether extracted from an annotated document, such as one of those in the TimeBank (Pustejovsky et al., 2006) corpus, or derived from some other source like the temporal properties of DRSs, the most straightforward kind of reasoning that can be done is to use superposition to determine the relations which are not yet made explicit, moving towards the computation of the temporal closure of a document if it is not already calculated. If a document's times and events have temporal closure, then each time and event has been related to each other time and event with exactly one relational label. Once strings have been superposed together, projection can be used to find the relation(s) that they specify between any pair of times and/or events.

Strings model sets of constraints between the times and events that they mention, and when two strings are superposed, all of these constraints also hold in the resulting language's strings. This can be shown through an example:

$$(127) \qquad \boxed{\,\boxed{a}\,\boxed{b}\,} \ \&_{vc}\ \boxed{\,\boxed{b}\,\boxed{c}\,} = \{\,\boxed{\,\boxed{a}\,\boxed{b}\,\boxed{c}\,}\,\}$$

The vocabulary-constrained superposition—see (95), p. 71—produces a language of all of the strings which have a vocabulary equal to the union of the vocabularies of its inputs and also contain the same constraints as its inputs. For instance, in (127) the first input string $\boxed{\,\boxed{a}\,\boxed{b}\,}$ entails the relation a occurs before b, which is also entailed by the sole member of the result language $\boxed{\,\boxed{a}\,\boxed{b}\,\boxed{c}\,}$. This is due to the fact that vocabulary-constrained superposition preserves *projections*—see (82), p. 65—which can be proven by taking the projection of $\boxed{\,\boxed{a}\,\boxed{b}\,\boxed{c}\,}$ with respect to the vocabulary of $\boxed{\,\boxed{a}\,\boxed{b}\,}$ and

seeing that it is equal to $\boxed{\;|\,a\,|\;|\,b\,|\;}$.

(128)
$$\mathrm{bc}(\rho_{\{a,b\}}(\boxed{\;|\,a\,|\;|\,b\,|\;|\,c\,|\;}))$$
$$\mathrm{bc}(\boxed{\;|\,a\,|\;|\,b\,|\;|\;|\;})$$
$$\boxed{\;|\,a\,|\,b\,|\;}$$

The string $\boxed{\;|\,a\,|\;|\,b\,|\;|\,c\,|\;}$ thus *projects* to the string $\boxed{\;|\,a\,|\;|\,b\,|\;}$, which says that all of the temporal information represented in $\boxed{\;|\,a\,|\;|\,b\,|\;}$ is also contained within $\boxed{\;|\,a\,|\;|\,b\,|\;|\,c\,|\;}$. Similarly, the constraint that event b occurs before event c holds in both $\boxed{\;|\,b\,|\;|\,c\,|\;}$ and in $\boxed{\;|\,a\,|\;|\,b\,|\;|\,c\,|\;}$; $\boxed{\;|\,a\,|\;|\,b\,|\;|\,c\,|\;}$ projects to $\boxed{\;|\,b\,|\;|\,c\,|\;}$.

Since the resulting strings of a vocabulary-constrained superposition always[57] project back to their 'parent' strings. they also model the constraints which are represented by each of these parents, although using a single string rather than two strings. However, $\boxed{\;|\,a\,|\;|\,b\,|\;|\,c\,|\;}$ also projects to the string $\boxed{\;|\,a\,|\;|\,c\,|\;}$, which represents the constraint that event a occurs before event c according to Table 7, p. 78. This constraint was not represented in either of the parent strings that were superposed to create $\boxed{\;|\,a\,|\;|\,b\,|\;|\,c\,|\;}$, but it is represented in this result string. Thus, a new relation between events has been discovered and made explicit. While this example in (127) is relatively simple and could be easily derived from simply using the transitivity table for Allen relations—see Table 1, p. 11—this procedure works identically for strings featuring an arbitrary number of events, without having to calculate long chains of transitivities. For instance, taking another slightly trivial example for the sake of illustrating the point, imagine a set of events labelled by a through z such that, for each pair of events which are alphabetically adjacent, the 'contains' relation holds, so a contains b, b contains c, ..., y contains z. Through superposition of the strings which represent these relations a rather large string is obtained:

(129)
$$\boxed{\;|\,a\,|\,a,b\,|\,a,b,c\,|\cdots|\,a,b,\dots,y\,|\,a,b,\dots,y,z\,|\,a,b,\dots,y\,|\cdots|\,a,b,c\,|\,a,b\,|\,a\,|\;}$$

---

[57]Except, of course, if the parent strings are contradictory and the result of superposition is an empty language.

While it is not hugely difficult to follow along the transitivity chain of this particular set of 25 relations to determine that the relation between a and z is also 'contains', it only takes one step to determine this relation while using strings: namely take the projection of the string in (129) relative to the vocabulary of interest, which in this case is $\{a, z\}$, which produces the string $\boxed{|a|a,z|a|}$ which entails the 'contains' relation between a and z, by (104), p. 78. One could further imagine a scenario with the same set of events where the relational transitivities are more difficult to compute, where the relation is not the same for every single event-pair. In this case, superposition will (possibly) produce more than one string as a result, but for every string produced there is exactly one relation between every time or event which appears in that string—effectively, taking each string into consideration in isolation from all of the others, the temporal closure has been computed. In order to find the relation or relations that hold—assuming it is still the relation between a and z that is of interest in this scenario—the projection is taken of each string relative to the set $\{a, z\}$, as before, and the union of all found relations is taken. This set will either be the relation or disjunction of possible relations that may hold between a and z.

For a more concrete example, using the sensible superposition operation on the <TLINK> tags from a document—see Figure 16 and (124), p. 110—from the TimeBank corpus in § 4.1.2 produced the set of languages found in (125). To find, for example, the relation in that document's narrative between the two marked up times, $t0$ and $t31$, the projection of every string across all of the languages is taken relative to $\{t0, t31\}$, producing a set of five strings which correspond to the relations 'during', 'after', 'met by', 'finishes', and 'overlapped by'. The disjunction of these five are the possibilities for the relation between $t0$ and $t31$, corresponding to the Freksa (1992) relation 'younger'.

4.3.2 Residuals and Gaps

Given some set of strings which represent times and events and their temporal relations to serve as a knowledge base, an operation can be used to determine what strings must be added to the knowledge base in order to make some other string entailed. The operation

is known as *gap* as it effectively finds the language that would bridge the gap between the premises (the knowledge base of strings) and the conclusion (the other string).

For example, with the language $L = \{\;\boxed{\;|a|c|\;}\;\}$ as the premises and $L' = \{\;\boxed{\;|b|c|\;}\;\}$ as the conclusion, the language $gap(L, L') = \{\;\boxed{\;|a,b|\;}, \boxed{\;|b|c|\;}, \boxed{\;|b|a,b|\;}, \boxed{\;a|a,b|\;}\;\}$. Note that the conclusion itself appears in the result, since adding the conclusion to the knowledge base would indeed make it entailed by the knowledge base. Superposing any string from this resulting language *gap* with the premises should lead to the resulting updated knowledge base entailing the conclusion.

For some pair of arbitrary languages L, L', the $gap(L, L')$ is formed as follows: first, calculate the superposition between the premises and the conclusion.

$$(130) \qquad L \mathbin{\&_{vc}} L' := \{s'' \mid s \in L,\; s' \in L',\; s'' \in s \mathbin{\&_{vc}} s'\}$$

Next, find the projections $p(L, L')$ in the superposition that, when superposed with the premises, project to the conclusion.

$$(131) \qquad p(L, L') := \{\pi_V(s'') \mid s'' \in L \mathbin{\&_{vc}} L',\; V \subseteq \mathcal{V}_{s''},$$
$$(\forall s \in L,\; \forall s' \in L')(\forall r \in (\pi_V(s'') \mathbin{\&_{vc}} s))\; \pi_{\mathcal{V}_{s'}}(r) = s')\}$$

Define an operation $min(\hat{L})$ that minimises a language $\hat{L}$, such that for any string in $min(\hat{L})$, no other string in $min(\hat{L})$ will project to it.

$$(132) \qquad min(\hat{L}) := \{s \mid s \in \hat{L},\; V \subset \mathcal{V}_r,\; \pi_V(r) \notin \hat{L}\}$$

Then, $gap(L, L')$ is defined as the minimised set $p(L, L')$

$$(133) \qquad gap(L, L') := min(p(L, L'))$$

To substantiate the claim that superposing any string from $gap(L, L')$ with the premises produces the conclusion, and the further claim that any string in the superposition of L with L' projects to some string in $gap(L, L')$, $gap(L, L')$ will be related to the residual

L'/L of L' by L, relative to superposition as a binary operation on languages of block compressed strings.

First fix some large set of symbols Θ, with $Fin(\Theta)$ as the set of finite set of subsets of Θ. For any $A \in Fin(\Theta)$, define the set of all block compressed strings whose vocabulary is in A

$$(134) \qquad \mathcal{L}_A := \{\mathrm{bc}(s) \mid s \in (2^A)^*\}$$

and for any $s \in Fin(\Theta)^*$, let

$$(135) \qquad [\![s]\!]_A := \{s' \in \mathcal{L}_A \mid s' \sqsupseteq s\}$$

which is lifted to languages disjunctively, with the implication that $[\![s]\!]_A = [\![\{s\}]\!]_A$

$$(136) \qquad [\![L]\!]_A := \bigcup \{[\![s]\!]_A \mid s \in L\}$$

Note then that

$$(137) \qquad L \sqsupseteq L' \iff [\![L]\!]_A \subseteq [\![L']\!]_A$$

where $L \sqsupseteq L'$ says that the language L projects to L'—see (85), p. 66—and also

$$(138) \qquad [\![L \,\&_{vc}\, L']\!]_A = [\![L]\!]_A \cap [\![L']\!]_A$$

The first claim, that superposing any string from $gap(L, L')$ with the premises produces the conclusion, comes to

$$(139) \qquad [\![L]\!]_A \cap [\![gap(L, L')]\!]_A \subseteq [\![L']\!]_A$$

and the second claim, that any string in the superposition of L with L' projects to some

string in $gap(L, L')$

(140)
$$[\![L]\!]_A \cap [\![L']\!]_A \subseteq [\![gap(L, L')]\!]_A$$

For residuals, let

(141)
$$res_A(L, L') := \{s \in \mathcal{L}_A \mid [\![s]\!]_A \cap [\![L]\!]_A \subseteq [\![L']\!]_A\}$$

noting that, for all $L, L', L'' \subseteq \mathcal{L}_A$

(142)
$$[\![L]\!]_A \cap [\![L'']\!]_A \subseteq [\![L']\!]_A \iff [\![L'']\!]_A \subseteq res_A(L, L')$$

To find the connection between res and gap, first define the *downset* $L_\downarrow$ of L as the set of all strings which the strings in L project to

(143)
$$L_\downarrow := \{\pi_V(s) \mid s \in L, \ V \subseteq \mathcal{V}_s\}$$

And so

(144)
$$p(L, L') = \{s \in (L \ \&_{vc} \ L')_\downarrow \mid (\{s\} \ \&_{vc} \ L) \sqsupseteq L'\}$$

Observe that, for any $\hat{L}$

(145)
$$[\![min(\hat{L})]\!]_A = [\![\hat{L}]\!]_A$$

and since $gap(L, L')$ is just equal to $min(p(L, L'))$

(146)
$$[\![gap(L, L')]\!]_A = [\![res_A(L, L') \cap (L \ \&_{vc} \ L')_\downarrow]\!]_A$$

The gap operation is useful for using strings in the context of question-answering systems, for example in the FRACAS semantic test suite (Cooper et al., 1996), and allows for inferences to be made about whether certains relations could possibly exist given some

knowledge base of strings.

The next chapter describes the details of how the string framework which was described in §3 and used throughout the current chapter is currently implemented. In particular, an illustrative tool is discussed which may be used to annotate text with a slightly modified subset of the TimeML 1.2 (TimeML Working Group, 2005) schema, or to edit and examine a document that has already been marked up.

5 String Temporal Annotation and Relation Tool (START)

In order to implement the string framework for representing times and events as described in § 3 and utilised in § 4, various approaches were trialled using different programming languages, including Prolog, JavaScript (Node.js), and Python (see Clocksin and Mellish, 2012; Tilkov and Vinoski, 2010; Kuhlman, 2009, respectively for more on each). Ultimately, Python (version 3) was chosen for its speed, wide cross-platform availability, and ease of use through familiarity.

Python is a high-level, dynamically-typed, interpreted programming language implemented in C, with a large standard library that has found widespread use among scientific communities, in part thanks to the availability of tooling such as the Natural Language Toolkit (NLTK) (Bird et al., 2009) for natural language processing and PyTorch (Ketkar, 2017) for machine learning, as well as many other packages which are easily accessible through an official package installer.

Although Python has support for object-oriented programming styles, a functional programming paradigm was used in the design of the framework, wherein the use of mutable, global state is avoided. Functions are designed so as to take some input and give some output, without causing side effects, so that running a function with a particular input will always give the same result. This style of programming was chosen so as to better reflect the logical forms of the string operations which are described in § 3.1.4—a mathematical function has no state, it simply takes some input and transforms it into an output.

In order to illustrate what is enabled by the use of the string framework, a proof-of-concept tool for annotation was built as a web-based application (though, with a little effort, it may also be downloaded and set up to be used locally without an internet connection), which is available at `http://scss.tcd.ie/~dwoods/thesis/code/start`[58], and all of the code is available to view at `https://github.com/dave-woods/thesis/tree/`

---

[58] Also available at `https://start-dwoodscs.vercel.app`.

`master/code/start`. The main drive behind this tool is on using strings for discovery and consistency-checking of relations between a document's times and events, while providing a useful visual aid for a human annotator to verify that the timeline of the document, inasmuch as it can be determined, lines up with their understanding of the narrative's temporal structure.

The front-end of the application, known as START for String Temporal Annotation and Relation Tool, was developed and built using the Next.js (`https://nextjs.org`) framework upon top of the React (`https://reactjs.org`) JavaScript library, both of which are open source software. React is designed for building fast user interfaces through a component-based library that only updates the parts of the web page's document object model (DOM) which need to be updated, rather than re-rendering the entire page on every request or interaction sent from the page. Next.js enables web pages to be rendered on the server rather than on the client's machine, and provides some extra abstractions over common tasks in React. Next.js is also tightly integrated with the cloud-based hosting platform Vercel (`https://vercel.com/docs`) which allows for Python code to be hosted as part of the application in the form of so-called serverless functions (Anderson et al., 1995) where "application logic is split into functions and executed in response to events" (McGrath and Brenner, 2017, p. 405). The result of this is that for most operations of the string framework made from the application, the client will send a request to an API endpoint hosted by Vercel. On receiving the request, Vercel will trigger the appropriate function from the string framework with any data passed along with the API request, using JSON as a format for communication between the front-end and back-end. Once the function has completed, it returns any data (or errors) to the client to be used or displayed as appropriate.

For the reader's convenience, two video files are available which demonstrate the tool in use. The first is located at `https://www.scss.tcd.ie/~dwoods/thesis/code/assets/annotate-text.mp4` and shows the process of working with a pre-annotated TimeML file. The second is located at `https://www.scss.tcd.ie/~dwoods/thesis/code/assets/annotate-plain-text.mp4` and shows the process of working with and adding anno-

tation to plain text.

An important note for understanding the notation in the code and the tool is that, since strings of times and events as used in the current work are sequences of sets of symbols, the appropriate format to represent these in the code would be lists of sets—and indeed, this is the underlying representation. However, since both Python and JavaScript have strings as built-in types, an alternative but equivalent representation is used to write the strings. This increases their portability, since list in Python and arrays in JavaScript are heavier objects that use more memory than the built-in strings. In this new notation, the pipe character '|' is used to denote the boundary between two consecutive string components. For example, the string $\boxed{\ \boxed{a}\ \boxed{a,b}\ \boxed{b}\ }$ can be written as "|a|a,b|b|". A simple function can translate from a string of this kind to a list of components and back again.

5.1   Back-end API

The majority of the implementation of the framework is contained within a single Python file, which is called `strfns.py` in the Appendices. Following the functional programming style meant that any of the operations defined in § 3.1.4 would be implemented by composing together several Python functions, each of which has a single clear task. The functions need to be pure, meaning that they have no side effects, and running a function any number of times on the same input will produce the same output every time. As such, the `strfns.py` file contains only functions—in order to use them, they must be imported into another file, which is exactly what happens in the other Python files included in the Appendices. In fact, all of the other Python files also only define functions—these are the serverless functions which are hosted by Vercel. When the front-end sends a request to one of these endpoints, the `handler` class in each of `freksa.py`, `newTLINKs.py`, `superpose.py`, and `test.py` triggers the (sole) function it contains, `do_POST`, which performs the requested data manipulations and sends the results back to the client.

There is, however, one caveat to the purity of the `strfns.py` file, which is that the `superpose_sensible` function uses a decorator which implements a LRU-style cache from

the standard `functools` library. The decorator wraps the function in another function, encapsulating it and giving it access to a cache object which uses the LRU stategy: that is, the 'Least Recently Used' items in the cache will be removed to make room for new items—compare with a FIFO or 'First In First Out' strategy which says that newer entries are the most likely to be reused, where LRU says that recently used entries are the most likely to be reused. So technically the `superpose_sensible` function does have a side effect of potential cache mutation, rather than being a completely pure function. However, for all intents and purposes it is still essentially pure, in that passing the function the same inputs will always produce the same outputs, it'll just retrieve them from the cache rather than computing them after the first time.

Since the algorithm for computing the sensible superposition of a list of languages—see Figure 15, p. 109—involves a nested for-loop as well as recursion, the tradeoff for sacrificing a little functional purity and some memory space is a dramatic increase in efficiency. This can be seen best through example, so going back to the `wsj_1073` file from the TimeBank corpus, the strings derived from the `<TLINK>` tags and the result of sensibly superposing these are reproduced from (124) and (125) below for convenience.

(147) $\quad \{\boxed{\boxed{e9, e30}}, \boxed{\boxed{e4}\,\boxed{t0}}, \boxed{\boxed{e2}\,\boxed{t0}}, \boxed{\boxed{e9}\,\boxed{t0}}, \boxed{\boxed{t31}\,e30, t31\,\boxed{t31}}\}$

(148) $\quad \{\boxed{\boxed{t31}\,e30, e9, t31\,\boxed{t31}\,\boxed{t0}}, \boxed{\boxed{t31}\,e30, e9, t31\,\boxed{t31}\,t0, t31\,\boxed{t31}},$
$\boxed{\boxed{t31}\,e30, e9, t31\,\boxed{t31}\,\boxed{t0}}, \boxed{\boxed{t31}\,e30, e9, t31\,\boxed{t31}\,t0, t31\,\boxed{t0}},$
$\boxed{\boxed{t31}\,e30, e9, t31\,\boxed{t31}\,t0, t31}\}, \{\boxed{\boxed{e4}\,\boxed{t0}}\}, \{\boxed{\boxed{e2}\,\boxed{t0}}\}$

The LRU cache implementation allows for a maxsize to be set, so the most recently used 1000 (string) superpositions are kept in the cache, which can vastly improve the speed of the calculation. Using Python's built-in `timeit` functionality to test how much time repetitions of a function takes[59], with the input from (147), we find the following times while varying whether sensible superposition used the cache or not:

---

[59]Running with Python v3.7.3 (64-bit) on Ubuntu 16.04 using an Intel i7-6700 CPU with 16GB memory.

Iterations	No caching (s)	Caching (s)	$\frac{caching}{nocaching}$ (%)
1	0.02069	0.00018	0.87%
100	1.61041	0.0063	0.39%
10000	163.17628	0.26268	0.16%

Table 22: Caching superpositions vastly improving efficiency.

On average, the version using caching took less than half of one percent of the time that the non-caching version took. Since these functions are to be used as part of an API which forms the back-end to a web-based application, long waiting times are very undesirable. An annotator using a tool does not want to have to wait for the tool to keep up with them—really, a tool that makes itself noticeable for any reason rather than just facilitating the task at hand is problematic.

Other than the superposition algorithm, there are also a number of functions defined for reasoning with strings. In particular, there are a set of projection functions which use the definition of projection in (82), p. 65 and extend it so that there are definitions of a language projecting to a string if all strings in the language project to the string, a language projecting to a language if all strings in the first language project to all strings in the second, a language containing a string if some string in the language projects to it, and a language contradicting a string if the language doesn't contain it and there is some string in the language whose vocabulary contains the vocabulary of the string.

(149) $$L \sqsupseteq s' \iff \forall s \in L \ (s \sqsupseteq s')$$

(150) $$L \sqsupseteq L' \iff \forall (s \in L, \ s' \in L') \ (s \sqsupseteq s')$$

(151) $$contains(L, s') \iff \exists s \in L \ (s \sqsupseteq s')$$

(152) $$contradicts(L, s') \iff \neg contains(L, s') \land \exists s \in L \ (\mathcal{V}_s \supseteq \mathcal{V}_{s'})$$

These functions are used as part of the relation testing procedure while annotating a text with TimeML—when a user wants to add some relation between a pair of events or times, which will be added in the form of a string, first it is reasonable to test whether that string and relation are possible given the current knowledge base of relations, or if

126

it's even contradicted by some other relation.

There's also an implementation of the gap algorithm described in §4.3.2 which calculates the set of strings which would need to be superposed with some premises in order to entail some conclusion, an algorithm to determine which string of some list uses the least simultaneous resources, which helps to answer the trains problem in Table 11, p. 87, and functions to translate to and from between a string using intervals and a string using semi-intervals, as described in §4.1.2.

The file `freksa.py` is effectively a lookup table, providing the set of strings which correspond with any given relation from either Allen's set or the extended Freksa set. `newTLINKs.py` uses the concept of analogous strings—see §3.1.2, p. 50—to determine the TimeML `<TLINK>` relation type to use when translating back from strings during the export process of the START application.

5.2   Front-end Interface

A web-based application was chosen for the graphical user interface (GUI) as this ensures the widest coverage in terms of accessibility, as the only requirement is a modestly up-to-date web browser. It is possible to download the system and run it locally, but it's not envisioned that there would be much need unless planning to annotate without a web connection for an extended period of time. The key benefit of running the application in a web browser is that there is no need to download or install anything to get started, and additionally being able to design the GUI using HTML and CSS is somewhat more straightforward than some of the existing GUI toolkits for Python—again, there is no need to download or install anything new, as everything operates within the same interface.

On visiting the web page at `http://scss.tcd.ie/~dwoods/thesis/code/start`, the user is greeted with a blank textarea input. Most of the functionality is inactive until some text has been entered into this input. There are two sample files from the TimeBank corpus (Pustejovsky et al., 2006) linked on the lower left of the screen, which the user can copy into the textarea to get a feel for how to use the tool. Once the user clicks the 'Annotate' button, the system will try to parse the input as a TimeML file, but if it fails

to parse it, then it will be considered as plain text. If this is the case, then all of the features described below will work just the same, but the user will have to begin marking up from scratch, as there must be some tagged events or times to begin with before they can be related and the relation transformed into a string.
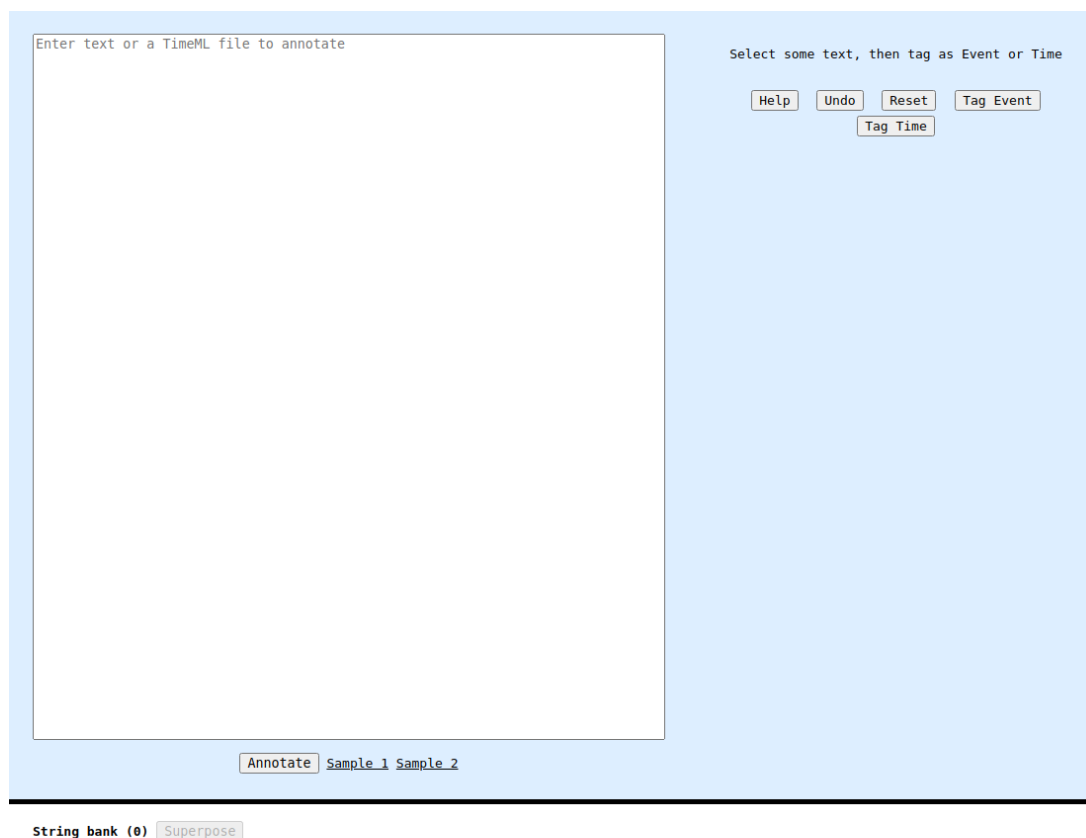
Figure 17: START: Initial blank input.

Assuming the user chose to input some TimeML[60], the input area will change to instead show the text content of the inputted document with the times and events highlighted in pink and yellow, respectively. All of these times and events are also added to a list on the right-hand side of the screen—noting that `<EVENT>` and `<MAKEINSTANCE>` tags have

---

[60]The file used in the below screenshots is the 'Sample 1' file available on the web page, which is the same document of the TimeBank corpus used in the example in Figure 16, document ID `wsj_1073`.

been amalgamated as described in §4.1.1. Finally, the strings that are derived from the <TLINK> tags are added to the 'String bank' section in the lower left of the screen, and a new panel becomes available on the lower right side.
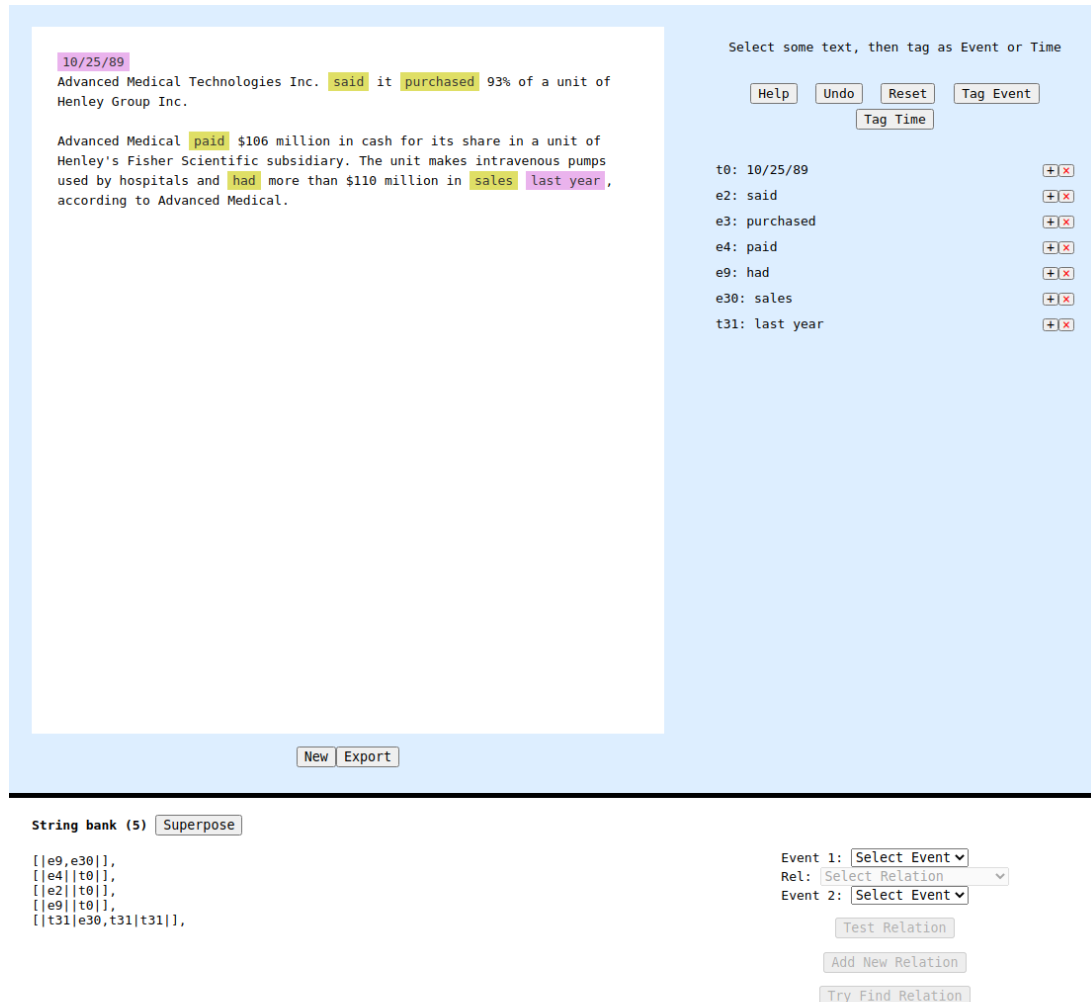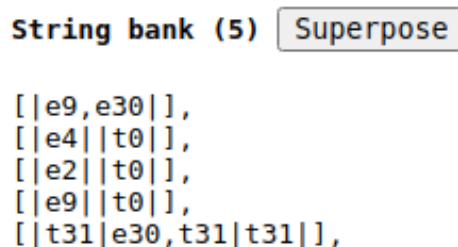
Figure 18: START: The screen after a TimeML file is imported.

At this point the user is in the main flow of the system. They are free to immediately click the 'Export' button to export the data to a new TimeML file and leave the program if they wish—noting that the export works from what the system knows of the data, rather than basing it on the original input. Internally, since the relations are stored as strings, which use Allen's set rather than the TimeML set, and some of the TimeML relations map to the same Allen relation—see Figure 7, p. 24—it is possible that the

129

relations specified in the original document will be different from those in the exported one, though it is guaranteed that they will map to the same Allen relation. Also, it is worth bearing in mind that the TimeML that is exported is a subset of the full schema, so the `<SIGNAL>`, `<SLINK>`, and `<ALINK>` tags are ignored, as they are not directly involved in the computations relating to temporal ordering.

If the user doesn't in fact leave immediately, they may notice that the temporal relations found in the document have been imported as languages of strings.

String bank (5) `Superpose`

```
[|e9,e30|],
[|e4||t0|],
[|e2||t0|],
[|e9||t0|],
[|t31|e30,t31|t31|],
```

Figure 19: START: The String bank after importing TimeML.

Clicking on the 'Superpose' button or pressing the S key on the keyboard will send a request to the back-end to compute the sensible superposition of all of the languages in the String bank, which may result in a smaller number of languages, as it does in this case. The strings have been combined where it made sense to do so and the result of superposition did not create a language with a cardinality greater than the set limit—see (122), p. 108. The limit is initially set to a default value of 12, so that strings which do not constrain each other tightly will not be superposed, though this limit can be altered by clicking the 'Help' button. Setting the limit to a value of 0 will perform the superposition without a limit at all, though this can produce unwieldy results and may be slower in terms of performance, so is not recommended if not necessary. If, during superposition, an empty set is ever returned from Python, this implies that an inconsistency was present in some of the strings in the String bank. The system will warn the user that a problem was found, and it will show the languages that were being superposed when the error occurred.

```
String bank (3)  [Superpose]

[|t31|e30,e9,t31|t31|t0,t31|t31|,
 |t31|e30,e9,t31|t31|t0|,
 |t31|e30,e9,t31|t31||t0|,
 |t31|e30,e9,t31|t31|t0,t31|t0|,
 |t31|e30,e9,t31|t31|t0,t31|      ],
[|e4||t0|],
[|e2||t0|],
```

Figure 20: START: The String bank after clicking 'Superpose'.

If the user hovers their mouse over one of the strings in the String bank, all of the highlights in the main text area will turn off except for any times and events that are mentioned in that string. If the user clicks on a string, an overlay will appear which shows the string in the conventional notation established in §3.1, with the event text appearing in place of the time or event IDs, so as to help the user decide whether the string seems valid or not, for example in the case of trying to narrow down a disjunction. A list of all the `<TLINK>` tags which can be derived from the string is also displayed. If the user wishes to remove the string from the knowledge base for whatever reason, the 'Remove' button at the top of the panel will do this. The user can press the Escape key, or click outside the panel to return to the main screen.
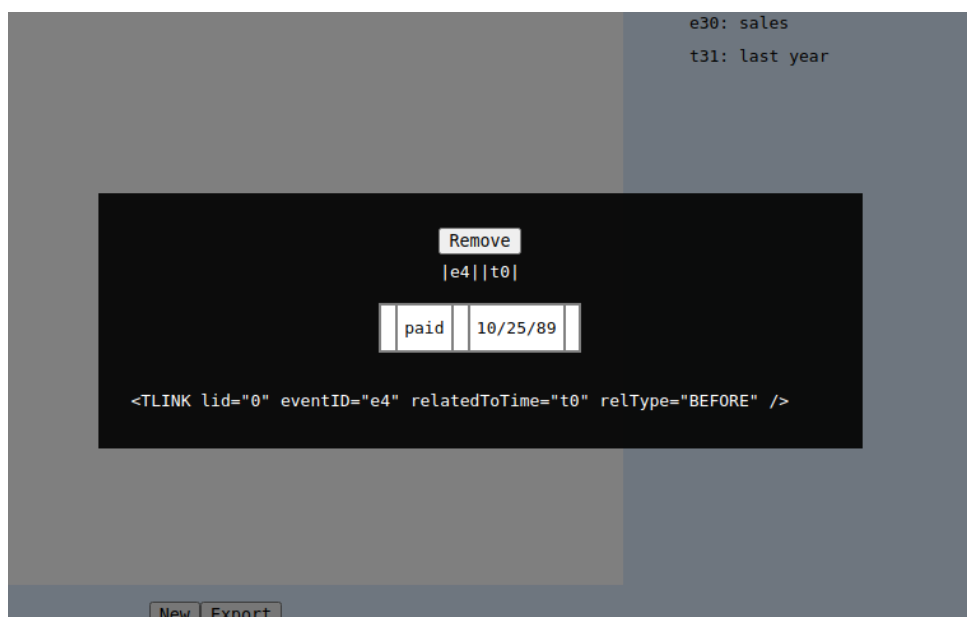
Figure 21: START: Examine a particular string.

131

Moving over to the lower right panel, there are three drop-down menus and three buttons that become available after importing at least one event or time. If the user picks two different intervals from the drop-down menus, the 'Try Find Relation' will become enabled. Clicking this will search the strings in the String bank and try to find a relation or disjunction of relations that is entailed by one or more of the strings.

Figure 22: START: The relation panel.

If the system finds any relations, these are reported to the user at the top of the screen. However, if no relations can be found from superposition of the strings, and both of the intervals are verbal events, then the system will attempt to make a recommendation of the relation based on the tenses and aspects of the pair of verbs, as in Derczynski and Gaizauskas (2013)'s empirical validation of Reichenbach (1947)'s framework of tense and aspect. The system follows the table in Figure 6, p. 16, though the suggestion will be accompanied by a warning that it *is* merely a suggestion, and if one of the verbs has the event class of 'REPORTING', this is highlighted to the user, since it raises the likelihood that the events are in separate temporal contexts. If no relations can be found for the selected pair of intervals via either methodology, the user is informed of this instead.

Figure 23: START: Relations found for a pair of intervals.

If the user selects one of the relations from the drop-down list, and then clicks 'Test Relation', the system will attempt to check the string(s)[61] relating to that relation against the String bank. It will then inform the user with one of three statuses:

- 'Found': that relation already exists in the knowledge base, though it may appear in a disjunctive language, so it may be safely added to the String bank.

- 'Possible': that relation was not found in the knowledge base, but neither was it contradicted. It may be added to the String bank.

- 'Contradicted': A string was found that was inconsistent with that relation. It should not be added to the String bank.

That relation is found according to the knowledge base.
Click 'Add New Relation' to add:
|t31||t0|

OK

Figure 24: START: The result of testing a particular relation for consistency within the knowledge base.

Clicking the 'Add New Relation' button will, as might be expected, add a string to the String bank representing the relation that is currently specified by the drop-down menus. The full set of TimeML relations is available here, with the addition of Allen's 'overlaps' and its inverse as 'OVERLAPS' and 'OVERLAPPED_BY'. The primary reasoning behind including these despite their scarcity in natural language is because of the fact that they appear in a number of the disjunctions that can occur from superposition of strings.

Moving up to the top left of the screen, where the main text is, if the user moves their mouse over one of the highlighted terms, its eventID or timeID as appropriate will be indicated to help the user distinguish in case there are several highlighted elements with the same text content. The user can tag a new event by highlighting the relevant text with the mouse, and either clicking on the 'Tag Event' button, or pressing the E

---

[61]The user can select an option in the 'Help' menu which will allow them to choose Freksa-labelled relations in addition to the normal set as the input.

key on their keyboard. The new event will be added to the panel on the right side of the screen, and will become available to be selected in the drop-down menus in order to relate the event to another interval. The process is the same for tagging a new time in the text, either clicking the 'Tag Time' button, or pressing the T key on the keyboard after selecting some text.

Finally, moving to the panel on the right, which contains a list of all of the times and events that have been tagged in the text, if the user moves their mouse over one of the items in the list, then only the text marked up for that event or time will be highlighted in the main text area, so as to ensure the user is aware exactly which interval they are dealing with. The user can click on the '+' button to the right of the time or event to bring up another overlay containing that time or event's attributes. If the interval is not one that was imported, the attributes will be set to default values, which can be edited as needed, clicking the 'Update' button to save the new values. The user can again press Escape or click outside of the panel to return to the main view. If the user clicks on the '×' button, that event or time will be removed from the document, along with any strings which mention it. This is a slightly heavy-handed approach, but it is preferable to the inconsistency of being able to export a file which contains `<TLINK>` tags that point to a non-existant event or time.
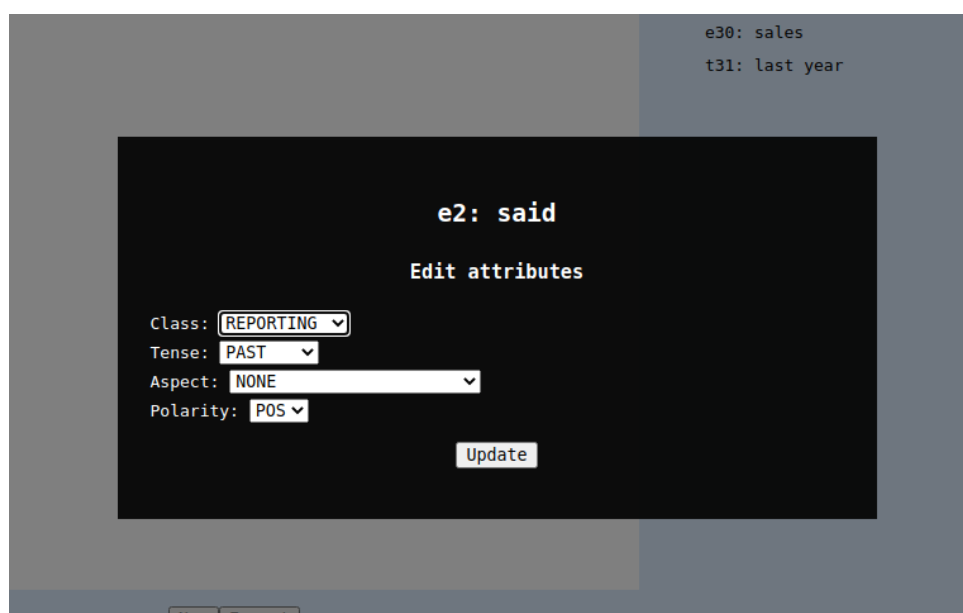
Figure 25: START: Editing some of an event's attributes.

If at any point the user makes an error or wishes to go back to a previous state for some other reason, they can click the Undo button or press the U key on their keyboard to retrace their steps through their last 50 actions. This limit was chosen somewhat arbitrarily as a means to keep a handle on the memory usage in the application, though increasing the limit is unlikely to have a significant performance impact. The use can also reset the application's state at any point by clicking the 'Reset' button, although this action cannot be undone, and this is effectively equivalent to refreshing the browser page.

When the user is finished with a file, they can click the 'Export' button which, as mentioned, will prompt the browser to download the TimeML file. One alteration to the format of the output not already mentioned is what happens when a disjunction of relations exists in the knowledge base for some pair of intervals. Rather than exporting multiple `<TLINK>` tags relating the same pair of intervals, which would instead be interpreted as the exported file containing inconsistencies, the `relType` attribute of the `<TLINK>` is interpreted as forming a disjunction of at least one relations, separated by the pipe character '|'. For example:

(153) `<TLINK eventID="e1" relatedToTime="t2" relType="BEFORE|IBEFORE" />`

While this does not conform with the TimeML schema, exactly, it is interpreted correctly if the exported file is re-imported back into START as a language of strings.

While this tool is not expected nor intended to compete with existing annotation tools, it does serve as a proof-of-concept of what is enabled when using the string framework to represent events and times. Superpositions can quickly find inconsistencies in a document's temporal relations, as well as generate new constraints and relations that may not have been previously explicit, while also presenting a convenient visual guide for the overall temporal structure of the annotated document. It is hoped that future work will have the opportunity to develop this tool further, and perhaps integrate it with other tooling as an alternate view alongside graphical and chart-based tools.

The next chapter examines some of the evaluation criteria for the present work, including ensuring that data cannot become lost or falsified when using the string framework.

6   Evaluation

In order to evaluate, it is asserted that the string framework does not produce results that are invalid—that is, string operations do not cause contradictions internally, and they do not lose, falsify, or otherwise corrupt the original source data.

6.1   Timeline Validity

The string framework allows for compact temporal data representation with an intuitive link to timelines due to the sequential nature of the data, as explored in §3.2.1. However, it is crucial that these string representations do not mislead users by creating contradictory or otherwise problematic results, or if the information being operated on becomes mutated in some way that would lead to incorrect inferences being drawn that would not have been drawn before the data was manipulated. In essence, it must be ensured that when strings containing data from a source such as a document that is marked up with TimeML from the TimeBank corpus are utilised in any way, but particular after superposition as the most heavily relied upon operation in the framework, that no information becomes lost or falsified in the process.

The notion of projection—see (82), p. 65—allows any given string to be tested as to whether it contains the same temporal data as some other string, which gives a way of verifying whether source material has been lost or corrupted in some way during data manipulation. It was shown in Table 2 that the asynchronous superposition operation, which allowed for data to be combined from pairs of strings of varying lengths, did not necessarily preserve projections from its results to its sources. As a consequence, the notion of projection was built into the definition of the vocabulary-constrained superposition operation, and it was enshrined that the only valid results of the operation would be those strings which could in fact project back to the original sources.

To validate this using a concrete example, take the strings from the TimeBank docu-

ment `wsj_1073` again:

(154) $\{ \boxed{\;\boxed{e9, e30}\;},\; \boxed{\;\boxed{e4}\;\boxed{t0}\;},\; \boxed{\;\boxed{e2}\;\boxed{t0}\;},\; \boxed{\;\boxed{e9}\;\boxed{t0}\;},\; \boxed{\;\boxed{t31}\boxed{e30, t31}\boxed{t31}\;}\}$

Doing the vocabulary-constrained superposition on these strings produces a language which contains 7,449 strings.

(155)

$\boxed{\;\boxed{e9, e30}\;} \;\&_{\!w\!c}\; \boxed{\;\boxed{e4}\;\boxed{t0}\;} \;\&_{\!w\!c}\; \boxed{\;\boxed{e2}\;\boxed{t0}\;} \;\&_{\!w\!c}\; \boxed{\;\boxed{e9}\;\boxed{t0}\;} \;\&_{\!w\!c}\; \boxed{\;\boxed{t31}\boxed{e30, t31}\boxed{t31}\;}$

(156) $= \{ \boxed{\;\boxed{t31}\boxed{e2, t31}\boxed{e2, e30, e9, t31}\boxed{e2, e30, e4, e9, t31}\boxed{e30, e9, t31}\boxed{t31}\boxed{t0}\;},$

$\boxed{\;\boxed{e2}\boxed{e2, t31}\boxed{e2, e30, e9, t31}\boxed{e2, e30, e4, e9, t31}\boxed{e30, e9, t31}\boxed{t31}\boxed{t0, t31}\boxed{t31}\;},$

$\boxed{\;\boxed{e2, t31}\boxed{e2, e30, e9, t31}\boxed{e2, e30, e4, e9, t31}\boxed{e30, e9, t31}\boxed{t31}\boxed{t0}\;},$

$\boxed{\;\boxed{t31}\boxed{e2, t31}\boxed{e2, e30, e9, t31}\boxed{e2, e30, e4, e9, t31}\boxed{e30, e9, t31}\boxed{t31}\boxed{t0, t31}\boxed{t0}\;},$

$\dots\}$

Checking each string individually, every single string in (156) projects to every string in (154), thus verifying that all of the original information has been preserved through the superposition to the resulting strings.

However, due to the unwieldy nature of vocabulary-constrained superposition which always performs superposition even when it sometimes doesn't make much sense from a practical point of view, the annotation tool in § 5 uses instead a so-called 'sensible' superposition, which avoids computing the result of superposition between strings if it is too costly to do so, based on the intersections of the inputs vocabularies and a pre-determined limit. Using this kind of superposition on (154) does not in fact produce a single language, but a set of three languages, as in (157)

(157) $\{ \boxed{\;\boxed{t31}\boxed{e30, e9, t31}\boxed{t31}\;\boxed{t0}\;},\; \boxed{\;\boxed{t31}\boxed{e30, e9, t31}\boxed{t31}\boxed{t0, t31}\boxed{t31}\;},$

$\boxed{\;\boxed{t31}\boxed{e30, e9, t31}\boxed{t31}\boxed{t0}\;},\; \boxed{\;\boxed{t31}\boxed{e30, e9, t31}\boxed{t31}\boxed{t0, t31}\boxed{t0}\;},$

$\boxed{\;\boxed{t31}\boxed{e30, e9, t31}\boxed{t31}\boxed{t0, t31}\;}\},\; \{\boxed{\;\boxed{e4}\;\boxed{t0}\;}\},\; \{\boxed{\;\boxed{e2}\;\boxed{t0}\;}\}$

It's clearly not possible for every string in (157) to project to every string in (154): for instance $\boxed{\;\boxed{e2}\;\boxed{t0}\;} \not\sqsupseteq \boxed{\;\boxed{e9, e30}\;}$. However, sensible superposition only ever produces either strings from the vocabulary-constrained superposition or the input strings themselves, so it should be possible to find the projections, and indeed it is. The sensible superposition effectively forms a segmentation of the data, meaning that rather than each string in (154) being projected to by every string in (157), every string in (154) is projected to by every string in one of the languages of (157). The second and third languages contain one string each, and each projects to one string from (154). The remaining three strings $\boxed{\;\boxed{e9, e30}\;}$, $\boxed{\;\boxed{e9}\;\boxed{t0}\;}$, and $\boxed{\;\boxed{t31}\;\boxed{e30, t31}\;\boxed{t31}\;}$ are indeed all projected to by every string in the remaining (first) language of (157), thus validating that all of the information has not been lost or corrupted during superpositions.

7 Conclusion

This thesis has described a framework for using strings as finite representations of temporal entities—times and events, as well as the relations between them. Strings have an intuitive comparison with timelines, a well-known method of conceptualising temporal information, due to their nature as sequential entities. Relevant literature was examined and discussed, in particular Allen (1983)'s interval algebra which forms a strong basis from which the string framework draws inspiration.

The string framework is described, including a number of operations which can be used to combine data from different strings and reason with them by inferring new temporal relations which weren't previously made explicit.

Some applications of the strings are given, in particular the use for augmenting semantic temporal annotation, for which a proof of concept tool is presented.

It is hoped that in the future this framework can be developed further and find a place among the tools of modern semantic temporal annotation.

Bibliography

Abzianidze, L., Bjerva, J., Evang, K., Haagsma, H., Van Noord, R., Ludmann, P., Nguyen, D.-D., and Bos, J. (2017). The Parallel Meaning Bank: Towards a Multilingual Corpus of Translations Annotated With Compositional Meaning Representations. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics (EACL)*, pages 242–247, Valencia, Spain.

Abzianidze, L., van Noord, R., Haagsma, H., and Bos, J. (2019). The First Shared Task on Discourse Representation Structure Parsing. In *Proceedings of the IWCS Shared Task on Semantic Parsing*, pages 1–15, Gothenburg, Sweden. Association for Computational Linguistics.

Allen, J. F. (1983). Maintaining Knowledge About Temporal Intervals. *Communications of the ACM*, 26(11):832–843.

Allen, J. F. and Ferguson, G. (1994). Actions and events in interval temporal logic. *Journal of logic and computation*, 4(5):531–579.

Anderson, T. E., Dahlin, M. D., Neefe, J. M., Patterson, D. A., Roselli, D. S., and Wang, R. Y. (1995). Serverless network file systems. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 109–126.

Applegate, D. and Cook, W. (1991). A computational study of the job-shop scheduling problem. *ORSA Journal on computing*, 3(2):149–156.

Aristotle. The Internet Classics Archive: Physics IV. Web Download: `http://classics.mit.edu/Aristotle/physics.4.iv.html`. Accessed: 2018-08-12.

Bird, S., Klein, E., and Loper, E. (2009). *Natural Language Processing with Python*. O'Reilly Media, Inc.

Bittar, A., Amsili, P., Denis, P., and Danlos, L. (2011). French TimeBank: an ISO-TimeML annotated reference corpus. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 130–134.

Bos, J. (2008). Wide-Coverage Semantic Analysis with Boxer. In *Proceedings of the 2008 Conference on Semantics in Text Processing - STEP '08*, pages 277–286.

Bos, J. and Abzianidze, L. (2019). Thirty musts for meaning banking. In *Proceedings of ACL 2019 First International Workshop on Designing Meaning Representations*, pages 15–27.

Bottini, R., Crepaldi, D., Casasanto, D., Crollen, V., and Collignon, O. (2015). Space and time in the sighted and blind. *Cognition*, 141:67–72.

Buchner, A. and Funke, J. (1993). Finite-state automata: Dynamic task environments in problem-solving research. *The Quarterly Journal of Experimental Psychology*, 46(1):83–118.

Bunt, H. (2020). Annotation of Quantification: The Current State of {ISO} 24617-12. *Proceedings of the 16th Joint ISO-ACL Workshop on Interoperable Semantic Annotation (ISA-16)*, (May):1–12.

Bunt, H. and Pustejovsky, J. (2010). Annotating temporal and event quantification. In *Proceedings of the 5th Joint ISO-ACL Workshop on Interoperable Semantic Annotation (ISA-5)*, Hong Kong.

Bunt, H. C. (1985). *Mass terms and model-theoretic semantics*. Cambridge University Press, Cambridge.

Clocksin, W. F. and Mellish, C. S. (2012). *Programming in Prolog: Using the ISO standard*. Springer Science & Business Media.

Cooper, R., Crouch, D., Van Eijck, J., Fox, C., Van Genabith, J., Jaspars, J., Kamp, H., Milward, D., Pinkal, M., Poesio, M., et al. (1996). Using the framework. Technical report, Technical Report LRE 62-051 D-16, The FraCaS Consortium.

Curran, J. R., Clark, S., and Bos, J. (2007). Linguistically motivated large-scale nlp with c&c and boxer. In *Proceedings of the 45th annual meeting of the Association for Computational Linguistics Companion volume proceedings of the demo and poster sessions*, pages 33–36.

Davidson, D. (1967). The logical form of action sentences. In Rescher, N., editor, *The Logic of Decision and Action*, pages 81–95. University of Pittsburgh Press.

Day, D., Aberdeen, J., Hirschman, L., Kozierok, R., Robinson, P., and Vilain, M. (1997). Mixed-Initiative Development of Language Processing Systems. In *Fifth Conference on Applied Natural Language Processing*, pages 348–355.

Derczynski, L. (2016). Representation and Learning of Temporal Relations. In *Proceedings of COLING 2016, the 26th International Conference on Computational Linguistics: Technical Papers*, pages 1937–1948.

Derczynski, L. and Gaizauskas, R. (2013). Empirical Validation of Reichenbach's Tense Framework. In *Proceedings of the 10th International Conference on Computational Semantics (IWCS 2013)*, pages 71–82.

Dowty, D. R. (1989). On the semantic content of the notion of thematic role. In *Properties, types and meaning*, pages 69–129. Springer.

Durand, I. and Schwer, S. (2008a). Reasoning about qualitative temporal information with s-words and s-languages. In *European Lisp Symposium*, pages 1–15.

Durand, I. and Schwer, S. R. (2008b). A Tool for Reasoning about Qualitative Temporal Information: the Theory of S-languages with a Lisp Implementation. *Journal of Universal Computer Science*, 14(20):3282–3306.

Elgot, C. C. and Rabin, M. O. (1966). Decidability and undecidability of extensions of second (first) order theory of (generalized) successor. *The Journal of Symbolic Logic*, 31(2):169–181.

Fellbaum, C. (2010). Wordnet. In *Theory and applications of ontology: computer applications*, pages 231–243. Springer.

Fernando, T. (2004). A Finite-State Approach to Events in Natural Language semantics. *Journal of Logic and Computation*, 14(1):79–92.

Fernando, T. (2015). The Semantics of Tense and Aspect: A Finite-State Perspective. In Lappin, S. and Fox, C., editors, *The Handbook of Contemporary Semantic Theory*, number August, pages 203–236. John Wiley & Sons.

Fernando, T. (2016a). On Regular Languages Over Power Sets. *Journal of Language Modelling*, 4(1):29–56.

Fernando, T. (2016b). Prior and Temporal Sequences for Natural Language. *Synthese*, 193(11):3625–3637.

Fernando, T. (2018). Intervals and Events with and without Points. In *Workshop on Logic and Algorithms in Computational Linguistics 2018 (LACompLing2018)*, pages 34–46, Stockholm.

Fernando, T. and Nairn, R. (2005). Entailments in finite-state temporality. In *Proceedings of the Sixth International Workshop on Computational Semantics*, pages 128–138, Tilburg University.

Fernando, T. and Vogel, C. (2019). Prior Probabilities of Allen Interval Relations over Finite Orders. In *Proceedings of the 11th International Conference on Agents and Artificial Intelligence*, volume 2: NLPinAI, pages 952–961.

Fernando, T., Woods, D., and Vogel, C. (2019). MSO with tests and reducts. In *Proceedings of the 14th International Conference on Finite-State Methods and Natural Language Processing*, pages 27–36.

Freksa, C. (1992). Temporal Reasoning Based on Semi-Intervals. *Artificial Intelligence*, 54:199–227.

Geurts, B., Beaver, D. I., and Maier, E. (2020). Discourse Representation Theory. In Zalta, E. N., editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, spring 2020 edition.

Goel, P., Prabhu, S., Debnath, A., Modi, P., and Shrivastava, M. (2020). Hindi TimeBank: An ISO-TimeML Annotated Reference Corpus. In *Proceedings of the 16th Joint ACL-ISO Workshop on Interoperable Semantic Annotation*, pages 13–21.

Gong, X., Deng, Q., Gong, G., Liu, W., and Ren, Q. (2018). A memetic algorithm for multi-objective flexible job-shop problem with worker flexibility. *International Journal of Production Research*, 56(7):2506–2522.

Hamblin, C. L. (1972). Instants and intervals. In *The Study of Time*, pages 324–331. Springer.

Hornstein, N. (1990). *As time goes by: Tense and universal grammar*. Mit Press.

ISO 24617-1:2012 (2012). Language resource management Semantic annotation framework (SemAF) Part 1: Time and events (SemAF-Time, ISO-TimeML). Standard, International Organization for Standardization, Geneva, CH. `https://www.iso.org/standard/37331.html` This standard was last reviewed and confirmed in 2017. Therefore this version remains current.

Kamp, H. (1981). A theory of truth and discourse representation. *Formal methods in the study of language*, (135).

Kamp, H. (1988). Discourse Representation Theory: What it is and where it ought to go. *Natural Language at the computer*, 320(1):84–111.

Kamp, H. and Reyle, U. (1993). *From Discourse to Logic; An Introduction to Model-theoretic Semantics of Natural Language, Formal Logic and DRT*. Dordrecht: Kluwer.

Ketkar, N. (2017). Introduction to pytorch. In *Deep learning with python*, pages 195–208. Springer.

Khourshid, D. (2015). XState. Web Download: `https://xstate.js.org/docs/about/concepts.html#finite-state-machines`. Accessed: 2021-03-10.

Kowalski, R. and Sergot, M. (1986). A Logic-based Calculus of Events. *New Generation Computing*, 4(1):67–95.

Kuhlman, D. (2009). *A python book: Beginning python, advanced python, and python exercises*. Dave Kuhlman Lutz.

Kumar, P. P. (2005). Effective use of Gantt chart for managing large scale projects. *Cost engineering*, 47(7):14.

Lakoff, G. and Johnson, M. (2008). *Metaphors we live by*. University of Chicago press.

Libkin, L. (2004). Monadic Second-Order Logic and Automata. In *Elements of Finite Model Theory*, pages 113–140. Springer-Verlag, Berlin, Heidelberg.

Liu, J., Cohen, S. B., and Lapata, M. (2019). Discourse representation structure parsing with recurrent neural networks and the transformer model. In *Proceedings of the IWCS Shared Task on Semantic Parsing*, pages 24–29, Gothenburg, Sweden. Association for Computational Linguistics.

Mani, I., Verhagen, M., Wellner, B., Lee, C., and Pustejovsky, J. (2006). Machine learning of temporal relations. In *Proceedings of the 21st International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics*, pages 753–760.

Manne, A. S. (1960). On the job-shop scheduling problem. *Operations Research*, 8(2):219–223.

Maylor, H. (2001). Beyond the Gantt chart:: Project management moving on. *European management journal*, 19(1):92–100.

McCarthy, J. (1980). Circumscription—a form of non-monotonic reasoning. *Artificial intelligence*, 13(1-2):27–39.

McCarthy, J. and Hayes, P. J. (1969). Some Philosophical Problems from the Standpoint of Artificial Intelligence. In *Machine Intelligence*, pages 463–502. Edinburgh University Press.

McGrath, G. and Brenner, P. R. (2017). Serverless computing: Design, implementation, and performance. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 405–410. IEEE.

Miller, R. and Shanahan, M. (1999). The Event Calculus in Classical LogicAlternative Axiomatizations. *Electronic Transactions on Artificial Intelligence*, 3:77–105.

Mitchell, W. J. (1980). Spatial form in literature: Toward a general theory. *Critical Inquiry*, 6(3):539–567.

Mueller, E. T. (2008). Event Calculus. In *Foundations of Artificial Intelligence*, volume 3, pages 671–708.

Ohtsuka, K. and Brewer, W. (1992). Discourse Organization in the Comprehension of Temporal Order in Narrative Texts. *Discourse Processes*, 15.

Pinedo, M. and Hadavi, K. (1992). Scheduling: theory, algorithms and systems development. In *Operations Research Proceedings 1991*, pages 35–42. Springer.

Pustejovsky, J., Castano, J. M., Ingria, R., Sauri, R., Gaizauskas, R. J., Setzer, A., Katz, G., and Radev, D. (2003a). TimeML: Robust specification of event and temporal expressions in text. *New directions in question answering*, 3:28–34.

Pustejovsky, J., Hanks, P., Saurí, R., See, A., Gaizauskas, R., Setzer, A., Radev, D., Sundheim, B., Day, D., and Ferro, L. (2003b). The TIMEBANK Corpus. In *Corpus Linguistics*, volume 2003, pages 647–656. Lancaster, UK.

Pustejovsky, J., Knippen, R., Littman, J., and Saurí, R. (2005). Temporal and Event Information in Natural Language Text. *Language Resources and Evaluation*, 39(2):123–164.

Pustejovsky, J., Lee, K., Bunt, H., and Romary, L. (2010). ISO-TimeML: An International Standard for Semantic Annotation. In *Proceedings of the Seventh International Conference on Language Resources and Evaluation (LREC'10)*, pages 394–397.

Pustejovsky, J., Mani, I., Belanger, L., van Guilder, L., Knippen, R., See, A., Schwarz, J., and Verhagen, M. (2003c). Tango Final Report. In *ARDA Summer Workshop on Graphical Annotation Toolkit for TimeML, MITRE Bedford and Brandeis University*.

Pustejovsky, J., Verhagen, M., Saurí, R., Littman, J., Gaizauskas, R., Katz, G., Mani, I., Knippen, R., and Setzer, A. (2006). TimeBank 1.2 LDC2006T08. Web Download: `https://catalog.ldc.upenn.edu/LDC2006T08`. Philadelphia: Linguistic Data Consortium.

Reichenbach, H. (1947). The Tenses of Verbs. In *Elements of Symbolic Logic*, chapter 51, pages 287–298. The Macmillan Company, New York.

Reimers, N., Dehghani, N., and Gurevych, I. (2016). Temporal anchoring of events for the timebank corpus. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2195–2204.

Rosenberg, D. and Grafton, A. (2013). *Cartographies of time: A history of the timeline*. Princeton Architectural Press.

Saurí, R., Littman, J., Knippen, B., Gaizauskas, R., Setzer, A., and Pustejovsky, J. (2006). TimeML Annotation Guidelines Version 1.2.1. Technical report, Brandeis University Linguistics Department.

Schuler, K. K. (2005). *VerbNet: A broad-coverage, comprehensive verb lexicon.* PhD thesis, University of Pennsylvania.

Schwer, S. R. (2000). Number of different relations between n intervals on a line. Sequence A055203 in *The On-Line Encyclopedia of Integer Sequences.* `https://oeis.org/A055203`.

Schwer, S. R. (2002). S-arrangements avec répétitions. *Comptes Rendus Mathematique*, 334(4):261–266. In French, abbreviated version in English.

Shanahan, M. (1997). *Solving the Frame Problem: A Mathematical Investigation of the Common Sense Law of Inertia.* MIT press.

Stangroom, J. (2009). *Einstein's Riddle: Riddles, Paradoxes, and Conundrums to Stretch Your Mind.* Bloomsbury USA.

Steedman, M. (2000). *The syntactic process*, volume 24. MIT press Cambridge, MA.

Steedman, M. and Baldridge, J. (2011). *Combinatory Categorial Grammar*, chapter 5, pages 181–224. John Wiley & Sons, Ltd.

Thielscher, M. (1999). From situation calculus to fluent calculus: State update axioms as a solution to the inferential frame problem. *Artificial intelligence*, 111(1-2):277–299.

Tilkov, S. and Vinoski, S. (2010). Node.js: Using JavaScript to Build High-Performance Network Programs. *IEEE Internet Computing*, 14(6):80–83.

TimeML Working Group (2005). TimeML 1.2.1. A Formal Specification Language for Events and Temporal Expressions. Web Download: `http://www.timeml.org/publications/timeMLdocs/timeml_1.2.1.html`. Accessed: 2018-08-10.

Trakhtenbrot, B. A. (1953). On recursive separability. *Doklady Akademii Nauk SSSR*, 88(6):953–956. In Russian.

UzZaman, N., Llorens, H., Derczynski, L., Allen, J., Verhagen, M., and Pustejovsky, J. (2013). Semeval-2013 Task 1: TempEval-3: Evaluating Time Expressions, Events, and Temporal Relations. In *Second Joint Conference on Lexical and Computational Semantics (* SEM), Volume 2: Proceedings of the Seventh International Workshop on Semantic Evaluation (SemEval 2013)*, pages 1–9.

Van Lambalgen, M. and Hamm, F. (2008). *The Proper Treatment of Events*, volume 6. John Wiley & Sons.

van Noord, R., Abzianidze, L., Toral, A., and Bos, J. (2018). Exploring neural methods for parsing discourse representation structures. *Transactions of the Association for Computational Linguistics*, 6:619–633.

Veanes, M., Hooimeijer, P., Livshits, B., Molnar, D., and Bjorner, N. (2012). Symbolic finite state transducers: Algorithms and applications. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 137–150.

Verhagen, M. (2005a). Drawing TimeML Relations with T-BOX. In Katz, G., Pustejovsky, J., and Schilder, F., editors, *Annotating, Extracting and Reasoning about Time and Events*, number 05151 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany. Internationales Begegnungs und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

Verhagen, M. (2005b). Temporal closure in an annotation environment. *Language Resources and Evaluation*, 39(2-3):211–241.

Wintner, S. (2007). Finite-state technology as a programming environment. In *International Conference on Intelligent Text Processing and Computational Linguistics*, pages 97–106. Springer.

Woods, D. and Fernando, T. (2018). Improving String Processing for Temporal Relations. In *Proceedings of the 14th Joint ISO-ACL Workshop on Interoperable Semantic Annotation (ISA-14)*, pages 76–86.

Woods, D., Fernando, T., and Vogel, C. (2017). Towards Efficient String Processing of Annotated Events. In *Proceedings of the 13th Joint ISO-ACL Workshop on Interoperable Semantic Annotation (ISA-13)*, pages 124–133.

Yu, S. (1997). Regular Languages. In Rozenberg, G. and Salomaa, A., editors, *Handbook of Formal Languages, Volume. 1: Word, Language, Grammar*, pages 41–110. Springer-Verlag New York, Inc., New York, NY.

Appendices

Python Code

```python
                          """strfns.py"""
from functools import reduce, lru_cache
from collections import Counter
import itertools
import re


def negate_component(component, as_string=False):
    """Take a component, and return it with all its fluents
        negated"""
    c = list(map(lambda f: f[1:] if f == '' or f.startswith('
        !') else '!'+f, component))
    return c if not as_string else ','.join(c)


def negate_string(string):
    """Take a string, and return it with every component
        negated"""
    return '|'.join([','.join(c) for c in map(
        negate_component, get_components(string))])


def hide_negated(string):
    """Take a string and filter out any negated fluents"""
    newc = []
    for c in get_components(string):
        newc.append(list(filter(lambda f: not f.startswith('!
            '), c)))
    return '|'.join([','.join(c) for c in newc])


def subsets(s):
    """Take an iterable and return the set of its subsets"""
    return frozenset(frozenset(x) for x in itertools.chain.
        from_iterable(itertools.combinations(s, r) for r in
        range(len(s)+1)))


def proper_subsets(s):
    """Take an iterable and return the set of its proper
```

```python
        subsets"""
    return frozenset(frozenset(x) for x in itertools.chain.
        from_iterable(itertools.combinations(s, r) for r in
        range(len(s))))


def nonempty_union(x, y):
    """Take two sets and return their union"""
    try:
        z = frozenset(x) | frozenset(y)
    except:
        z = x + y
    return list(filter(None, z)) if len(z) > 1 else list(z)


def get_components(string):
    """Take a string and return a list of components"""
    try:
        return [s.split(',') for s in string.replace(' ', '')
            .split('|')]
    except AttributeError:
        return string


def string_from_components(components):
    """Take a list of components and return a string"""
    return '|'.join([','.join(c) for c in components])


def vocabulary(string):
    """Take a string, return its vocabulary as a set"""
    if string == '' or string == []:
        return frozenset()
    try:
        components = get_components(string)
        return frozenset(filter(None, reduce(lambda x, y:
            list(frozenset(x) | frozenset(y)), components)))
    except AttributeError:
        return frozenset(filter(None, reduce(lambda x, y:
            list(frozenset(x) | frozenset(y)), string)))


def vocabulary_lang(lang):
    """Take a language, return its vocabulary"""
```

```python
    return frozenset(reduce(lambda x, y: list(vocabulary(x) |
        vocabulary(y)), lang, []))

def string_length(string):
    """Take a string, return the number of components it has
        """
    return len(string.split('|'))

def sort_fluents(string):
    """Take a string, return a string with all its fluents
        alphabetised within their components"""
    return '|'.join([','.join(sorted(x.split(','))) for x in
        string.split('|')])

def string_equals(a, b):
    """Take two strings, return whether they contain the same
         data"""
    return [sorted(x.split(',')) for x in a.split('|')] == [
        sorted(x.split(',')) for x in b.split('|')]

def reduct(string, new_vocab):
    """Take a string and a set, return the string with its
        vocab set to the set"""
    components = get_components(string)
    reducted = [[f for f in c if f in new_vocab] for c in
        components]
    return '|'.join([','.join(c) for c in reducted])

def block_compress(string):
    """Take a string, return the string without stutter"""
    components = get_components(string)
    if len(components) < 2:
        return '|'.join([','.join(c) for c in components])
    elif Counter(components[0]) == Counter(components[1]):
        return block_compress('|'.join([','.join(c) for c in
            components[1:]]))
    else:
        return ','.join(components[0]) + '|' + block_compress
            ('|'.join([','.join(c) for c in components[1:]]))
```

```python
def projection(string, proj_set):
    """Take a string and a set, return the block compressed
        reduct"""
    return block_compress(reduct(string, proj_set))


def projection_lang(lang, new_vocab):
    """Take a language and a set, return the block compressed
        reduct of every string in the language"""
    return list(set(filter(lambda n: n != '', [projection(s,
        new_vocab) for s in lang])))


def string_projects_to_string(a, b):
    """Take two strings, return whether the first projects to
        the second"""
    return string_equals(b, projection(a, vocabulary(b)))


def lang_projects_to_string(a, b):
    """Take a language and a string, return whether the first
        projects to the second"""
    return all([string_projects_to_string(s, b) for s in a])


def lang_projects_to_lang(a, b):
    """Take two languages, return whether the first projects
        to the second"""
    return all([lang_projects_to_string(a, s) for s in b])


def lang_contains_string(a, b):
    """Take a language and a string, return whether the first
        contains the second"""
    return any([string_projects_to_string(s, b) for s in a])


def lang_contradicts_string(a, b):
    """Take a language and a string, return whether the first
        contradicts the second"""
    return not lang_contains_string(a,b) and any([vocabulary(
        s).issuperseteq(vocabulary(b)) for s in a])


def projection_full_vocab(string, vocab):
```

```python
    """Take a string and a set, return the block compressed
        reduct or an empty string if the full vocab isn't used
        """
    p = projection(string, vocab)
    return p if vocabulary(p) == frozenset(vocab) else ''


def projection_lang_full_vocab(lang, new_vocab):
    """Take a language and a set, return the block compressed
        reduct of every string in the language or empty
        strings if the full vocab isn't used"""
    return list(set(filter(lambda n: n != '', [
        projection_full_vocab(s, new_vocab) for s in lang])))


def analogous_strings(a, b):
    """Take two strings and return whether they are analogous
        """
    v_a = vocabulary(a)
    v_b = vocabulary(b)
    mapping = dict()
    for v in v_a:
        for w in v_b.difference(frozenset(mapping.values())):
            if reduct(a, [v]) == reduct(b, [w]).replace(w, v)
                :
                    mapping[v] = w
                    break
    return frozenset(mapping.keys()) == v_a and frozenset(
        mapping.values()) == v_b, mapping


def basic_sp(string_a, string_b):
    """Take two strings, return their basic superposition (a
        string)"""
    components_a = get_components(string_a)
    components_b = get_components(string_b)
    return sort_fluents(string_from_components([
        nonempty_union(a, b) for (a, b) in zip(components_a,
        components_b)]))


def basic_sp_lang(lang_a, lang_b):
    """Take two languages, return their basic superposition (
```

```python
        a language)"""
    result = []
    for a in lang_a:
        for b in lang_b:
            result.append(basic_sp(a, b))
    return frozenset(result)


def pad(string, length):
    """Take a string and an int, and pad the string to the
        length of the int"""
    sl = string_length(string)
    if length < sl:
        return []
    elif length == sl:
        return [string]
    else:
        result = []
        for s in pad(string, length-1):
            c = get_components(s)
            result = result + [string_from_components(c[:i] +
                [c[i]] + c[i:]) for i in range(len(c))]
        return frozenset(result)


def async_sp(string_a, string_b):
    """Take two strings and return their asynchronous
        superposition (a language)"""
    len_a = string_length(string_a)
    len_b = string_length(string_b)
    pad_len = len_a + len_b - 1
    padded_a = pad(string_a, pad_len)
    padded_b = pad(string_b, pad_len)
    return frozenset(map(lambda s: block_compress(s),
        basic_sp_lang(padded_a, padded_b)))


def superpose(string_a, string_b, vocab_a = None, vocab_b =
    None, remove_negated_pairs = True):
    """Take two strings and return their vocabulary-
        constrained superposition (a language)"""
    components_a = get_components(string_a)
```

```
        components_b = get_components(string_b)
        if vocab_a is None and vocab_b is None:
            vocab_a = vocabulary(string_a)
            vocab_b = vocabulary(string_b)
            return frozenset(map(lambda x: sort_fluents(
                string_from_components(x)), superpose(string_a,
                string_b, vocab_a, vocab_b)))

        if not components_a and not components_b:
            return [[]]
        if not components_a or not components_b:
            return []

        if frozenset(vocab_a) & frozenset(components_b[0]) <=
            frozenset(components_a[0]) and frozenset(vocab_b) &
            frozenset(components_a[0]) <= frozenset(components_b
            [0]):
            head_union = nonempty_union(components_a[0],
                components_b[0])

            # removes '|a|b| & ||a|b|' cases
            if remove_negated_pairs:
                for zz in head_union:
                    if zz.startswith('!') and zz[1:] in
                        head_union:
                            return []

            l = L(components_a[0], components_a[1:], vocab_a,
                components_b[0], components_b[1:], vocab_b)
            return [[head_union] + l_item for l_item in l]

        return []

def L(head_a, tail_a, vocab_a, head_b, tail_b, vocab_b):
    """Part of vocabulary-constrained superposition"""
    part_1 = superpose([head_a] + tail_a, tail_b, vocab_a,
        vocab_b)
    part_2 = superpose(tail_a, [head_b] + tail_b, vocab_a,
        vocab_b)
```

```python
    part_3 = superpose(tail_a, tail_b, vocab_a, vocab_b)
    return nonempty_union(nonempty_union(part_1, part_2),
        part_3)


def superpose_all(list_of_strings):
    """Take a list of strings, return the result of
        superposing them all (a language)"""
    if type(list_of_strings) != list:
        raise TypeError
    elif len(list_of_strings) == 0:
        raise Exception('Cannot use empty list')
    elif len(list_of_strings) == 1:
        yield list_of_strings[0]
    else:
        for sp in superpose(list_of_strings[0],
            list_of_strings[1]):
            yield from superpose_all([sp] + list_of_strings
                [2:])


def superpose_langs(lang1, lang2):
    """Take a pair of languages, return their superposition (
        a language)"""
    for s1 in lang1:
        for s2 in lang2:
            yield from superpose(s1, s2)


def superpose_all_langs(list_of_langs, filt=None):
    """Take a list of languages with an optional filter for
        external constraints, return the result of superposing
        them all (a language)"""
    running = list_of_langs[0]
    for lang in list_of_langs[1:]:
        yield running
        running = [s for s in superpose_langs(running, lang)
            if filt is None or filt(s)]
    yield running


@lru_cache(maxsize=1000)
def superpose_sensible(a, b, limit = 0):
```

```python
    """Take two strings and a limit, return the superposition
        of the strings (a language) where it sensible to do
        so"""
    if sort_fluents(a) == sort_fluents(b):
        return [frozenset([a])]
    v_a = vocabulary(a)
    v_b = vocabulary(b)
    if v_a == v_b:
        return []
    elif len(v_a & v_b) == 0:
        return [frozenset([a]), frozenset([b])]
    else:
        sp = superpose(a, b)
        if limit > 0 and len(sp) > limit:
            return [frozenset([a]), frozenset([b])]
        else:
            return [sp]


def superpose_langs_sensible(lang1, lang2, limit = 0):
    """Take two languages and a limit, return the
        superposition of the languages (a language) where it
        sensible to do so"""
    yield from set([item for s1 in lang1 for s2 in lang2 for
        sublist in superpose_sensible(s1, s2, limit) for item
        in sublist])


def superpose_all_langs_sensible(list_of_langs, limit = 0):
    """Take a list of languages and a limit, return the
        superposition of the languages (a list of languages)
        """
    yielded = False
    for i, l in enumerate(list_of_langs):
        for j, ll in enumerate(list_of_langs[i+1:]):
            if yielded:
                break
            sp = list(superpose_langs_sensible(l, ll, limit))
            if sp == []:
                raise Exception('Contradiction:␣{}␣and␣{}'.
                    format(l, ll))
```

```python
            elif set(l + ll) == set(sp) or (limit > 0 and len
                (sp) > limit):
                pass # no sp
            else:
                yielded = True
                yield from superpose_all_langs_sensible([sp]+
                    list_of_langs[:i]+list_of_langs[i+1:i+1+j
                    ]+list_of_langs[i+1+j+1:], limit)
    if not yielded:
        yield from list_of_langs


def flatten_list(deep_list):
    """Take a nested list and return it flattened"""
    return [item for sublist in deep_list for item in sublist
        ]


def gap(premises, conclusion):
    """Take two languages and return the set of strings which
        , when superposed with the premises, would entail the
        conclusion"""
    sl = superpose_langs(premises, conclusion)
    presiduals = set()
    for s in sl:
        for proj in [projection(s, v) for v in subsets(
            vocabulary(s))]:
            prem_proj = flatten_list([superpose(proj, prem)
                for prem in premises])
            if all(string_equals(projection(pp, vocabulary(c)
                ), c) for c in conclusion for pp in prem_proj)
                :
                presiduals.add(proj)
    minimal = []
    for r in presiduals:
        if all(projection(r, v) not in presiduals for v in
            proper_subsets(vocabulary(r))):
            minimal.append(r)
    return minimal


def most_simultaneous_events_occurring(string):
```

```python
    """Take a string and return the size of the largest
       component"""
    return max(map(lambda c: len(c), get_components(string)))


def least_simultaneous_resources(strings):
    """Take a list of strings and return the list sorted by
       whish uses the least simultaneous resources"""
    min_set = set([strings[0]])
    min_len = most_simultaneous_events_occurring(strings[0])
    for s in strings[1:]:
        l = most_simultaneous_events_occurring(s)
        if l < min_len:
            min_len = l
            min_set = set([s])
        elif l == min_len:
            min_set.add(s)
    return sorted(min_set, key=lambda s: string_length(s))


def to_semiintervals(string, keep_fluents=False):
    """Take a string containing interval fluents, return its
       translation to use semi-intervals"""
    vocab = vocabulary(string)
    pp = map(lambda v: ('α({})'.format(v),'ω({})'.format(v)),
        vocab)
    lookup = { v: p for v, p in zip(vocab, pp) }
    used = []
    result = []
    for c in get_components(string):
        used += [f for f in c if f != '']
        # pre if fluent not occurred + post if occurred
        new_c = [lookup[k][0] for k in lookup if k not in
            used and k not in c] + [lookup[k][1] for k in
            lookup if k in used and k not in c]
        result.append(new_c if not keep_fluents else new_c +
            [f for f in c if f != ''])
    return '|'.join([','.join(c) for c in result])


def from_semiintervals(string):
    """Take a string containing semi-interval fluents, return
```

```python
    its translation to use intervals"""
vocab = set(re.findall(r'[αω]\((\w+)\)', string))
pp = map(lambda v: ('α({})'.format(v),'ω({})'.format(v)),
    vocab)
lookup = { v: p for v, p in zip(vocab, pp) }
used = []
result = []
for c in get_components(string):
    used += [k for k in lookup if lookup[k][1] in c]
    result.append([k for k in lookup if k not in used and
        lookup[k][0] not in c])
return '|'.join([','.join(c) for c in result])
```

```python
                        """freksa.py"""
from http.server import BaseHTTPRequestHandler
import json


# Allens
def equals(x, y):
    return ['|' + x + ',' + y + '|']
def before(x, y):
    return ['|' + x + '||' + y + '|']
def after(x, y):
    return before(y, x)
def meets(x, y):
    return ['|' + x + '|' + y + '|']
def meets_inv(x, y):
    return meets(y, x)
def starts(x, y):
    return ['|' + x + ',' + y + '|' + y + '|']
def starts_inv(x, y):
    return starts(y, x)
def finishes(x, y):
    return ['|' + y + '|' + x + ',' + y + '|']
def finishes_inv(x, y):
    return finishes(y, x)
def during(x, y):
    return ['|' + y + '|' + x + ',' + y + '|' + y + '|']
def during_inv(x, y):
    return during(y, x)
def overlaps(x, y):
    return ['|' + x + '|' + x + ',' + y + '|' + y + '|']
def overlaps_inv(x, y):
    return overlaps(y, x)


# Freksa
def older(x, y):
    return fi(x,y) + di(x,y) + m(x,y) + b(x,y) + o(x,y)
def younger(x, y):
    return older(y, x)
def head_to_head(x, y):
    return s(x, y) + si(x, y) + e(x, y)
```

```python
def tail_to_tail(x, y):
    return f(x, y) + fi(x, y) + e(x, y)
def survived_by(x, y):
    return b(x, y) + m(x, y) + o(x, y) + s(x, y) + d(x, y)
def survives(x, y):
    return survived_by(y, x)
def precedes(x, y):
    return b(x, y) + m(x, y)
def succeeds(x, y):
    return precedes(y, x)
def contemporary(x, y):
    return o(x, y) + fi(x, y) + di(x, y) + si(x, y) + e(x, y)
        + s(x, y) + d(x, y) + f(x, y) + oi(x, y)
def born_before_death(x, y):
    return precedes(x, y) + contemporary(x, y)
def died_after_birth(x, y):
    return born_before_death(y, x)
def older_survived_by(x, y):
    return precedes(x, y) + o(x, y)
def younger_survives(x, y):
    return older_survived_by(y, x)
def older_contemporary(x, y):
    return o(x, y) + fi(x, y) + di(x, y)
def younger_contemporary(x, y):
    return older_contemporary(y, x)
def surviving_contemporary(x, y):
    return di(x, y) + si(x, y) + oi(x, y)
def survived_by_contemporary(x, y):
    return surviving_contemporary(y, x)
def unknown(x, y):
    return precedes(x, y) + contemporary(x, y) + succeeds(x,
        y)


# Mnemonics
def e(x, y):
    return equals(x, y)
def b(x, y):
    return before(x, y)
def bi(x, y):
```

```python
        return after(x, y)
def m(x, y):
        return meets(x, y)
def mi(x, y):
        return meets_inv(x, y)
def s(x, y):
        return starts(x, y)
def si(x, y):
        return starts_inv(x, y)
def f(x, y):
        return finishes(x, y)
def fi(x, y):
        return finishes_inv(x, y)
def d(x, y):
        return during(x, y)
def di(x, y):
        return during_inv(x, y)
def o(x, y):
        return overlaps(x, y)
def oi(x, y):
        return overlaps_inv(x, y)


def un(x, y):
        return unknown(x, y)
def ol(x, y):
        return older(x, y)
def hh(x, y):
        return head_to_head(x, y)
def yo(x, y):
        return younger(x, y)
def sb(x, y):
        return survived_by(x, y)
def tt(x, y):
        return tail_to_tail(x, y)
def sv(x, y):
        return survives(x, y)
def pr(x, y):
        return precedes(x, y)
def bd(x, y):
```

```python
        return born_before_death(x, y)
def ct(x, y):
    return contemporary(x, y)
def db(x, y):
    return died_after_birth(x, y)
def sd(x, y):
    return succeeds(x, y)
def ob(x, y):
    return older_survived_by(x, y)
def oc(x, y):
    return older_contemporary(x, y)
def sc(x, y):
    return surviving_contemporary(x, y)
def bc(x, y):
    return survived_by_contemporary(x, y)
def yc(x, y):
    return younger_contemporary(x, y)
def ys(x, y):
    return younger_survives(x, y)


class handler(BaseHTTPRequestHandler):

    def do_POST(self):
        content_length = int(self.headers['Content-Length'])
        post_data = self.rfile.read(content_length).decode('
            utf-8')
        passed_data = json.loads(post_data)['data']
        result = dict()
        try:
            result['strings'] = globals()[passed_data['rel'
                ]](passed_data['e1'], passed_data['e2'])
        except Exception as e:
            result['error'] = str(e)

        self.send_response(200)
        self.send_header('Content-type', 'application/json')
        self.end_headers()
        self.wfile.write(json.dumps(result).encode('utf-8'))
        return
```

```python
                         """newTLINKs.py"""
from http.server import BaseHTTPRequestHandler
import apiscripts.strfns
import json


def string_to_rel(string, x):
    t, m = apiscripts.strfns.analogous_strings(string, '|X||Y
        |')
    if t:
        return 'BEFORE' if m[x] == 'X' else 'AFTER'
    t, m = apiscripts.strfns.analogous_strings(string, '|X|Y|
        ')
    if t:
        return 'IBEFORE' if m[x] == 'X' else 'IAFTER'
    t, m = apiscripts.strfns.analogous_strings(string, '|X,Y|
        ')
    if t:
        return 'SIMULTANEOUS'
    t, m = apiscripts.strfns.analogous_strings(string, '|X|X,
        Y|X|')
    if t:
        return 'INCLUDES' if m[x] == 'X' else 'DURING'
    t, m = apiscripts.strfns.analogous_strings(string, '|X|X,
        Y|')
    if t:
        return 'ENDED_BY' if m[x] == 'X' else 'ENDS'
    t, m = apiscripts.strfns.analogous_strings(string, '|X,Y|
        X|')
    if t:
        return 'BEGUN_BY' if m[x] == 'X' else 'BEGINS'
    t, m = apiscripts.strfns.analogous_strings(string, '|X|X,
        Y|Y|')
    if t:
        return 'OVERLAPS' if m[x] == 'X' else 'OVERLAPPED_BY'
    return 'UNKNOWN'


class handler(BaseHTTPRequestHandler):
    def do_POST(self):
        content_length = int(self.headers['Content-Length'])
```

```python
post_data = self.rfile.read(content_length).decode('
    utf -8')
passed_data = json.loads(post_data)['data']
result = dict()
try:
    vocab = passed_data['vocabulary']
    strings = None
    try:
        strings = frozenset(apiscripts.strfns.
            flatten_list(apiscripts.strfns.
            superpose_all_langs_sensible(passed_data['
            strings'], 12)))
    except:
        strings = frozenset(apiscripts.strfns.
            flatten_list(passed_data['strings']))
    idx = 0
    lid = 0
    links = []
    for v in vocab:
        for w in vocab[idx+1:]:
            p = apiscripts.strfns.
                projection_lang_full_vocab(strings, [v
                , w])
            if len(p) > 0:
                r = '|'.join(map(lambda s:
                    string_to_rel(s, v), p))
                links.append('<TLINK␣lid="{0}"␣
                    {1}="{2}"␣{3}="{4}"␣relType="{5}"␣
                    />'.format(lid, 'eventID' if v[0]
                    == 'e' else 'timeID', v, '
                    relatedToEvent' if w[0] == 'e'
                    else 'relatedToTime', w, r))
                lid += 1
        idx += 1
    result['tlinks'] = links
except Exception as e:
    result['error'] = str(e)

self.send_response(200)
```

```python
self.send_header('Content-type', 'application/json')
self.end_headers()
self.wfile.write(json.dumps(result).encode('utf-8'))
return
```

```python
"""superpose.py"""
from http.server import BaseHTTPRequestHandler
import apiscripts.strfns
import json


class handler(BaseHTTPRequestHandler):

    def do_POST(self):
        content_length = int(self.headers['Content-Length'])
        post_data = self.rfile.read(content_length).decode('
            utf-8')
        passed_data = json.loads(post_data)['data']
        result = dict()
        try:
            result['strings'] = list(apiscripts.strfns.
                superpose_all_langs_sensible(passed_data['
                strings'], passed_data['limit']))
        except Exception as e:
            result['error'] = str(e)

        self.send_response(200)
        self.send_header('Content-type', 'application/json')
        self.end_headers()
        self.wfile.write(json.dumps(result).encode('utf-8'))
        return
```

```python
"""test.py"""
from http.server import BaseHTTPRequestHandler
import apiscripts.freksa
import apiscripts.strfns
import json


class handler(BaseHTTPRequestHandler):

    def do_POST(self):
        content_length = int(self.headers['Content-Length'])
        post_data = self.rfile.read(content_length).decode('
            utf-8')
        passed_data = json.loads(post_data)['data']
        kb = passed_data['strings']
        result = dict()
        try:
            test_strings = getattr(apiscripts.freksa,
                passed_data['rel'])(passed_data['e1'],
                passed_data['e2'])
            result['strings'] = test_strings
            result['status'] = 'contradicted'
            if not any(any(apiscripts.strfns.
                lang_contradicts_string(lang, ts) for lang in
                kb) for ts in test_strings):
                result['status'] = 'possible'
                if any(any(apiscripts.strfns.
                    lang_contains_string(lang, ts) for lang in
                     kb) for ts in test_strings):
                    result['status'] = 'found'
        except Exception as e:
            result['error'] = str(e)

        self.send_response(200)
        self.send_header('Content-type', 'application/json')
        self.end_headers()
        self.wfile.write(json.dumps(result).encode('utf-8'))
        return
```

JavaScript Code

```
                        """index.js"""
import { useState, useEffect, useCallback, useReducer } from
    'react'
import Head from 'next/head'

import { parseTML } from '../fns/parseTML'
import * as freksa from '../fns/freksa'
import { suggestTenseAspectRelation } from '../fns/
    suggestTenseAspectRelation'

import Details from '../components/Details'
import Help from '../components/Help'
import ExamineString from '../components/ExamineString'
import StringBank from '../components/StringBank'
import CreateRelation from '../components/CreateRelation'
import TextEntry from '../components/TextEntry'
import TextDisplay from '../components/TextDisplay'
import EventList from '../components/EventList'

export default function Annotate() {
  const [helpDisplayed, setHelpDisplayed] = useState(false)
  const [details, setDetails] = useState(null)
  const [superposeLimit, setSuperposeLimit] = useState(12)
  const [extendedRels, setExtendedRels] = useState(false)
  const [noHighlight, setNoHighlight] = useState(false)
  const [examineStringDisplay, setExamineStringDisplay] =
      useState(null)

  const initialState = {
    textHTML: '',
    parsedEvents: [],
    eventStrings: [],
    nextId: 1,
    prevStates: []
  }

  // helps to set the reducer's initial state
```

```
const init = initialObj => ({
  ...initialObj
})


// current state + action => next state
const [state, dispatch] = useReducer((curState, action) =>
  {
  // make sure nextId always gets a unique value
  const getLastId = (evs, curId) => {
    let sortedIds = evs.map(ev => parseInt(ev.id.substring
      (1)))
    sortedIds.sort((a, b) => a - b)
    return (sortedIds[sortedIds.length - 1] || curId)
  }
  switch (action.type) {
    case 'SET_TEXT':
      return {
        ...curState,
        textHTML: action.payload,
        prevStates: [{...curState, prevStates: []}, ...
          curState.prevStates.slice(0, 50)]
      }
    case 'SET_EVENTS':
      return {
        ...curState,
        parsedEvents: action.payload,
        nextId: getLastId(action.payload, curState.nextId)
          + 1,
        prevStates: [{...curState, prevStates: []}, ...
          curState.prevStates.slice(0, 50)]
      }
    case 'ADD_EVENT':
      return {
        ...curState,
        parsedEvents: [...curState.parsedEvents, action.
          payload.event],
        textHTML: action.payload.newText,
        nextId: getLastId([...curState.parsedEvents, action
          .payload.event], curState.nextId) + 1,
```

```
        prevStates: [{...curState, prevStates: []}, ...
            curState.prevStates.slice(0, 50)]
    }
case 'REMOVE_EVENT':
    return {
        ...curState,
        parsedEvents: curState.parsedEvents.filter(e => e.
            id !== action.payload.id),
        eventStrings: curState.eventStrings.map(lan => lan.
            filter(s => !s.split(/[,|]/).includes(action.
            payload.id))).filter(lan => lan.length > 0),
        textHTML: action.payload.newText,
        prevStates: [{...curState, prevStates: []}, ...
            curState.prevStates.slice(0, 50)]
    }
case 'UPDATE_EVENT':
    return {
        ...curState,
        parsedEvents: curState.parsedEvents.map(e => e.id
            === action.payload.id ? {...e, attr: {...action.
            payload.newAttribs}} : e),
        prevStates: [{...curState, prevStates: []}, ...
            curState.prevStates.slice(0, 50)]
    }
case 'SET_STRINGS':
    return {
        ...curState,
        eventStrings: action.payload,
        prevStates: [{...curState, prevStates: []}, ...
            curState.prevStates.slice(0, 50)]
    }
case 'ADD_STRINGS':
    return {
        ...curState,
        eventStrings: [action.payload, ...curState.
            eventStrings],
        prevStates: [{...curState, prevStates: []}, ...
            curState.prevStates.slice(0, 50)]
    }
```

```
case 'REMOVE_STRINGS':
  return {
    ...curState,
    eventStrings: curState.eventStrings.map(lan => lan.
      filter(s => !action.payload.includes(s))).filter
      (lan => lan.length > 0),
    prevStates: [{...curState, prevStates: []}, ...
      curState.prevStates.slice(0, 50)]
  }
case 'DO_PARSE':
  if (action.payload.trim() !== '') {
    const parseResult = parseTML(action.payload)
    // try to parse the input as TimeML, else assume
      plaintext
    if (parseResult.transformed) {
      return {
        ...curState,
        textHTML: parseResult.transformed,
        parsedEvents: [...parseResult.events],
        eventStrings: parseResult.tlinks.map(tlink => {
          return tlink.map(t => {
            // usually means e1 === e2
            if (t.warning) {
              window.alert(`Bad TLINK: ${t.e1} ${t.rel}
                ${t.e2}`)
              return []
            }
            return freksa[t.rel](t.e1, t.e2)
          }).flat()
        }).filter(l => l.length > 0),
        nextId: getLastId([...parseResult.events],
          curState.nextId) + 1,
        prevStates: [{...curState, prevStates: []}, ...
          curState.prevStates.slice(0, 50)]
      }
    } else {
      return {
        ...curState,
        textHTML: parseResult.imported,
```

```
          prevStates: [{...curState, prevStates: []}, ...
            curState.prevStates.slice(0, 50)]
        }
      }
    }
  case 'UNDO':
    // go to the last state if possible
    if (curState.prevStates.length > 0) {
      const [cur, ...pre] = curState.prevStates
      return {
        ...cur,
        prevStates: pre
      }
    } else {
      return {
        ...curState
      }
    }
  case 'RESET':
    return init(action.payload)
  default:
    throw new Error('Undefined action type.')
  }
}, initialState, init)

// handle keyboard listening
useEffect(() => {
  document.addEventListener('keyup', handleKeyboard, false)
    return () => {
      document.removeEventListener('keyup', handleKeyboard,
        false)
    }
}, [helpDisplayed, details, state.nextId])

const handleKeyboard = useCallback(e => {
  const kb = e.key.toLowerCase()
    if (kb === 'e') {
      createMark('EVENT')
    } else if (kb === 't') {
```

```
        createMark('TIMEX3')
      } else if (kb === 'u') {
        dispatch({type: 'UNDO'})
      } else if (kb === 's') {
        document.getElementById('dosp').click()
      } else if (kb === '?') {
        // close other overlays before opening help
        setDetails(null)
        setExamineStringDisplay(null)
        setHelpDisplayed(!helpDisplayed)
      } else if (kb === 'escape') {
        setHelpDisplayed(false)
        setDetails(null)
        setExamineStringDisplay(null)
      }
}, [helpDisplayed, details, state.nextId])

const dismissOverlay = e => {
  e.stopPropagation()
  if (e.currentTarget === e.target) {
    setDetails(null)
    setHelpDisplayed(false)
    setExamineStringDisplay(null)
  }
}

// only highlight events under the mouse
const hoverLang = vocab => {
  if (!!vocab) {
    setNoHighlight(true)
    vocab.forEach(v => {
      document.querySelector('.tml-ev-ano[data-id="${v}"]')
        .classList.add('highlight')
    })
  } else {
    setNoHighlight(false)
    document.querySelectorAll('.tml-ev-ano').forEach(e => e
      .classList.remove('highlight'))
  }
```

```javascript
}

// view TLINK etc
const examineString = async string => {
  const vocab = [...new Set(string.split(/[,|]+/).filter(v
    => v !== ''))]
  const newString = string.split('|').map(c => c.split(',')
    .map(e => state.parsedEvents.find(ev => ev.id === e)?.
    text.replace(/\s+/, '_')).join(','))
  const res = await fetch(process.env.
    NEXT_PUBLIC_NEW_TLINKS_ENDPOINT, {
    method: 'POST',
    headers: {
      'Accept': 'application/json',
      'Content-Type': 'application/json'
    },
    body: JSON.stringify({
      data: {
        vocabulary: vocab,
        strings: [[string]]
      }
    })
  })
  const data = await res.json()
  if (data.error) {
    console.error(data)
    return []
  }
  setExamineStringDisplay({orig: string, string: newString,
    tlinks: data.tlinks})
}

const getNewTLINKs = async () => {
  const res = await fetch(process.env.
    NEXT_PUBLIC_NEW_TLINKS_ENDPOINT, {
    method: 'POST',
    headers: {
      'Accept': 'application/json',
      'Content-Type': 'application/json'
```

```
      },
      body: JSON.stringify({
        data: {
          vocabulary: state.parsedEvents.map(e => e.id),
          strings: state.eventStrings
        }
      })
    })
    const data = await res.json()
    if (data.error) {
      console.error(data)
      window.error(data)
      return []
    }
    return data.tlinks
  }


  // tell the user what relation exists in the KB for a given
      event pair
  const tryFindRelation = async (e1, e2) => {
    const res = await fetch(process.env.
      NEXT_PUBLIC_NEW_TLINKS_ENDPOINT, {
      method: 'POST',
      headers: {
        'Accept': 'application/json',
        'Content-Type': 'application/json'
      },
      body: JSON.stringify({
        data: {
          vocabulary: [e1, e2],
          strings: state.eventStrings
        }
      })
    })
    const data = await res.json()
    if (data.error) {
      console.error(data)
      return []
    }
```

```javascript
    const found = data.tlinks?.[0]?.match(/relType="([A-Z|_
      ]+)"/)?.[1].split('|');
  if (found) {
    window.alert(`The following possible relations were
      found:\n${found.map(f => e1 + ' ' + f + ' ' + e2).
      join(',\n')}`)
  } else {
    const ev1 = state.parsedEvents.find(e => e.id === e1)
    const ev2 = state.parsedEvents.find(e => e.id === e2)

    // try to suggest a relation based on Derczynski (2013)
    const suggestion = suggestTenseAspectRelation({tense1:
      ev1.attr.tense, aspect1: ev1.attr.aspect}, {tense2:
      ev2.attr.tense, aspect2: ev2.attr.aspect})
    if (!suggestion || suggestion === 'un') {
      window.alert('That relation could not be determined
        from current knowledge.')
    } else {
      const suggestionStrings = freksa[suggestion](e1, e2)
      window.alert(`No relations were found, but the ${
        suggestion.toUpperCase()} relation (see Help for
        extended relation types) is suggested based on the
         tenses and aspects of the events. Please exercise
         judgement whether this suggestion seems accurate$
        {ev1.attr.class === 'REPORTING' || ev2.attr.class
        === 'REPORTING' ? ', as it appears that these
        events may have different temporal contexts' : ''
        }. Strings:\n${suggestionStrings.join(',\n')}`)

    }
  }
}


// see if the string is entailed in the KB
const testRelation = async (e1, e2, rel) => {
  fetch(process.env.NEXT_PUBLIC_TEST_ENDPOINT, {
    method: 'POST',
    headers: {
      'Accept': 'application/json',
```

```
        'Content-Type': 'application/json'
      },
      body: JSON.stringify({
        data: {
          strings: state.eventStrings,
          e1,
          e2,
          rel
        }
      })
    }).then(response => response.json()).then(data => {
      if (data.error) {
        console.error(data.error)
        window.alert(data.error)
        return
      }
      const { status, strings } = data
      window.alert(`That relation is ${status} according to
          the knowledge base.${status !== 'contradicted' ? `\
          nClick 'Add New Relation' to add:\n${strings}` : `\
          nAdding this relation is not recommended.'}`)
    }).catch(e => console.error(e))
}


// convert START data back into TimeML
const exportTML = async elem => {
  const newTLINKs = await getNewTLINKs()
  const newText = elem.current.cloneNode(true)
  newText.querySelectorAll('.EVENT').forEach(node => {
    const restAttr = Object.entries(state.parsedEvents.find
        (e => e.id === node.dataset.id).attr).map(a => `${a
        [0]}="${a[1]}"`).join(' ');
    node.replaceWith(document.createTextNode(`<EVENT eid="$
        {node.dataset.id}" ${restAttr}>${node.textContent}</
        EVENT>`))
  })
  newText.querySelectorAll('.TIMEX3').forEach(node => {
    const restAttr = Object.entries(state.parsedEvents.find
        (e => e.id === node.dataset.id).attr).map(a => `${a
```

```
          [0]}="${a[1]}"').join('␣');
        node.replaceWith(document.createTextNode('<TIMEX3 tid="
          ${node.dataset.id}" ${restAttr}>${node.textContent
          }</TIMEX3>'))
      })
      return '<TimeML xmlns:xsi="http://www.w3.org/2001/
        XMLSchema-instance" xsi:noNamespaceSchemaLocation="
        http://timeml.org/timeMLdocs/TimeML_1.2.1.xsd">
${newText.textContent}
${newTLINKs.join('\n')}
</TimeML>'
  }

  const downloadTML = async elem => {
    const dlElem = document.createElement('a')

    dlElem.setAttribute('href', 'data:text/xml;charset=utf-8,
      ' + encodeURIComponent(await exportTML(elem)))
    dlElem.setAttribute('download', 'export.tml')
    dlElem.style.display = 'none'

    document.body.appendChild(dlElem)
    dlElem.click()
    document.body.removeChild(dlElem)
  }

  // tag the selected (highlighted) text as an event or time
  const createMark = useCallback(tag => {
    if (window.getSelection) {
      const sel = window.getSelection()
      if (sel.type === 'Range' && sel.anchorNode === sel.
        focusNode && sel.anchorNode.parentElement ===
        document.querySelector('.text␣pre') && sel.toString
        ().trim() !== '') {
        if (sel.rangeCount) {
          const wrapEl = document.createElement('span')
          wrapEl.classList.add('tml-ev-ano', tag)
          if (tag === 'EVENT') {
            wrapEl.dataset.id = 'e${state.nextId}'
```

```
          wrapEl.dataset.eventClass = 'OCCURRENCE'
        } else {
          wrapEl.dataset.id = `t${state.nextId}`
          wrapEl.dataset.type = 'DATE'
        }
        wrapEl.textContent = sel.toString()
        const range = sel.getRangeAt(0).cloneRange()
        range.deleteContents()
        range.insertNode(wrapEl)
        sel.removeAllRanges()
        sel.addRange(range)
        dispatch({type: 'ADD_EVENT', payload: {
          event: {
            id: wrapEl.dataset.id,
            type: tag,
            text: wrapEl.textContent,
            elem: wrapEl,
            attr: {}
          },
          newText: sel.anchorNode.parentElement.innerHTML}
        })
      }
    }
  }
}, [state.nextId])

// remove the highlighting element
const removeMark = useCallback(id => {
  const ev = document.querySelector(`[data-id="${id}"].tml-
    ev-ano`)
  const textNode = document.createTextNode(ev.textContent)
  const par = ev.parentElement
  ev.replaceWith(textNode)
  par.normalize()
  dispatch({type: 'REMOVE_EVENT', payload: {id, newText:
    par.innerHTML}})
})

return (
```

```
<main id="annotate">
  <Head><title>START (String Temporal Annotation and
    Relation Tool)</title></Head>
  {helpDisplayed && <Help extendedRels={extendedRels}
    setExtendedRels={setExtendedRels} dismiss={
    dismissOverlay} limit={superposeLimit} setLimit={
    setSuperposeLimit}/>}
  {details && <Details event={state.parsedEvents.find(e
    => e.id === details)} dismiss={dismissOverlay}
    update={(id, newAttribs) => dispatch({type: '
    UPDATE_EVENT', payload: {id, newAttribs}})} />}
  {examineStringDisplay && <ExamineString data={
    examineStringDisplay} dismiss={dismissOverlay}
    removeString={s => dispatch({type: 'REMOVE_STRINGS',
     payload: [s]})}/>}
  {!state.textHTML ? <TextEntry grabParse={val =>
    dispatch({type: 'DO_PARSE', payload: val})}/> : <
    TextDisplay noHighlight={noHighlight} text={state.
    textHTML} reset={() => dispatch({type: 'SET_TEXT',
    payload: ''})} download={downloadTML}/>}
  <div className="panel">
    <p>Select some text, then tag as Event or Time</p>
    <div className="btns">
      <button id="btn-help" onClick={() =>
        setHelpDisplayed(true)}>Help</button>
      <button id="btn-undo" onClick={() => dispatch({type
        : 'UNDO'})}>Undo</button>
      <button id="btn-reset" onClick={() => window.
        confirm('Are␣you␣sure?␣No␣undo.') && dispatch({
        type: 'RESET', payload: initialState})}>Reset</
        button>
      <button id="btn-tag-event" onClick={() =>
        createMark('EVENT')}>Tag Event</button>
      <button id="btn-tag-time" onClick={() => createMark
        ('TIMEX3')}>Tag Time</button>
    </div>
    <EventList events={state.parsedEvents} remove={
      removeMark} edit={setDetails} hoverLang={hoverLang
      } />
```

```
        </div>
        <StringBank hoverLang={hoverLang} limit={superposeLimit
           } strings={state.eventStrings} updateStrings={res =>
            dispatch({type: 'SET_STRINGS', payload: [...res]})}
            examineString={examineString}/>
        {state.parsedEvents.length > 0 ? <CreateRelation
           extendedRels={extendedRels} events={state.
           parsedEvents} addRelation={res => dispatch({type: '
           ADD_STRINGS', payload: res})} testRelation={
           testRelation} tryFindRelation={tryFindRelation}/> :
           <div className="relations"></div>}
    </main>
  )
}
```

```
                        """parseTML.js"""
export const parseTML = inputString => {
    const parser = new window.DOMParser()
    const xml = parser.parseFromString(inputString.trim(), '
      text/xml')

    if (xml.documentElement.nodeName == "parsererror" || xml.
      documentElement.getElementsByTagName('parsererror').
      length > 0) {
      return {
        imported: inputString
      }
    }
    let nodes = []
    if (xml.firstChild.getElementsByTagName('TEXT').length
      === 0) {
      nodes = xml.firstChild.childNodes
    } else {
      nodes = [...xml.firstChild.getElementsByTagName('DCT')
        [0].childNodes, ...xml.firstChild.
        getElementsByTagName('TEXT')[0].childNodes]
    }

    const instances = [...xml.getElementsByTagName('
      MAKEINSTANCE')].map(mi => {
      const {eventID, eiid, ...rest} = Object.assign({}, ...
        Array.from(mi.attributes, ({name, value}) => ({[name
        ]: value})));
      return {
        eventID,
        eiid,
        rest
      }
    })

    const pieces = []
    const events = []
    for (let node of nodes) {
        if (node.nodeName === 'EVENT') {
```

```javascript
    const nodeAttr = Object.assign({}, ...Array.from(
      node.attributes, ({name, value}) => ({[name]:
      value})));
    const span = document.createElement('span')
    span.classList.add('tml-ev-ano', 'EVENT')
    span.textContent = node.textContent
    const {eid, ...nodeRest} = nodeAttr
    span.dataset.id = eid
    const inst = instances.find(i => i.eventID ===
       eid)
    events.push({
      id: eid,
      type: 'EVENT',
      text: span.textContent,
      elem: span,
      attr: inst !== undefined ? {...nodeRest, ...
         inst.rest} : {...nodeRest}
    })
    pieces.push(span.outerHTML)
} else if (node.nodeName === 'TIMEX3') {
    const nodeAttr = Object.assign({}, ...Array.from(
      node.attributes, ({name, value}) => ({[name]:
      value})));
    const span = document.createElement('span')
    span.classList.add('tml-ev-ano', 'TIMEX3')
    span.textContent = node.textContent
    const {tid, ...nodeRest} = nodeAttr
    span.dataset.id = tid
    events.push({
      id: tid,
      type: 'TIMEX3',
      text: span.textContent,
      elem: span,
      attr: {...nodeRest}
    })
    pieces.push(span.outerHTML)
} else {
    pieces.push(node.textContent)
}
```

```
        }

        const tlinks = [...xml.getElementsByTagName('TLINK')].map
            (tlink => {
          const { eventID, eventInstanceID, timeID,
              relatedToEvent, relatedToEventInstance,
              relatedToTime, relType } = tlink.attributes
          const e1 = eventID ? eventID.value : instances.find(i
              => eventInstanceID !== undefined && i.eiid ===
              eventInstanceID.value)?.eventID ?? timeID.value
          const e2 = relatedToEvent ? relatedToEvent.value :
              instances.find(i => relatedToEventInstance !==
              undefined && i.eiid === relatedToEventInstance.value
              )?.eventID ?? relatedToTime.value
          return relType.value.split('|').map(r => ({
            rel: tmlToAllen[r],
            e1,
            e2,
            warning: e1 === e2
          }))
        })
        return {
            imported: inputString,
            transformed: pieces.join('').trim(),
            text: xml.documentElement.textContent.trim(),
            events,
            tlinks
        }
}

export const tmlToAllen = {
  BEFORE: 'b',
  AFTER: 'bi',
  INCLUDES: 'di',
  DURING_INV: 'di',
  IS_INCLUDED: 'd',
  DURING: 'd',
  SIMULTANEOUS: 'e',
  IDENTITY: 'e',
```

```
    IAFTER: 'mi',
    IBEFORE: 'm',
    BEGINS: 's',
    ENDS: 'f',
    BEGUN_BY: 'si',
    ENDED_BY: 'fi',
    OVERLAPS: 'o',
    OVERLAPPED_BY: 'oi'
}
```

```
                    """"suggestTenseAspectRelation.js"""
export function suggestTenseAspectRelation(e1, e2) {
  const {tense1, aspect1} = e1
  const {tense2, aspect2} = e2
  if (!tense1 || !tense2 || tense1 === 'NONE' || tense2 === '
     NONE') {
    return 'un'
  }
  return lookup[`${tense1}-${aspect1 === '
     PERFECTIVE_PROGRESSIVE' ? 'PREFECTIVE' : aspect1}_${
     tense2}-${aspect2 === 'PERFECTIVE_PROGRESSIVE' ? '
     PREFECTIVE' : aspect2}`]
}


const lookup = {
  'PAST-NONE_PAST-NONE': 'un',
  'PAST-NONE_PAST-PROGRESSIVE': 'contemprary',
  'PAST-NONE_PAST-PERFECTIVE': 'succeeds',
  'PAST-NONE_PRESENT-NONE': 'survived_by',
  'PAST-NONE_PRESENT-PROGRESSIVE': 'survived_by',
  'PAST-NONE_PRESENT-PERFECTIVE': 'un',
  'PAST-NONE_FUTURE-NONE': 'precedes',
  'PAST-NONE_FUTURE-PROGRESSIVE': 'survived_by',
  'PAST-NONE_FUTURE-PERFECTIVE': 'before',

  'PAST-PROGRESSIVE_PAST-NONE': 'contemprary',
  'PAST-PROGRESSIVE_PAST-PROGRESSIVE': 'contemprary',
  'PAST-PROGRESSIVE_PAST-PERFECTIVE': 'survives',
  'PAST-PROGRESSIVE_PRESENT-NONE': 'older',
  'PAST-PROGRESSIVE_PRESENT-PROGRESSIVE': 'un',
  'PAST-PROGRESSIVE_PRESENT-PERFECTIVE': 'un',
  'PAST-PROGRESSIVE_FUTURE-NONE': 'older',
  'PAST-PROGRESSIVE_FUTURE-PROGRESSIVE': 'born_before_death',
  'PAST-PROGRESSIVE_FUTURE-PERFECTIVE': 'older',

  'PAST-PERFECTIVE_PAST-NONE': 'precedes',
  'PAST-PERFECTIVE_PAST-PROGRESSIVE': 'survived_by',
  'PAST-PERFECTIVE_PAST-PERFECTIVE': 'un',
  'PAST-PERFECTIVE_PRESENT-NONE': 'precedes',
```

```
'PAST-PERFECTIVE_PRESENT-PROGRESSIVE': 'survived_by',
'PAST-PERFECTIVE_PRESENT-PERFECTIVE': 'precedes',
'PAST-PERFECTIVE_FUTURE-NONE': 'before',
'PAST-PERFECTIVE_FUTURE-PROGRESSIVE': 'survived_by',
'PAST-PERFECTIVE_FUTURE-PERFECTIVE': 'before',

'PRESENT-NONE_PAST-NONE': 'survives',
'PRESENT-NONE_PAST-PROGRESSIVE': 'younger',
'PRESENT-NONE_PAST-PERFECTIVE': 'succeeds',
'PRESENT-NONE_PRESENT-NONE': 'contemprary',
'PRESENT-NONE_PRESENT-PROGRESSIVE': 'contemprary',
'PRESENT-NONE_PRESENT-PERFECTIVE': 'survives',
'PRESENT-NONE_FUTURE-NONE': 'precedes',
'PRESENT-NONE_FUTURE-PROGRESSIVE': 'older',
'PRESENT-NONE_FUTURE-PERFECTIVE': 'older',

'PRESENT-PROGRESSIVE_PAST-NONE': 'survives',
'PRESENT-PROGRESSIVE_PAST-PROGRESSIVE': 'un',
'PRESENT-PROGRESSIVE_PAST-PERFECTIVE': 'survives',
'PRESENT-PROGRESSIVE_PRESENT-NONE': 'contemprary',
'PRESENT-PROGRESSIVE_PRESENT-PROGRESSIVE': 'contemprary',
'PRESENT-PROGRESSIVE_PRESENT-PERFECTIVE': 'survives',
'PRESENT-PROGRESSIVE_FUTURE-NONE': 'older',
'PRESENT-PROGRESSIVE_FUTURE-PROGRESSIVE': '
    born_before_death',
'PRESENT-PROGRESSIVE_FUTURE-PERFECTIVE': 'older',

'PRESENT-PERFECTIVE_PAST-NONE': 'un',
'PRESENT-PERFECTIVE_PAST-PROGRESSIVE': 'un',
'PRESENT-PERFECTIVE_PAST-PERFECTIVE': 'succeeds',
'PRESENT-PERFECTIVE_PRESENT-NONE': 'survived_by',
'PRESENT-PERFECTIVE_PRESENT-PROGRESSIVE': 'survived_by',
'PRESENT-PERFECTIVE_PRESENT-PERFECTIVE': 'un',
'PRESENT-PERFECTIVE_FUTURE-NONE': 'before',
'PRESENT-PERFECTIVE_FUTURE-PROGRESSIVE': 'survived_by',
'PRESENT-PERFECTIVE_FUTURE-PERFECTIVE': 'before',

'FUTURE-NONE_PAST-NONE': 'succeeds',
'FUTURE-NONE_PAST-PROGRESSIVE': 'younger',
```

```
    'FUTURE-NONE_PAST-PERFECTIVE': 'after',
    'FUTURE-NONE_PRESENT-NONE': 'succeeds',
    'FUTURE-NONE_PRESENT-PROGRESSIVE': 'younger',
    'FUTURE-NONE_PRESENT-PERFECTIVE': 'after',
    'FUTURE-NONE_FUTURE-NONE': 'un',
    'FUTURE-NONE_FUTURE-PROGRESSIVE': 'contemprary',
    'FUTURE-NONE_FUTURE-PERFECTIVE': 'survived_by',

    'FUTURE-PROGRESSIVE_PAST-NONE': 'survives',
    'FUTURE-PROGRESSIVE_PAST-PROGRESSIVE': 'died_after_birth',
    'FUTURE-PROGRESSIVE_PAST-PERFECTIVE': 'survives',
    'FUTURE-PROGRESSIVE_PRESENT-NONE': 'younger',
    'FUTURE-PROGRESSIVE_PRESENT-PROGRESSIVE': 'died_after_birth
        ',
    'FUTURE-PROGRESSIVE_PRESENT-PERFECTIVE': 'survives',
    'FUTURE-PROGRESSIVE_FUTURE-NONE': 'contemprary',
    'FUTURE-PROGRESSIVE_FUTURE-PROGRESSIVE': 'contemprary',
    'FUTURE-PROGRESSIVE_FUTURE-PERFECTIVE': 'survives',

    'FUTURE-PERFECTIVE_PAST-NONE': 'after',
    'FUTURE-PERFECTIVE_PAST-PROGRESSIVE': 'younger',
    'FUTURE-PERFECTIVE_PAST-PERFECTIVE': 'after',
    'FUTURE-PERFECTIVE_PRESENT-NONE': 'younger',
    'FUTURE-PERFECTIVE_PRESENT-PROGRESSIVE': 'younger',
    'FUTURE-PERFECTIVE_PRESENT-PERFECTIVE': 'after',
    'FUTURE-PERFECTIVE_FUTURE-NONE': 'survived_by',
    'FUTURE-PERFECTIVE_FUTURE-PROGRESSIVE': 'survived_by',
    'FUTURE-PERFECTIVE_FUTURE-PERFECTIVE': 'un',
}
```

```
                        """CreateRelation.js"""
import { useState , useEffect } from 'react'
import * as freksa from '../fns/freksa'

export default function CreateRelation(props) {
  const [ev1, setEv1] = useState('')
  const [ev2, setEv2] = useState('')
  const [rel, setRel] = useState('')

  const addNewRelation = () => {
    props.addRelation(freksa[rel](ev1, ev2))
  }

  useEffect(() => {
    setEv1('')
    setEv2('')
    setRel('')
  }, [props.extendedRels])

  return (
    <div className="relations">
      <ul>
        <li>
        <label htmlFor="ev1-select">Event 1:</label> <select
          id="ev1-select" value={ev1} className="ev1"
          onChange={e => setEv1(e.currentTarget.value)}>
          {ev1 === '' && <option disabled value=''>Select
            Event</option>}
          {props.events.map(ev => <option key={`ev1-${ev.id
            }`} value={ev.id}>{ev.id}</option>)}
        </select>
        </li>
        <li>
        <label htmlFor="rel-select">Rel:</label> <select
          value={rel} className="rel" onChange={e => setRel(
          e.currentTarget.value)} disabled={ev1 === ev2 ||
          ev1 === '' || ev2 === ''}>
          {rel === '' && <option disabled value=''>Select
            Relation</option>}
```

```
<option value="b">BEFORE (|{ev1}||{ev2}|)</option>
<option value="bi">AFTER (|{ev2}||{ev1}|)</option>
<option value="m">IBEFORE (|{ev1}|{ev2}|)</option>
{/* <option value="m">MEETING (|{ev1}|{ev2}|)</
    option> */}
<option value="mi">IAFTER (|{ev2}|{ev1}|)</option>
{/* <option value="mi">MET BY (|{ev2}|{ev1}|)</
    option> */}
<option value="s">BEGINS (|{ev1},{ev2}|{ev2}|)</
    option>
{/* <option value="s">STARTING (|{ev1},{ev2}|{ev2
    }|)</option> */}
<option value="si">BEGUN_BY (|{ev1},{ev2}|{ev1}|)</
    option>
{/* <option value="si">STARTED BY (|{ev1},{ev2}|{
    ev1}|)</option> */}
<option value="f">ENDS (|{ev2}|{ev1},{ev2}|)</
    option>
{/* <option value="f">FINISHING (|{ev2}|{ev1},{ev2
    }|)</option> */}
<option value="fi">ENDED_BY (|{ev1}|{ev1},{ev2}|)</
    option>
{/* <option value="fi">FINISHED BY (|{ev1}|{ev1},{
    ev2}|)</option> */}
<option value="di">INCLUDES (|{ev1}|{ev1},{ev2}|{
    ev1}|)</option>
<option value="d">IS_INCLUDED (|{ev2}|{ev1},{ev2}|{
    ev2}|)</option>
<option value="d">DURING (|{ev2}|{ev1},{ev2}|{ev2
    }|)</option>
<option value="di">DURING_INV (|{ev1}|{ev1},{ev2}|{
    ev1}|)</option>
{/* <option value="di">CONTAINING (|{ev1}|{ev1},{
    ev2}|{ev1}|)</option> */}
{/* <option value="d">CONTAINED BY (|{ev2}|{ev1},{
    ev2}|{ev2}|)</option> */}
{<option value="o">OVERLAPS (|{ev1}|{ev1},{ev2}|{
    ev2}|)</option>}
{<option value="oi">OVERLAPPED BY (|{ev2}|{ev1},{
```

```jsx
          ev2}|{ev1}|)</option>}
        <option value="e">SIMULTANEOUS (|{ev1},{ev2}|)</
          option>
        <option value="e">IDENTITY (|{ev1},{ev2}|)</option>
        {props.extendedRels && <option value="ol">OLDER</
          option>}
        {props.extendedRels && <option value="yo">YOUNGER</
          option>}
        {props.extendedRels && <option value="hh">
          HEAD_TO_HEAD</option>}
        {props.extendedRels && <option value="tt">
          TAIL_TO_TAIL</option>}
        {props.extendedRels && <option value="sv">SURVIVES
          </option>}
        {props.extendedRels && <option value="sb">
          SURVIVED_BY</option>}
        {props.extendedRels && <option value="pr">PRECEDES
          </option>}
        {props.extendedRels && <option value="sd">SUCCEEDS
          </option>}
        {props.extendedRels && <option value="bd">
          BORN_BEFORE_DEATH</option>}
        {props.extendedRels && <option value="db">
          DIED_AFTER_BIRTH</option>}
        {props.extendedRels && <option value="ct">
          CONTEMPORARY</option>}
        {props.extendedRels && <option value="ob">
          OLDER_SURVIVED_BY</option>}
        {props.extendedRels && <option value="oc">
          OLDER_CONTEMPORARY</option>}
        {props.extendedRels && <option value="sc">
          SURVIVING_CONTEMPORARY</option>}
        {props.extendedRels && <option value="bc">
          SURVIVED_BY_CONTEMPORARY</option>}
        {props.extendedRels && <option value="yc">
          YOUNGER_CONTEMPORARY</option>}
        {props.extendedRels && <option value="ys">
          YOUNGER_SURVIVES</option>}
      </select>
```

```
          </li>
          <li>
          <label htmlFor="ev2-select">Event 2:</label> <select
             value={ev2} className="ev2" onChange={e => setEv2(
             e.currentTarget.value)}>
            {ev2 === '' && <option disabled value=''>Select
               Event</option>}
            {props.events.map(ev => <option key={`ev2-${ev.id
               }`} value={ev.id}>{ev.id}</option>)}
          </select>
          </li>
        </ul>
        <button id="testRelation" disabled={ev1 === ev2 || ev1
           === '' || ev2 === '' || rel === ''} onClick={() =>
           props.testRelation(ev1, ev2, rel)}>Test Relation</
           button>
        <button id="createRelation" disabled={ev1 === ev2 ||
           ev1 === '' || ev2 === '' || rel === ''} onClick={
           addNewRelation}>Add New Relation</button>
        <button id="tryFindRelation" disabled={ev1 === ev2 ||
           ev1 === '' || ev2 === ''} onClick={() => props.
           tryFindRelation(ev1, ev2)}>Try Find Relation</button
           >
      </div>
    )
}
```

```
                    """Details.js"""
import { useState, useEffect, useRef } from 'react'

export default function Details(props) {
  const [attribs, setAttribs] = useState({...props.event.attr
    })
  const detailsRef = useRef(null)
  useEffect(() => {
    detailsRef.current && detailsRef.current.focus()
  }, [])


  return (
    <div className="overlay" onClick={props.dismiss}>
      <div className="add-details">
        <h2>{props.event.id}: {props.event.text}</h2>
        <h3>Edit attributes</h3>
        {props.event.type === 'EVENT' ? <ul className="attr-
          list">
          <li>
            <label htmlFor="class-select">Class:</label> <
              select ref={detailsRef} id="class-select"
              defaultValue={attribs.class} onChange={e =>
              setAttribs(prev => ({...prev, class: e.target.
              value}))}>
              <option value="OCCURRENCE">OCCURRENCE</option>
              <option value="PERCEPTION">PERCEPTION</option>
              <option value="REPORTING">REPORTING</option>
              <option value="ASPECTUAL">ASPECTUAL</option>
              <option value="STATE">STATE</option>
              <option value="I_STATE">I_STATE</option>
              <option value="I_ACTION">I_ACTION</option>
            </select>
          </li>
          <li>
            <label htmlFor="tense-select">Tense:</label> <
              select id="tense-select" defaultValue={attribs
              .tense} onChange={e => setAttribs(prev =>
              ({...prev, tense: e.target.value}))}>
              <option value="NONE">NONE</option>
```

```
            <option value="PAST">PAST</option>
            <option value="PRESENT">PRESENT</option>
            <option value="FUTURE">FUTURE</option>
          </select>
        </li>
        <li>
          <label htmlFor="aspect-select">Aspect:</label> <
            select id="aspect-select" defaultValue={
            attribs.aspect} onChange={e => setAttribs(prev
             => ({...prev, aspect: e.target.value}))}>
            <option value="NONE">NONE</option>
            <option value="PROGRESSIVE">PROGRESSIVE</option
              >
            <option value="PERFECTIVE">PERFECTIVE</option>
            <option value="PERFECTIVE_PROGRESSIVE">
              PERFECTIVE_PROGRESSIVE</option>
          </select>
        </li>
        <li>
          <label htmlFor="polarity-select">Polarity:</label
            > <select id="polarity-select" defaultValue={
            attribs.polarity} onChange={e => setAttribs(
            prev => ({...prev, polarity: e.target.value}))
            }>
            <option value="POS">POS</option>
            <option value="NEG">NEG</option>
          </select>
        </li>
    </ul>
    :
    <ul className="attr-list">
        <li>
          <label htmlFor="type-select">Type:</label> <
            select ref={detailsRef} id="type-select"
            defaultValue={attribs.type} onChange={e =>
            setAttribs(prev => ({...prev, type: e.target.
            value}))}>
            <option value="DATE">DATE</option>
            <option value="TIME">TIME</option>
```

```
        <option value="DURATION">DURATION</option>
        <option value="SET">SET</option>
      </select>
  </li>
  <li>
    <label htmlFor="function-in-doc-select">Function
      in doc:</label> <select id="function-in-doc-
      select" defaultValue={attribs.
      functionInDocument} onChange={e => setAttribs(
      prev => ({...prev, functionInDocument: e.
      target.value}))}>
      <option value="NONE">NONE</option>
      <option value="CREATION_TIME">CREATION_TIME</
        option>
      <option value="EXPIRATION_TIME">EXPIRATION_TIME
        </option>
      <option value="MODIFICATION_TIME">
        MODIFICATION_TIME</option>
      <option value="PUBLICATION_TIME">
        PUBLICATION_TIME</option>
      <option value="RELEASE_TIME">RELEASE_TIME</
        option>
      <option value="RECEPTION_TIME">RECEPTION_TIME</
        option>
    </select>
  </li>
  <li>
    <label htmlFor="temporal-func-select">Temporal
      function:</label> <select id="temporal-func-
      select" defaultValue={attribs.temporalFunction
      } onChange={e => setAttribs(prev => ({...prev,
        temporalFunction: e.target.value}))}>
      <option value="false">false</option>
      <option value="true">true</option>
    </select>
  </li>
  <li>
    <label htmlFor="value-input">Value:</label> <
      input id="value-input" defaultValue={attribs.
```

```
          value} onChange={e => setAttribs(prev => ({...
          prev, value: e.target.value}))}/>
        </li>
        <li>
          <label htmlFor="anchor-time-input">Anchor time ID
          :</label> <input id="anchor-time-input"
          defaultValue={attribs.anchorTimeID} onChange={
          e => setAttribs(prev => ({...prev,
          anchorTimeID: e.target.value}))}/>
        </li>
      </ul>
      }
      <button className="update-attr" onClick={e => {
        props.update(props.event.id, attribs)
        props.dismiss(e)
      }}>Update</button>
    </div>
  </div>
  )
}
```

```
                        """EventList.js"""
export default function EventList(props) {
  return (
    <ul className="event-list">
      {props.events.map(ev => <EventListItem key={ev.id}
        event={ev} remove={() => props.remove(ev.id)} edit
        ={() => props.edit(ev.id)} hover={props.hoverLang}
        />)}
    </ul>
  )
}


function EventListItem({event, remove, edit, hover}) {
  return (
    <li onMouseEnter={() => hover([event.id])} onMouseLeave
      ={() => hover()}>{event.id}: {event.text} <div style
      ={{display: 'flex'}}><button onClick={edit}>&#43;</
      button><button onClick={remove}>&times;</button></div
      ></li>
  )
}
```

```
"""ExamineString.js"""
export default function ExamineString(props) {
  return (
    <div className="overlay" onClick={props.dismiss}>
      <div className="examine-string">
       <div style={{overflowX: 'auto'}}>
        <button style={{marginBottom: '0.2em'}} onClick={(e)
          => {props.dismiss(e); props.removeString(props.
          data.orig)}}>Remove</button>
        <div className="source">{props.data.orig}</div>
        <div className="examined">{props.data.string.map((com
          , i) => <span key={i} className="component-box">{
          com === "" ? '␣' : com}</span>)}</div>
       </div>
       <ul className="tlinks">{props.data.tlinks.map((t, i)
         => <li key={i}>{t}</li>)}</ul>
      </div>
    </div>
  )
}
```

```
                        """Help.js"""
import { useState } from 'react'


export default function Help(props) {
  const [page, setPage] = useState(1)


  return (
    <div className="overlay" onClick={props.dismiss}>
        <div className="help">
          <h3>START (String Temporal Annotation and Relation
            Tool)</h3>
          {page === 1 && <div id="help-p1">
            <p>
              Use the mouse to highlight text, then press '
                Tag␣Event' to tag that text as an event, or
                press 'Tag␣Time' to tag that text as a time.
                 Tagged events and times will appear in the
                upper right panel, where they may be removed
                 or their attributes may be edited.
            </p>
            <p>
              Keyboard shortcuts:
            </p>
            <ul>
              <li>E: Tag Event</li>
              <li>T: Tag Time</li>
              <li>U: Undo</li>
              <li>S: Superpose</li>
              <li>?: Toggle Help</li>
            </ul>
            <a style={{cursor: 'pointer', float: 'right'}}
              onClick={() => setPage(2)}>More</a>
          </div>}
          {page === 2 && <div id="help-p2">
            <p>
              Test or create relations from the tagged events
                 and times in the lower left panel. Created
                relations will appear as strings in the
                String Bank. The string that will appear can
```

```
      be previewed to the right of the relation
      name when selecting a relation. You can also
       try to find the relations between a pair of
       temporal entities that are currently
    present in the String Bank.
</p>
<p>
  Check this box to allow Freksa relation input (
      these will not appear in the exported TimeML
      , click <a target="_blank" rel="noopener␣
      noreferrer" href="/freksa.html">here for
      more info</a>): <input type="checkbox"
      checked={props.extendedRels} onChange={e =>
      props.setExtendedRels(e.currentTarget.
      checked)} />
</p>
<p>
  The String Bank contains all of the known
      temporal relations, as strings. Triggering a
       superposition will attempt to consolidate
      this knowledge into a smaller number of
      strings by combining the data where it is
      sensible to do so, i.e. |a|b| + |b|c| = |a|b
      |c|. Since some superpositions may produce
      multiple strings as a result, the program
      will attempt to limit the resulting set to
      no more than <input style={{width: '6ch'}}
      type="number" min="0" max="99" defaultValue
      ={props.limit} onChange={e => props.setLimit
      (parseInt(e.currentTarget.value))}/> strings
       (use 0 for no limit - this may be slow and
      produce unwieldy results). Clicking on a
      string will show the <code>{'<TLINK>'}</code
      > tags derivable from it.
</p>
<a style={{cursor: 'pointer', float: 'left'}}
    onClick={() => setPage(1)}>Back</a>
<a style={{cursor: 'pointer', float: 'right'}}
    onClick={() => setPage(3)}>More</a>
```

```
        </div >}
        {page === 3 && <div id="help-p3">
          <p>
            Clicking Export will export the annotated text
                as a TimeML (v1.2) file. Note that <code>{'<
                MAKEINSTANCE>'}</code> tags are not output,
                and their attributes are shifted to the <
                code>{'<EVENT>'}</code> tags. <code>{'<TLINK
                >'}</code> tags use eventIDs instead of
                eventInstanceIDs, and the relType may
                include a disjunction of relations if
                multiple options were derivable from the
                String bank, e.g. relType="BEFORE|IBEFORE".
                The OVERLAPS and OVERLAPPED_BY relations are
                 also permitted in the output if they are
                derived.
          </p>
          <a style={{cursor: 'pointer', float: 'left'}}
              onClick={() => setPage(2)}>Back</a>
        </div >}
      </div >
    </div >
  )
}
```

```
                        """StringBank.js"""
import { useState } from 'react'
export default function StringBank(props) {
  const [loading, setLoading] = useState(false)

  const doSuperposition = async () => {
    setLoading(true)
    const res = await fetch(process.env.
      NEXT_PUBLIC_SUPERPOSE_ENDPOINT, {
      method: 'POST',
      headers: {
        'Accept': 'application/json',
        'Content-Type': 'application/json'
      },
      body: JSON.stringify({
        data: {
          strings: props.strings,
          limit: props.limit
        }
      })
    })
    setLoading(false)
    const data = await res.json()
    if (data.error) {
      console.error(data.error)
      window.alert(data.error)
    } else {
      props.updateStrings(data.strings)
    }
  }

  return (
    <div className="string-bank">
      <h4>String bank ({props.strings.length}) <button
        disabled={loading || props.strings.length < 2} id="
        dosp" onClick={doSuperposition}>Superpos{loading ? '
        ing' : 'e'}</button></h4>
      <ul className="strings">
      {props.strings.map((l, i) => <Language key={`sb-${i}`}
```

```
          langid ={i} language ={l} hoverLang ={ props.hoverLang }
          examineString ={ props.examineString } />)}
      </ul>
    </div>
  )
}


function Language ( props ) {
  const vocab = [... new Set ( props.language.reduce (( acc , cur )
    => {
    return [... acc , ... cur.split (/[,|]+/).filter (v => v !== '
      ')]
  }, [])) ]

  return (
    <li onMouseEnter ={() => props.hoverLang ( vocab )}
      onMouseLeave ={() => props.hoverLang ()} className ="sb -
      language "><span className ="sb - lbracket ">[</span ><div >{
      props.language.map ((s, i) => <String examineString ={
      props.examineString } key ={'s-${ props.langid }-${i}'}>{s
      }</String >)}</div ><span className ="sb - rbracket ">]</
      span ></li >
  )
}


function String ( props ) {
  return (
    <span onClick ={() => props.examineString ( props.children )}
      className ="sb - string ">{ props.children }</span >
  )
}
```

```
"""TextDisplay.js"""
import { useRef } from 'react'

export default function TextDisplay(props) {
  const elem = useRef(null)

  return (
    <div className="text">
      <pre className={props.noHighlight ? 'no-highlight' : ''
        } dangerouslySetInnerHTML={{ __html: props.text}}
        ref={elem}></pre>
      <div className="btns">
        <button id="btn-new" onClick={props.reset}>New</
          button>
        <button id="btn-export" onClick={() => props.download
          (elem)}>Export</button>
      </div>
    </div>
  )
}
```

```javascript
"""TextEntry.js"""
import { useState, useCallback } from 'react'

export default function TextEntry(props) {
  const [textareaValue, setTextareaValue] = useState('')
  const handleTextarea = useCallback(e => {
    setTextareaValue(e.target.value)
  }, [textareaValue])

  return (
    <div className="input-text">
      <textarea value={textareaValue} onChange={
        handleTextarea} placeholder="Enter text or a TimeML
        file to annotate"></textarea>
      <div className="btns">
        <button onClick={() => { props.grabParse(
          textareaValue) }}>Annotate</button> <a target="
          _blank" rel="noopener noreferrer" href="https://
          www.scss.tcd.ie/~dwoods/thesis/code/example.tml">
          Sample 1</a> <a target="_blank" rel="noopener
          noreferrer" href="https://www.scss.tcd.ie/~dwoods/
          thesis/code/example2.tml">Sample 2</a>
      </div>
    </div>
  )
}
```