Title

Example Author
Affiliation
example@email.org

Someone Else
Another Affiliation
another@email.org

1 Introduction

This paper explores using string-based models to effectively represent event data such as might be found in a document annotated with TIMEML. We describe how such data may be simply translated to strings, as well as operations on the these resultant strings which may be used to infer new information.

We believe strings pose a strong candidate for modelling this information over other approaches such as models based in predicate logic, which may inform an infinite number of possibilities, whereas strings are computational entities, and are often simpler to handle, as we are able to operate on the level of individual strings, which would be impossible using other models.

An *event-string* is a string $s \in \Sigma^+$ for an alphabet Σ, the powerset of some fixed set A of fluents *i.e.* predicates which have an associated temporality. If the string $s = \alpha_1 \cdots \alpha_n$ is taken as a sequence of n moments in time, each symbol $\alpha_i \in s$ represents the set of relevant fluents holding at the moment indexed by i. The string is read from left to right chronologically, so that any predicates which hold at the moment at index i are understood to have held before the moment indexed by j if and only if $i < j$.

The precise duration of each moment is taken as unimportant in the current discussion, and thus the strings model an inertial world, whereby *change* is the only mark of progression from one moment to the next – "there is no time without change" (Aristotle).

Thus, if $\alpha_i = \alpha_{i+1}$ for any $1 \leq i < n$, then either symbol may be safely deleted from s without affecting the interpretation of the string, as the remaining symbol is simply taken as representing a longer moment. This operation of removing repetition from the event-string is known as *block-compression*. We may also describe the inverse of this process, which introduces repeated elements in an event-string without changing its interpretation, allowing us greater flexibility in our manipulation of the string. These operations are detailed in the next section.

2 Superposition and Block Compression

With two strings s and s' of the same length n built from an alphabet Σ, the powerset of some fixed set A, the *superposition* $s \,\&\, s'$ of s and s' is their componentwise union:

$$\alpha_1 \cdots \alpha_n \,\&\, \alpha'_1 \cdots \alpha'_n \;:=\; (\alpha_1 \cup \alpha'_1) \cdots (\alpha_n \cup \alpha'_n)$$

For convenience of notation, we will use boxes rather than curly braces $\{\ \}$ to represent sets in Σ, such that each symbol α in a string s corresponds to exactly one box. For example, with $a, b, c, d \in A$:

$$\boxed{a}\boxed{c} \,\&\, \boxed{b}\boxed{d} \;=\; \boxed{a, b}\boxed{c, d} \;\in\; \Sigma^2$$

Extending this function to languages L and L' over the same alphabet is a simple matter of collecting the superpositions of strings from each language of equal length:

$$L \,\&\, L' \;:=\; \bigcup_{n \geq 0} \{ s \,\&\, s' \mid s \in L \cap \Sigma^n \text{ and } s' \in L' \cap \Sigma^n \}$$

For example, $L \,\&\, \boxed{\ }^* = L$. If L and L' are regular languages computed by finite automata with transitions $\rightarrow$ and $\rightarrow'$, then the superposition $L \;\&\; L'$ is a regular language computed by a finite automaton with transitions $\Rightarrow$ formed by running $\rightarrow$ and $\rightarrow'$ in lockstep according to the rule

$$\frac{q \xrightarrow{\alpha} r \qquad q' \overset{\alpha'}{\rightarrow}{}' \; r'}{(q, q') \overset{\alpha \cup \alpha'}{\Rightarrow} (r, r')}$$

The disadvantage of this process is that it requires the strings to be of equal length, which is an idealised situation. In order to work around this, and introduce asynchrony, we may manipulate the strings in such a way that they conform to our needs. For example, we can cause a string $s = \alpha_1 \cdots \alpha_n$ to *stutter* such that $\alpha_i = \alpha_i + 1$ for some integer $0 < i < n$. For example, $\boxed{a}\boxed{a}\boxed{a}\boxed{c}\boxed{c}$ is a stuttering version of $\boxed{a}\boxed{c}$. If a string does not stutter, it is *stutterless*, and we can transform a stuttering string to this state by using "block compression":

$$\mathrm{bc}(s) \;:=\; \begin{cases} s & \text{if } \mathit{length}(s) \leq 1 \\ \mathrm{bc}(\alpha s') & \text{if } s = \alpha\alpha s' \\ \alpha\, \mathrm{bc}(\alpha' s') & \text{if } s = \alpha\alpha' s' \text{ with } \alpha \neq \alpha' \end{cases}$$

This function can be applied multiple times to a string, but the output will not change after the first application: $\mathrm{bc}(\mathrm{bc}(s)) = \mathrm{bc}(s)$. We can also use the inverse of this function to generate infinitely many stuttering strings:

$$\begin{aligned} \mathrm{bc}^{-1}(\boxed{a}\boxed{c}) &= \boxed{a}\boxed{a}\boxed{c} \\ &= \boxed{a}\boxed{c}\boxed{c} \\ &= \boxed{a}\boxed{a}\boxed{c}\boxed{c} \\ &\;\;\vdots \end{aligned}$$

We can say that any of the strings generated by this inverse block compression are bc-*equivalent*. Precisely, a string s' is bc-equivalent to a string s iff $s' \in \mathrm{bc}^{-1}\mathrm{bc}(s)$.

We can now define the *asynchronous superposition* of strings s and s' as the (provably) *finite* set obtained by block compressing the *infinite* language generated by superposing the strings which are bc-equivalent to s and s':

$$s \,\&_{\mathrm{bc}}\, s' \;:=\; \{\mathrm{bc}(s'') \mid s'' \in \mathrm{bc}^{-1}\mathrm{bc}(s) \;\&\; \mathrm{bc}^{-1}\mathrm{bc}(s')\}$$

For example $\boxed{a}\boxed{c}\; \&_{\mathrm{bc}}\; \boxed{b}\boxed{d}$ will comprise three strings:

$$\begin{array}{r} \boxed{a,b}\boxed{c,d} \\ \boxed{a,b}\boxed{a,d}\boxed{c,d} \\ \boxed{a,b}\boxed{b,c}\boxed{c,d} \end{array}$$

In order to avoid generating all possible strings when using the inverse block compression, we introduce an upper bound to the length of the strings which will be superposed. It can be shown that with two strings of length n and n', the longest bc-unique string (one which has no shorter bc-equivalent strings) produced through asynchronous superposition will be of length $n + n' - 1$.

3 Upper Bound on Asynchronous Superposition

Proposition 1. *For all $s, s' \in \Sigma^+$ and all $s'' \in s \hat{\&} s'$,*

$$length(s'') \leq length(s) + length(s') - 1$$

For all $s, s' \in \Sigma^*$, we define a finite set $s \hat{\&} s'$ of strings over Σ with enough of the strings in $\mathrm{bc}^{-1}\mathrm{bc}(s) \,\&\, \mathrm{bc}^{-1}\mathrm{bc}(s')$ to form $s \,\&_{\mathrm{bc}}\, s'$. The definition proceeds by induction on s and s', with

$$\begin{aligned} \epsilon \hat{\&} \epsilon &:= \{\epsilon\} \\ \epsilon \hat{\&} s &:= \emptyset \quad \text{for } s \neq \epsilon \\ s \hat{\&} \epsilon &:= \emptyset \quad \text{for } s \neq \epsilon \end{aligned}$$

and for all $\alpha, \alpha' \in \Sigma$,

$$\begin{aligned} \alpha s \hat{\&} \alpha' s' &:= (\alpha \cup \alpha')(\alpha s \hat{\&} s' \mid s \hat{\&} \alpha' s' \mid s \hat{\&} s') \\ &= \{(\alpha \cup \alpha')s'' \mid s'' \in (\alpha s \hat{\&} s') \cup (s \hat{\&} \alpha' s') \cup (s \hat{\&} s')\} \end{aligned}$$

Note that even if neither s nor s' stutters, a string in $s \hat{\&} s'$ may stutter (e.g., $\boxed{a,c}\boxed{a,c} \in \boxed{a}\boxed{c} \hat{\&} \boxed{c}\boxed{a}$) but can be made stutterless through bc.

Proposition 2. *For all $s, s' \in \Sigma^+$,*

$$s \hat{\&} s' \subset \mathrm{bc}^{-1}\mathrm{bc}(s) \,\&\, \mathrm{bc}^{-1}\mathrm{bc}(s')$$

and

$$\{\mathrm{bc}(s'') \mid s'' \in s \hat{\&} s'\} = s \,\&_{\mathrm{bc}}\, s'$$

Now, for any integer $k > 0$ and string $s = \alpha_1 \cdots \alpha_n$ over Σ, we introduce a new function pad_k which will generate the set of strings with length k which are bc-equivalent to s:

$$\begin{aligned} pad_k(\alpha_1 \cdots \alpha_n) &:= \alpha_1^+ \cdots \alpha_n^+ \cap \Sigma^k \\ &= \{\alpha_1^{k_1} \cdots \alpha_n^{k_n} \mid k_1, \ldots, k_n \geq 1 \text{ and } \sum_{i=1}^{n} k_i = k\} \\ &\subset \mathrm{bc}^{-1}\mathrm{bc}(\alpha_1 \cdots \alpha_n) \end{aligned}$$

For example, $pad_4(\boxed{a}\boxed{c})$ will generate $\{\ \boxed{a}\boxed{a}\boxed{a}\boxed{c},\ \boxed{a}\boxed{a}\boxed{c}\boxed{c},\ \boxed{a}\boxed{c}\boxed{c}\boxed{c}\ \}$. We can use this new function in our calculation of asynchronous superposition, to limit the generation of strings from the inverse block compression step. Since we know from Proposition 1 that the maximum possible length we might need is $n + n' - 1$, we can use this value in the pad function to just generate the strings of that length, giving us a new definition of asynchronous superposition:

Corollary 3. *For any $s, s' \in \Sigma^+$ with nonzero lengths n and n' respectively,*

$$s \,\&_{\mathrm{bc}}\, s' = \{\mathrm{bc}(s'') \mid s'' \in pad_{n+n'-1}(s) \,\&\, pad_{n+n'-1}(s')\}$$

Neither $s \hat{\&} s'$ nor $pad_{n+n'-1}(s) \,\&\, pad_{n+n'-1}(s')$ need be a subset of the other, even though, under the assumptions of Corollary 3, both sets block compress to $s \,\&_{\mathrm{bc}}\, s'$.

4 Event Representation

Now we may use asynchronous superposition to generate the 13 strings in $\boxed{\ }\boxed{e}\boxed{\ } \;\&_{\mathrm{bc}}\; \boxed{\ }\boxed{e'}\boxed{\ }$, each of which corresponds to one of the unique interval relations in Allen (1983). We use the empty box $\boxed{\ }$ as a string of length 1 (not to confused with the empty string ϵ, which is length 0) to bound events, allowing us to represent the fact that they are finite – they have a beginning and ending point. It is prudent to assume that we will deal only with finite event data, such that there are no fluents which do not have both an associated start-point and end-point. If such a fluent were to occur, it would be trivial to add it to every position in the string.

The bounding boxes represent the time before and after the event occurs, during which no other fluents belonging to the alphabet Σ hold. The Allen Relation Strings are laid out below:

$e = e'$	$\boxed{\ }\boxed{e, e'}\boxed{\ }$	equal
e **s** e'	$\boxed{\ }\boxed{e, e'}\boxed{e'}\boxed{\ }$	starts
e **si** e'	$\boxed{\ }\boxed{e, e'}\boxed{e}\boxed{\ }$	starts (inverse)
e **f** e'	$\boxed{\ }\boxed{e'}\boxed{e, e'}\boxed{\ }$	finishes
e **fi** e'	$\boxed{\ }\boxed{e}\boxed{e, e'}\boxed{\ }$	finishes (inverse)
e **d** e'	$\boxed{\ }\boxed{e'}\boxed{e, e'}\boxed{e'}\boxed{\ }$	during
e **di** e'	$\boxed{\ }\boxed{e}\boxed{e, e'}\boxed{e}\boxed{\ }$	during (inverse)
e **o** e'	$\boxed{\ }\boxed{e}\boxed{e, e'}\boxed{e'}\boxed{\ }$	overlaps
e **oi** e'	$\boxed{\ }\boxed{e'}\boxed{e, e'}\boxed{e}\boxed{\ }$	overlaps (inverse)
e **m** e'	$\boxed{\ }\boxed{e}\boxed{e'}\boxed{\ }$	meets
e **mi** e'	$\boxed{\ }\boxed{e'}\boxed{e}\boxed{\ }$	meets (inverse)
$e < e'$	$\boxed{\ }\boxed{e}\boxed{\ }\boxed{e'}\boxed{\ }$	before
$e > e'$	$\boxed{\ }\boxed{e'}\boxed{\ }\boxed{e}\boxed{\ }$	after

These Allen Relations are included in the attributes of TIMEML, the standard markup language used for event annotation in texts, as TLINKs. By extracting the TLINKs from an annotated document, and translating them to our event-string representation, we may begin to reason about the relationships between annotated events which do not have an associated TLINK in the markup. For example, the document may give us a relation between events e and e', and another relation between e' and e'', and from this we may infer the possible relations between e and e''.

As asynchronous superposition is a commutative, associative, homomorphic binary relation over event-strings, we may superpose arbitrary numbers of event-strings. Note, however, that superposing even a small number of unconstrained, bounded events leads to a massive blowup in the number of possible outcomes. Currently, attempting to compute the result of more than five is beyond the capabilities of available hardware:

$$2 \text{ bounded events} \rightarrow 13 \text{ outcomes}$$
$$3 \text{ bounded events} \rightarrow 409 \text{ outcomes}$$
$$4 \text{ bounded events} \rightarrow 23917 \text{ outcomes}$$
$$5 \text{ bounded events} \rightarrow 2244361 \text{ outcomes}$$

Clearly, simply superposing bounded events in this manner is infeasible, as it is unreasonable to expect that any given document would contain five or less events. In order to avoid generating such a large

number of computed event-strings, it is necessary to add constraints where appropriate as to what strings may be considered allowable for a particular context.

5 Constraints on Event-Strings

Two approaches to constraints may be implemented, which are not mutually exclusive. The first is to prevent unwanted strings from being generated, based on the nature of the operand strings, and the second is to remove disallowed strings from the set of outputs. The former approach is preferred from a computational standpoint, as there is less data to store and process.

We assume that every fluent we encounter has exactly one beginning and one ending – that is, that events do not *resume* once they have ended. Events of the same type may stop and start frequently, but by assuming that every instance of an event will have a uniquely identifying fluent, we can discard any strings which feature such a resumption.

Additionally, fluents may be referred to multiple times by different TLINKs in an annotated document, and we assume that they will be *consistent* within the context of that document *i.e.* if a relation holds between e and e', and a relation holds between e' and e'', then both instances of e' refer to the same fluent, and the two relations will not contradict each other.

These last two points are interesting in particular, as they lead to a specific kind of superposition between strings $s, s' \in \Sigma^+$ when some symbol $\alpha \in s$ is equal to some other symbol $\alpha' \in s'$. In this scenario, the symbols must unify when superposing the strings, in order to create a well-formed event-string in accordance with the above two constraints. The algorithm in this case involves using the pad_k operation on each string as usual, but with the maximum length reduced to $n + n' - 3$ (where n and n' are the lengths of s and s', respectively). However, at this point, superposition is only carried out on those results of padding where the indices of any shared fluents match exactly.

Finally, we may also introduce simple binary constraints if external information is available (*e.g.* "the string may not begin with e"), and these might be simply intersected with the result of a superposition: $(s \mathbin{\&_{\mathrm{bc}}} s') \cap C$, where C represents the constraints to be applied.

6 title

Something else goes here.

7 Conclusion

In this paper we have done a thing.

References

Allen, J. F. (1983). Maintaining knowledge about temporal intervals. *Communications of the ACM 26*(11), 832–843.