Preliminary 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. It is described 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.

It is put forward that strings pose a strong candidate as a basis for modelling this information over other approaches such as models rooted in predicate logic. Due to a string's nature as a computational entity, it is both simple and intuitive to understand as a human, and efficiently tractable for a machine. Strings also permit operations on an individual level, which may be impossible for models which generate all formulae at once.

An *event-string* is a string $s \in \Sigma^+$ for an alphabet Σ, the powerset of some fixed set $\mathcal{F}$ 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* (Fernando, 2016). 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 $\mathcal{F}$, 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

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

A disadvantage of this operation is that it requires the string operands to be of equal length, which is an overly specific case. In order to generalise this procedure to strings of arbitrary lengths, we may manipulate the strings to introduce asynchrony. 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":

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

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:

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

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 $s'' \in s \mathbin{\hat{\&}} s'$,*

$$\text{length}(s'') \ \leq\ \text{length}(s) + \text{length}(s') - 1$$

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

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

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

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

Note that a string in $s \mathbin{\hat{\&}} s'$ might stutter, even if neither of the operands s or s' do (*e.g.* $\boxed{a,c}\boxed{a,c} \in \boxed{a}\boxed{c} \mathbin{\&} \boxed{c}\boxed{a}$) – however, it can be made stutterless through block compression.

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

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

and

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

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,*

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

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

4 Event Representation

Now we may use asynchronous superposition to generate the 13 strings in $\boxed{}\boxed{e}\boxed{} \,\&_{\rm bc}\, \boxed{}\boxed{e'}\boxed{}$, 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
$e\ \mathbf{s}\ e'$	$\boxed{}\boxed{e,e'}\boxed{e'}\boxed{}$	starts
$e\ \mathbf{si}\ e'$	$\boxed{}\boxed{e,e'}\boxed{e}\boxed{}$	starts (inverse)
$e\ \mathbf{f}\ e'$	$\boxed{}\boxed{e'}\boxed{e,e'}\boxed{}$	finishes
$e\ \mathbf{fi}\ e'$	$\boxed{}\boxed{e}\boxed{e,e'}\boxed{}$	finishes (inverse)
$e\ \mathbf{d}\ e'$	$\boxed{}\boxed{e'}\boxed{e,e'}\boxed{e'}\boxed{}$	during
$e\ \mathbf{di}\ e'$	$\boxed{}\boxed{e}\boxed{e,e'}\boxed{e}\boxed{}$	during (inverse)
$e\ \mathbf{o}\ e'$	$\boxed{}\boxed{e}\boxed{e,e'}\boxed{e'}\boxed{}$	overlaps
$e\ \mathbf{oi}\ e'$	$\boxed{}\boxed{e'}\boxed{e,e'}\boxed{e}\boxed{}$	overlaps (inverse)
$e\ \mathbf{m}\ e'$	$\boxed{}\boxed{e}\boxed{e'}\boxed{}$	meets
$e\ \mathbf{mi}\ e'$	$\boxed{}\boxed{e'}\boxed{e}\boxed{}$	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 ISO-TimeML (Pustejovsky et al., 2010), 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: $s \,\&_{\rm bc}\, s' \,\&_{\rm bc}\, s'' \,\&_{\rm bc}\, \cdots$. 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:

$$\begin{aligned}
&2 \text{ bounded events} \to 13 \text{ outcomes}\\
&3 \text{ bounded events} \to 409 \text{ outcomes}\\
&4 \text{ bounded events} \to 23917 \text{ outcomes}\\
&5 \text{ bounded events} \to 2244361 \text{ outcomes}
\end{aligned}$$

Clearly, simply superposing bounded events in this manner is not feasible, 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. To achieve this, when a symbol α in s is also present in s', and the asynchronous superposition of these strings is desired, padding is carried out as normal, but superposition is only permitted of those results of padding in which the indices of the matching symbols are equal:

$$\alpha_1 \cdots \alpha_n \,\&\, \alpha'_1 \cdots \alpha'_n \;:=\; \{(\alpha_1 \cup \alpha'_1) \cdots (\alpha_n \cup \alpha'_n) \mid \alpha_i = \alpha'_j \Rightarrow i = j\}$$

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: $(s \,\&_{\mathrm{bc}}\, s') \cap C$, where C represents the constraints to be applied.

6 Grounding

The TIMEBANK Corpus (Pustejovsky et al., 2003) provides a large number of documents annotated with TimeML, from which we may extract the event data – in particular the TLINKs. As mentioned in Section 4, attempting to generate the possible worlds from just the bounded event instances alone is not possible, as there are just too many. Instead, we begin by looking at just those events which are linked to another by Allen Relation.

We translate these relations immediately to their corresponding event-strings, and superpose these according to the constraints mentioned in Section 5. In this way, we may generate a much smaller set of possible outcomes from a larger number of bounded events.

A drawback here is that for this to be effective, it relies on the events being heavily constrained by their interrelations. If there are too few TLINKs relative to the number of events, we still run into the problem of combinatorial explosion. An additional issue in computation of the superposition of events arises from unordered data leading to a much less efficient calculation of final results. For example, $\boxed{}\boxed{a}\boxed{}\boxed{b}\boxed{} \,\&_{\mathrm{bc}}\, \boxed{}\boxed{b}\boxed{}\boxed{c}\boxed{} \,\&_{\mathrm{bc}}\, \boxed{}\boxed{c}\boxed{}\boxed{d}\boxed{}$ and $\boxed{}\boxed{a}\boxed{}\boxed{b}\boxed{} \,\&_{\mathrm{bc}}\, \boxed{}\boxed{c}\boxed{}\boxed{d}\boxed{} \,\&_{\mathrm{bc}}\, \boxed{}\boxed{b}\boxed{}\boxed{c}\boxed{}$ should produce the same, single output: $\boxed{}\boxed{a}\boxed{}\boxed{b}\boxed{}\boxed{c}\boxed{}\boxed{d}\boxed{}$, which they do. However, due to the respective orderings, the first sequence will arrive at that conclusion much faster as $\boxed{}\boxed{a}\boxed{}\boxed{b}\boxed{} \,\&_{\mathrm{bc}}\, \boxed{}\boxed{b}\boxed{}\boxed{c}\boxed{}$ has one outcome ($\boxed{}\boxed{a}\boxed{}\boxed{b}\boxed{}\boxed{c}\boxed{}$), which can immediately be asynchronously superposed with $\boxed{}\boxed{c}\boxed{}\boxed{d}\boxed{}$ to produce the final output. However, $\boxed{}\boxed{a}\boxed{}\boxed{b}\boxed{}$ $\&_{\mathrm{bc}}$ $\boxed{}\boxed{c}\boxed{}\boxed{d}\boxed{}$ has 321 output strings, each of which must be individually asynchronously superposed with $\boxed{}\boxed{b}\boxed{}\boxed{c}\boxed{}$, only to come to the same conclusion, as only one of these results is consistent.

One potential way to work around this pitfall is a grouping and ordering stage, where initially only events linked by some relation may be superposed, and only after the operand strings have been sorted to some optimal order. It may be prudent to only perform superposition at all on event strings which may be linked through one or more relations or shared fluents. In this way, new, underspecified events may be formed from the output strings. Consider the scenario with $e_1, \cdots, e_8 \in \mathcal{F}$, and the following Allen Relations:

$$\begin{aligned} e_1 &< e_4 \\ e_1 &\ \mathbf{m}\ e_2 \\ e_2 &\ \mathbf{di}\ e_3 \\ e_5 &\ \mathbf{s}\ e_7 \\ e_8 &> e_5 \end{aligned}$$

Note that three obvious clusters of events can be seen: e_1, e_2, e_3, and e_4 are interlinked, as are e_5, e_7, and e_8, separately, while e_6 is not linked to either group. We might form the underspecified event groups E_1 and E_2 to refer to these first two clusters, at which point we may freely treat these groups as normal bounded events, and perform asynchronous superposition on their event strings, as well as with that of e_6 – thus reducing the number of inputs from 8 to 3.

Additionally, various weightings might be considered as a method of priority-ordering in the case of a large $\mathcal{F}$, such as the number of component events in an underspecified event groups, or the number of relations linking to a particular event.

7 Conclusion

We have explored in this work the possibility of using strings as basis for modelling event data, motivated by their nature as computational entities. The operation of asynchronous superposition was described for composing strings which represent finite, bounded events, as well as its limits in terms of blowup when the operation is repeated in sequence. The problem is addressed by constraining the strings which may be superposed, with the thirteen unique Allen Relations forming the main part of these, as these can be found in annotated corpora such as TIMEBANK, using the TimeML standard.

Future work on this topic will examine using alternative model-bases to approach the same issue, such as using finite state automata, or a hybrid string/FSA approach, as well as the potential of employing parallel processing methods in order to tackle the combinatorial explosion that occurs in asynchronously superposing unconstrained bounded events.

References

Allen, J. F. (1983). Maintaining Knowledge About Temporal Intervals. *Communications of the ACM 26*(11), 832–843.

Fernando, T. (2016). Prior and temporal sequences for natural language. *Synthese 193*(11), 3625–3637.

Pustejovsky, J., P. Hanks, R. Sauri, A. See, R. Gaizauskas, A. Setzer, D. Radev, B. Sundheim, D. Day, L. Ferro, et al. (2003). The TIMEBANK Corpus. In *Corpus Linguistics*, Volume 2003, pp. 647–656. Lancaster, UK.

Pustejovsky, J., K. Lee, H. Bunt, and L. Romary (2010). ISO-TimeML: An International Standard for Semantic Annotation. In *LREC*, Volume 10, pp. 394–397.

A Map-Reduce

The Map-Reduce model is designed to process very large data sets efficiently, taking advantage of parallel processing methods and distributed computing. It uses a repeated multi-step algorithm which divides the data into chunks, converts these chunks into a number of key/value pairs using a *Mapper* function, then distributes these pairs over a number of *Reducers*. The Reducer will be called once for each unique key created by the Mapper, and will iterate through the values associated with it to produce zero or more outputs.

For example, a word-counting program may break a document into lines, each of which will be the input to a Mapper. The Mapper will break the line into words and output a key/value pair for each unique word, with the word itself as the key, and the number of its occurrences in the line as the value. The Reducer takes all the key/value pairs from all lines which have the same key and sums the values to output a total count of that word for the document.

In our asynchronous superposition of unconstrained bounded events, as we saw, a large amount of data is created in the form of the event-string outputs. This seems like a potential target for applying the Map-Reduce model.

In theory, our Mapper might take as input the event-strings to be superposed and would return as key pairs of the strings, and as value the set of strings resulting from the superposition of the key-pair:

$$\mathit{map}(s_1 \,\&_{\mathrm{bc}}\, s_2 \,\&_{\mathrm{bc}}\, \cdots \,\&_{\mathrm{bc}}\, s_{n-1} \,\&_{\mathrm{bc}}\, s_n) = \{(s_1 s_2, s_1 \,\&_{\mathrm{bc}}\, s_2), \cdots, (s_{n-1}s_n, s_{n-1} \,\&_{\mathrm{bc}}\, s_n)\}$$

However, in this case, the Reducer has nothing to do, as each key is unique, and therefore pairs with matching keys cannot be collected and reduced to a single value. Similarly, if the key was chosen as just one of the string pair, rather than both.

Perhaps instead we take a particular string (*e.g* the first) as a key, and then superposition of it with each other string as the values:

$$\mathit{map}(s_1 \,\&_{\mathrm{bc}}\, s_2 \,\&_{\mathrm{bc}}\, \cdots \,\&_{\mathrm{bc}}\, s_{n-1} \,\&_{\mathrm{bc}}\, s_n) = \{(s_1, s_1 \,\&_{\mathrm{bc}}\, s_2), \cdots, (s_1, s_1 \,\&_{\mathrm{bc}}\, s_n)\}$$

We can use the Reducer now, but there's not a whole lot that the function can do, except collect the values. If it were to do that, it must be done in such a way that the result of each superposition from the previous step were kept separate, so we would have a collection of n separate sets of generated strings, in order to prevent superposing two strings from the same earlier generation (which should be considered as parallel worlds). While this could potentially fulfill the goals of Map-Reduce, a large amount of redundant superposition has been performed, each time increasing the maximum length of the generated strings, making the computation less efficient at each stage.

As a third alternative, suppose the Mapper just used a generic key each time, and returned the result of the padding function.

$$\mathit{map}(s_1 \,\&_{\mathrm{bc}}\, s_2 \,\&_{\mathrm{bc}}\, \cdots \,\&_{\mathrm{bc}}\, s_{n-1} \,\&_{\mathrm{bc}}\, s_n) = \{(s_{key}, \mathit{pad}_k(s_1)), \cdots, (s_{key}, \mathit{pad}_k(s_n))\}$$

where s_{key} is a generic key, and $k = \text{length}(s_1) + \cdots + \text{length}(s_n) - (n-1)$. In this scenario, the Reducer will be able to perform a simple, synchonous superposition on all the pairings of the values – however, at just $n = 5$, k becomes 11, and for a simple bounded event $\boxed{}\boxed{e}\boxed{}$ there are 66 different strings returned by pad_{11}, meaning there may be 66^5 different superposition operations to perform, which is over a billion.

Due to the fact that the Mapper and Reducer should both be pure, stateless functions, we are unfortunately quite limited in what applications it has in the context of asynchronous superposition. The best case would be to use a generic key for the Mapper, as in the last example above, but to have the values be superposed pairs of strings as in the first example. Then the Reducer could combine the generated strings in order to create new data and repeat the process. However, using a generic key in the manner described somewhat defeats the purpose of using the Mapper at all, and thus is not a good implementation of the Map-Reduce model either.