Timeline granularity and probabilities of Allen's interval relations

Pavel Petrukhin, MCS

A Dissertation

Presented to the University of Dublin, Trinity College

in partial fulfilment of the requirements for the degree of

Master in Computer Science

Supervisor: Dr. Tim Fernando

April 2024

Timeline granularity and probabilities of Allen's interval relations

Pavel Petrukhin, Master in Computer Science

University of Dublin, Trinity College, 2024

Supervisor: Dr. Tim Fernando

Reasoning about time in a qualitative manner has always been an attractive paradigm in Computer Science. Work of Allen (1983) introduced 13 qualitative relations for intervals. These relations have had a tremendous impact across several areas of Computer Science. This work belongs to the area of timeline probability. We extend the notion of superposition originally defined for finite temporality strings to the domain of finite state machines. The key contribution of this work is the introduction of a framework for modelling timeline probabilities of complex events using the superposition of finite state automata. We apply this framework to the probabilities of Allen's relations and show that it helps correct some of the unintuitive results that were obtained in the literature.

Acknowledgments

I would like to thank my family and friends for supporting me during the college journey. I would also like to express gratitude to my supervisor, Dr. Tim Fernando, for guiding me through the dissertation process.

Pavel Petrukhin

University of Dublin, Trinity College
April 2024

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This section will first discuss the concept of time and the ways in which temporal relationships can be represented. It will then explain how the time representation paradigms impact our work, its motivation and aim. Finally, it will outline the structure of the dissertation and briefly mention the content and goals of each of the subsequent chapters.

1.1 Concept of time

Representations of time date all the way back to the ancient days. Humanity used various time keeping devices throughout its history ranging from hourglasses and sundials to modern electronic clocks. In principle, there are 2 ways to think about time: quantitative and qualitative.

The quantitative representation is a more intuitive paradigm that each of us uses daily. The foundational assumption of this approach is that there exists an independent timeline which may be continuous or discrete. Each event is assigned a starting point and a duration that precisely define the start and end of the interval corresponding to the event relatively to the assumed timeline.

The distinctive feature of the qualitative paradigm is that it does not require an explicit timeline and hence the event's start time and its duration. The qualitative approach aims to represent time by explicitly reasoning about the relationships which hold between the events. For example, if event A is before event B and event B is before event C we can conclude that event A is before event C via transitivity. Note that we did not make any assumption about the event's start times or duration in our reasoning.

It is clear that quantitative representation also allows for temporal reasoning. The relationships between events are, however, specified implicitly through the values assigned to their starting points and duration. The idea of temporal reasoning is less important in the quantitative paradigm, since we can directly compare the timeline intervals without the need for inference.

1.2 Motivation

Temporal reasoning has various applications in areas such as finance, actuarial science, healthcare and software verification. For example, Kong et al. (2010) introduces a notion of multi-temporal patterns based on four predicates such as before, during, equal and overlap. They apply the multi-temporal pattern mining algorithm to a database of mainland China and Hong Kong stock markets in order to identify associative movements between the two. Moreover, Haimowitz et al. (1996) discusses a temporal reasoning framework for healthcare enterprises, while Gooneratne et al. (2008) introduces a global communication model which helps detect deadlocks and synchronisation conflicts in composite web applications. Finally, Manna and Pnueli (2012) talks about temporal verification of reactive systems, which include concurrent programs and operating systems.

Reasoning about time in a qualitative manner is an attractive paradigm because it allows us to consider timelines consisting of several events solely from the initial set of relationships that hold between them. The initial relationships could come from a temporal database or be defined as assumptions. However, only identifying the possible timelines may not be sufficient for some domains. For instance, a financial risk manager may also want to know how probable the timelines are as opposed to only knowing that they are possible based on the initial temporal data.

1.3 Project goals

The goal of our project is to build a timeline probability framework based on the qualitative time representation paradigm. In doing so, it is natural to start with exploring the probabilities of relations between simple intervals. However, we also aim to enable the representation of complex events which may consist of several logical intervals. For example, an event which represents a lifetime of a person could consist of multiple intervals such as young age, adulthood and old age. The number and precision of such intervals could be referred to as event's granularity. Therefore, through modelling complex events we also aim to explore the concept of timeline granularity.

1.4 Structure & Contents

Chapter 2 provides a detailed over of the existing literature. It starts by introducing the reader to the basic concepts of the field such as Allen's interval relations. After that, several approaches for representing timelines and computing probabilities of Allen's relations are presented. This chapter concludes with a short discussion of limitations and gaps in the existing literature which serves as a starting point to the next chapter.

Chapter 3 begins with a detailed discussion of the limitations identified in the previous chapter. Based on them, it outlines 3 research directions. After that the notion of automata superposition is introduced and an initial analysis of probabilities of Allen's interval relations is presented. The chapter concludes by discussing the design of the proposed software system, including the key algorithms that will need to be implemented.

Chapter 4 discusses the initial implementation of our system and outlines its several limitations which led us to creating the second iteration of the system in the form of a web app. The design of the web app is then thoroughly presented including the implementation of key algorithms and explanation of the main features.

Chapter 5 begins with various sanity checks in order to verify that the software system behaves correctly. The sanity checks are based on our own analytical results as well as the existing literature. After that, we explore two approaches for eliminating unintuitive probability results that were obtained in the literature. One of the approaches includes modelling timelines of complex events.

Chapter 6 summarizes our work and outlines how we contributed to the original research directions defined in Chapter 3. The chapter concludes with listing several areas of future work.

Chapter 2

State of the Art

This chapter introduces some of the foundational concepts in the area of temporal representation that constitute a required background to understand the rest of our work.

We start by talking about the history of temporal logic systems and discussing the intuition behind prior probabilities of interval relations. After that, a detailed analysis of existing work in the area of timeline probabilities is presented. As a result of this analysis, we formulate several areas of improvement for the existing approaches which serve as a starting point of our research problem.

2.1 Background

Temporal logic was first introduced by Prior (1957). Several other temporal logic systems were developed after that, including the linear temporal logic introduced in Pnueli (1977) with the purpose of formal verification of computer programs. However, the most influential and widely adopted temporal system used in computer science was introduced in Allen (1984). Allen's interval calculus had a significant influence on multiple fields in computer science including temporal reasoning Rost et al. (2022), Zhou et al. (2020), Georgala et al. (2016), Van Beek and Manchak (1996), time pattern mining Lai et al. (2024), Patel et al. (2008), timeline probability Suliman (2021), Fernando and Vogel (2019) and spatial reasoning Morales and Sciavicco (2006), Guesgen (1989).

Allen introduced 13 relations which define various temporal configurations that may hold between two intervals. Conceptually there is only 7 main relationships: "before", "meets", "starts", "overlaps", "equals", "finishes" and "during". The remaining 6 relationships are defined as inverses of the original ones, namely "before inverse" which is equivalent

to "after", "meets inverse", "starts inverse", "overlaps inverse", "finishes inverse" and "during inverse". Please note that "equals inverse" is the same as "equals" and, hence, it is omitted. The graphical representations of the relations are given in Figure 2.1.

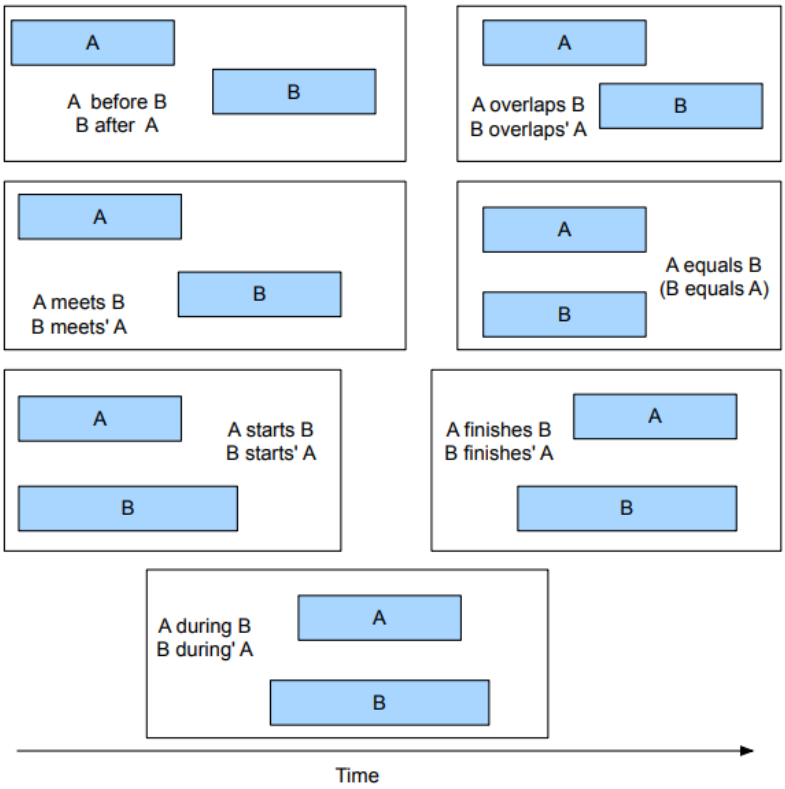


Figure 2.1: Allen's interval relations, Figure 19.11 from Jurafsky and Martin (2024)

Allen relations are qualitative which means they are not concerned with intervals' start times and duration. Allen's calculus enables us to reason about the possible relations that may hold between two intervals A and B based on the relations that hold between each of these intervals and an external set of intervals. For example, Allen (1983) provides a transitivity table for the relations which is the simplest form of temporal reasoning. The table is given in Figure 2.2 In the table, Ar_1B indicates a relation r_1 that holds between intervals A and B, while Br_2C indicates a relation r_2 that holds between intervals B and C. The cells in the table represent which relations r_3 may hold between A and C. For example if Ar_1B is "during" and Br_2C is "before" then we can conclude that Ar_3C can only be "before".

Some of the cells in the table contain multiple possibilities for r_3 and certain values of r_3 seem to appear more often than others. This naturally raises the question of whether certain relations are more probable than the rest. One might argue that due to principle

of indifference all relations must have the same prior probability of $\frac{1}{13}$. However, this proposition does not seem reasonable given the observed structure of the transitivity table.

B r2 C / A r1 B	<	>	d	di	o	oi	m	mi	s	si	f	fi
"before" <	<	no info	< o m d s	<	<	< o m d s	<	< o m d s	<	<	< o m d s	<
"after" >	no info	>	> oi mi d f	>	> oi mi d f	>	> oi mi d f	>	> oi mi d f	>	>	>
"during" d	<	>	d	no info	< o m d s	> oi mi d f	<	>	d	> oi mi d f	d	< o m d s
"contains" di	< o m di fi	> oi di mi si	o oi dur con =	di	o di fi	oi di si	o di fi	oi di si	di fi o	di	di si oi	di
"overlaps" o	<	> oi di mi si	o d s	< o m di fi	< o m	o oi dur con =	<	oi di si	o	di fi o	d s o	< o m
"over-lapped-by" oi	< o m di fi	>	oi d f	> oi mi di si	o oi dur con =	> oi mi	o di fi	>	oi d f	oi > mi	oi	oi di si
"meets" m	<	> oi mi di si	o d s	<	<	o d s	<	f fi =	m	m	d s o	<
"met-by" mi	< o m di fi	>	oi d f	>	oi d f	>	s si =	>	d f oi	>	mi	mi
"starts" s	<	>	d	< o m di fi	< o m	oi d f	<	mi	s	s si =	d	< m o
"started by" si	< o m di fi	>	oi d f	di	o di fi	oi	o di fi	mi	s si =	si	oi	di
"finishes" f	<	>	d	> oi mi di si	o d s	> oi mi	m	>	d	> oi mi	f	f fi =
"finished-by" fi	<	> oi mi di si	o d s	di	o	oi di si	m	si oi di	o	di	f fi =	fi

Figure 2.2: Transitivity table for Allen interval relations. Figure 4 from Allen (1983)

Studying probabilities of Allen's relations is a vital first step in exploring the probabilities of timelines, because the relationship between two simple intervals can be thought of as a trivial timeline. Since our work belongs to the area of timeline probability, we are going to thoroughly discuss the existing research in that field in the next section, starting with prior probabilities of the interval relations.

2.2 Closely-Related Work

In this section, we are going to discuss the existing work in the areas of prior probability of Allen's interval relations, timeline representation and timeline probabilities.

We are also going to outline some of the gaps and limitations of the existing techniques, which will give us a concrete set of research questions to explore in the further sections.

2.2.1 Prior probabilities of Allen's interval relations

The work of Fernando and Vogel (2019) observes the same structure in the transitivity table of Allen as we saw in the previous section and hence poses the following question:

Given an Allen relation R, what is the probability that R relates intervals a and a', aRa'?

The emphasis in this question is on the fact that we know nothing about intervals a and a'. Therefore, the probabilities that are derived are referred to as 'prior'. Answering the above question requires a way to model the Allen's relations. The paper considers two modelling approaches: n ordered points and n interval names.

Probabilities over n ordered points

Let's denote $\mathcal{AR} = \{b, bi, d, di, o, oi, m, mi, s, si, f, fi, e\}$ as the set of Allen's relations. The authors start by defining a linear order of n points as $[n] := \{i \in \mathbb{Z} \mid 1 \leq i \leq n\}$. After that, they define intervals as $(l, r] := \{i \in [n] \mid l < i \leq r\}$. This allows them to express the probability of aRa' as the probability that $(l, r]\, R\, (l', r']$. For example, $(3, 5]\, d\, (1, 10]$ means that interval $(3, 5]$ happens during the interval $(1, 10]$.

The paper goes on to classify each of the 13 Allen's relations as short, medium and long. This classification is based on the string of sets representation of Allen's relations suggested in Durand and Schwer (2008). In this representation, each box denotes a set of terms which are happening at a given time point. The time points are defined by the order of the boxes from left to right.

Table 2.1 presents the string of sets representations for all of Allen's relations. The length of relations is determined by the length of their representations. In turn, the length of the strings of sets depends on how many distinct time points (denoted as boxes) are required to represent the corresponding relations. For example, "before" or $(l, r]\, b\, (l', r']$ requires 4 distinct points and hence the string representing it consists of 4 boxes. Similarly, "meets"

or $(l, r]\, m\, (l', r']$ only needs 3 points, because the end and start of the intervals are the same and therefore are a part of the same box (set). Hence, "meets" is represented with 3 boxes. From the table, we can see that b, bi, d, di, o and oi have length of 4, m, mi, s, si, f and fi have length of 3 and e has length of 2. Relations of length 4 are called "long" while relations of length 3 and 2 are called "medium" and "short" respectively.

Relation $(l, r]\, R\, (l', r']$	**String for** R	**Inverse relation** R^{-1}	**String for** R^{-1}
$(l, r]\, b\, (l', r']$	$l \mid r \mid l' \mid r'$	bi	$l' \mid r' \mid l \mid r$
$(l, r]\, d\, (l', r']$	$l' \mid l \mid r \mid r'$	di	$l \mid l' \mid r' \mid r$
$(l, r]\, o\, (l', r']$	$l \mid l' \mid r \mid r'$	oi	$l' \mid l \mid r' \mid r$
$(l, r]\, m\, (l', r']$	$l \mid r, l' \mid r'$	mi	$l' \mid r', l \mid r$
$(l, r]\, s\, (l', r']$	$l', l \mid r \mid r'$	si	$l, l' \mid r' \mid r$
$(l, r]\, f\, (l', r']$	$l \mid l' \mid r, r'$	fi	$l' \mid l \mid r', r$
$(l, r]\, e\, (l', r']$	$l, l' \mid r, r'$	e	$l, l' \mid r, r'$

Table 2.1: Strings of sets representing Allen's relations based on Figure 4 from Durand and Schwer (2008) and Table 1 from Fernando and Vogel (2019). Symbols appearing in the same box (set) are assumed to be equal in terms of time. The boxes in the strings are arranged from left to right, meaning that symbols on the left are strictly less than symbols on the right time wise.

This classification enables the authors to derive the prior probabilities of relations based on their lengths. For a given relation, the probability is computed as the number of ways to select two intervals from a linear order $[n]$ such that they satisfy the relation divided by the total number of ways to select two intervals . The total number of ways to select two intervals from a linear order $[n]$ is $\Omega_n = \binom{n}{2}\binom{n}{2}$. For short relations i.e. only "equal", we need to select a two-element subset from linear order $[n]$, this yields $\binom{n}{2}$ possibilities. Similarly, for medium relations which are defined using three unique points, the number of subsets is $\binom{n}{3}$, while there are $\binom{n}{4}$ subsets corresponding to the long relations defined by four distinct points. The formulas for prior probabilities of short, medium and long relations are given in Table 2.2

Probabilities over n interval names

Another model considered in Fernando and Vogel (2019) is based on representing the relations using named intervals. Firstly, the authors redefine $n \geq 2$ as the number of intervals under consideration as opposed to the number of points. Hence, each element i in linear order $[n]$ now corresponds to an interval name. The authors define a labelling

Relation length	**Probability over a finite order** $[n]$
short	$\frac{2}{n(n-1)}$
medium	$\frac{2(n-2)}{3n(n-1)}$
long	$\frac{(n-3)(n-2)}{6n(n-1)}$

Table 2.2: Summary of Allen's relation probabilities over finite order $[n]$ obtained in Fernando and Vogel (2019).

function $\omega : ([n] \times [n]) \rightarrow \mathcal{AR}$, which maps each pair of intervals to a relation. It is important to note that without further restriction this function may behave inconsistently. For example, assume we have 3 intervals denoted as A, B and C. The function can assign (A, B) to relation b and (B, C) to relation b. In other words, A is before B and B is before C. Now, if (A, C) is assigned to bi, meaning that C happens before A, we get a contradiction, because due to the transitivity of relation before C is actually supposed to happen after A. In order to tackle this problem, the authors introduce the following restrictions on ω:

$$\exists\, \alpha : [n] \rightarrow [2n], \beta : [n] \rightarrow [2n] \text{ such that,}$$
$$\forall i \in [n]\, \alpha(i) < \beta(i),$$
$$\forall (i, j) \in [n] \times [n]\, (\alpha(i), \beta(i)]\, R\, (\alpha(j), \beta(j)] \wedge \omega(i, j) = R$$

In simple terms, these restrictions mean that we need to find functions α and β that can assign each of the intervals to a start and end point respectively such that the resulting relations do not have contradictions. The complexity of the problem comes from the fact that for some interval configurations there may be multiple possible pairs of functions α and β which satisfy the above. For example, if all intervals are the same it is sufficient that $\alpha(i) = a$ and $\beta(i) = b$ where (a, b) is a pair from linear order $[2n]$ and $a < b$. Hence, there exists $\binom{2n}{2}$ of such function pairs. The authors suggest that the key to solving the problem lies in understanding the sample space that is generated by $\omega(i, j)$. In other words, we want to find a way to compute the consistent labellings given intervals from $[n]$.

They approach this by looking at strings of sets representing possible configurations of intervals coming from $[n]$. The authors show that the function that maps strings of sets for $[n]$ to the set of consistent labellings over $[n]$ is a bijection. To find all strings of sets for $[n]$, the paper introduces a recurrent procedure for construction of all possible strings

of sets for intervals in $[n]$. It starts with a base case of $n = 1$ which has only one string $\boxed{1}\boxed{1}$ and defines inductive rules which describe strings for $[k+1]$ based on the strings for $[k]$. Please note that instead of using l and r for start and end of intervals we simply use its index. There is no loss of generality here, because intervals only start and end once and each interval is assigned to a unique index from $[n]$. The paper denotes the set of strings corresponding to consistent labellings for intervals in $[n]$ as L_n which is the language consisting of all strings over n intervals that are derived using the inductive rules.

This allows the authors to compute all possible strings of sets corresponding to the labellings. However, in order to derive the probabilities of Allen's relations it is necessary to know in which of these labellings some predefined intervals, say 1 and 2, are related by R i.e. $1R2$. The authors introduce a projection operation for a string of sets in order to formally define this. Given set X, $\pi_X(s)$ or X-projection of string of sets $s = a_1a_2...a_n$ is defined using the rules below:

$$\rho_X(s) := (a_1 \cap X)(a_2 \cap X)...(a_n \cap X)$$
$$\pi_X(s) := \text{result of deleting all occurrences of empty set in } \rho_X(s)$$

Let's illustrate the above with a simple example of a string of sets representing one possible labelling of 4 intervals: $\boxed{1,2,4}\boxed{1}\boxed{2,3}\boxed{3}\boxed{4}$. Suppose we are concerned with finding out how intervals 1 and 2 are related in this labelling. We are going to compute $\pi_{\{1,2\}}(\boxed{1,2,4}\boxed{1}\boxed{2,3}\boxed{3}\boxed{4})$ in order to achieve this. $\rho_{\{1,2\}} = (\boxed{1,2,4}\boxed{1}\boxed{2,3}\boxed{3}\boxed{4}) = \boxed{1,2}\boxed{1}\boxed{2}\boxed{}\boxed{}$. After deleting the empty sets we are left with $\boxed{1,2}\boxed{1}\boxed{2}$. This string describes only the relation between intervals 1 and 2 without the additional information about the rest of the intervals. If we relate this string back to Table 2.1 we can see that it corresponds to the "starts" relation.

Finally, the authors can define the probability of Allen's relation over n intervals as the proportion of strings in language L_n for which intervals 1 and 2 are related by R i.e.

$$p_n(R) = \frac{|\{s \in L_n \mid \pi_{\{1,2\}}(s) \text{ corresponds to relation } R\}|}{|L_n|}$$

We are going to omit the derivation of $p_n(R)$ for brevity. However, the most important result is that the probabilities of relations over n interval names also turn out to be

conditioned on whether the relation is short, medium or long. The probabilities for two intervals are $\frac{1}{13}$ for all relations which is expected, since there are 13 strings of sets for two intervals each corresponding to one of Allen's relations.

Limitations of the approach

One of the main limitations of the above approaches arises from the combinatorial treatment of the problem. This results in an implicit independence assumption for the intervals. For example, relations "during inverse" and "overlaps" turn out to have the same probability purely due to the fact that they operate over 4 distinct points on the timeline and therefore are classified as long.

To illustrate this, let's us suppose that intervals represented lifespans of living creatures A and B. $A\,di\,B$ implies that creature A started living before B and creature B died before A did. At the same time, "overlaps" or $A\,o\,B$ suggests that A started living before B and that A died before B did. Having the same probability for these outcomes is unintuitive. One may imagine that, assuming creatures are identical in every way other than their birth times, the one that is born earlier should be more likely to die first.

The above illustration describes the primary limitation of treating intervals in the timelines in an independent manner. Namely, we have no way of remembering the past events at any given point of the timeline which can result in unrealistic outcomes.

2.2.2 Finite temporality strings

Another way to represent timelines is to define which fluents hold at any give point in the timeline. Fluents are temporal propositions, which can be thought of as events. Fernando (2016) discusses defining strings of sets $s = a_1a_2...a_n$, where each of the sets a_i contains symbolic representations of fluents which hold at time point i. The key difference from the above approaches is that we do not consider start and end points of intervals explicitly as part of the string symbols. Instead, each symbol comes from the powerset of 2^F of the set F of all event names or fluents.

For example, let's take timeline we considered in the previous section defined in terms of intervals' start and end points. Its representation is as follows:

$1, 2, 4$	1	$2, 3$	3	4

Let us assign fluents f_1, f_2, f_3 and f_4 from set of event names $F = \{f_1, f_2, f_3, f_4\}$ to

intervals 1, 2, 3 and 4 respectively. The finite temporality string for the timeline in question is going to look like this: $\boxed{}\boxed{f_1, f_2, f_4}\boxed{f_2, f_4}\boxed{f_3, f_4}\boxed{f_4}\boxed{}$. The empty boxes at the start and end of the representation indicate that there exists a definite start and end times for the timeline thus highlighting its finite nature.

Work of Woods et al. (2017) defines several important operations for finite temporality strings, which are instrumental to generating possible timelines of events based on initial knowledge. Let us first explain each of these operations in detail. After that, we are going to give an example of how these operations help us generate timelines.

The first operation is called superposition. Suppose we have a set of fluents F, then the alphabet for our strings will be the power set of F i.e. 2^F. The reason for this is that at each time point there can be an arbitrary collection of fluents that hold so each of the symbols in the string can be one of all possible subsets of the set of event names. Suppose we are given two strings of the same length built using the same alphabet 2^F, $s = \alpha_1, \alpha_2 ... \alpha_n$ and $s' = \alpha'_1, \alpha'_2 ... \alpha'_n$. We can collect information from them into one string over the same alphabet via performing a component wise union:

$$\alpha_1, \alpha_2 ... \alpha_n \,\&\, \alpha'_1, \alpha'_2 ... \alpha'_n := (\alpha_1 \cup \alpha'_1) ... (\alpha_n \cup \alpha'_n)$$

Below we give an example for the superposition operation. Suppose $F = \{f_1, f_2, f_3, f_4\}$ and $s_1 = \boxed{f_1}\boxed{f_2}$ and $s_2 = \boxed{f_3}\boxed{f_4}$, then their superposition $s_1 \& s_2$ is equal to $\boxed{f_1, f_3}\boxed{f_2, f_4}$.
The notion can be extended to languages i.e. collections of strings of sets. Suppose L and L' are languages over a common alphabet Σ which is the powerset of some set of fluents F. The superposition of languages can be defined as the language produced by superposing every pair of strings (s, s') from $L \times L'$, where length of s is equal to length of s'. Below we give a formula for this definition, please note that Σ^n means set of strings of length n over alphabet Σ:

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

Woods et al. (2017) outlines that the key disadvantage of the above operation is that we require the strings to be of the same length. Therefore, they explore the notion of introducing "stutter" to strings. For example, $\boxed{f_1}\boxed{f_2}\boxed{f_2}\boxed{f_2}$ is a stuttered form of $\boxed{f_1}\boxed{f_2}$, where extra occurrences of symbol $\boxed{f_2}$ do not introduce extra information but change the length of the string. They do not introduce extra information, because this string concerns

only two fluents f_1 and f_2, meaning that all we can conclude from both of them is that f_1 happens before f_2. To define this formally, the author suggests the notion of block compression bc:

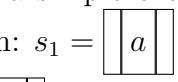
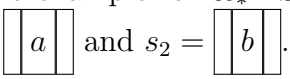
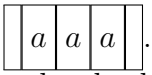
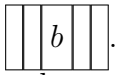
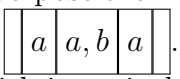
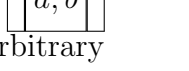
$$bc(s) := \begin{cases} s, \text{ if length(s)} \leq 1 \\ bc(\alpha s'), \text{ if } s = \alpha\alpha s' \\ \alpha\, bc(\alpha' s') \text{ if } s = \alpha\alpha' s' \text{ and } \alpha \neq \alpha' \end{cases}$$

The idea of the above procedure is to remove the first element in the string if it is the same as the next one. Alternatively, we should keep the first symbol and then consider the remainder of the string until there is 1 or 0 symbols left.

Another important notion is the inverse of block compression denoted as $bc^{-1}(s)$. $bc^{-1}(s)$ returns an infinite set of strings which have equivalent block compression. In other words, $bc^{-1}(s)$ introduces all possible configurations of stutter into string s. With the use of bc^{-1}, we can superpose strings of different lengths s and s':

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

Operation $s\&_*s'$ produces a finite set of strings by block compressing the superposition of infinite languages $bc^{-1}(bc(s))$ and $bc^{-1}(bc(s'))$, which correspond to all ways to introduce stutter in strings s and s' respectively.

Let us consider a simple example for $\&_*$. Suppose we have two strings consisting of a single event each: $s_1 =$  and $s_2 =$  . We can introduce stutter to s_1 and make it equal to  . For string s_2 we can repeat its empty boxes like this:  . We can notice that both strings have length of 5 and, therefore, we can superpose them using the component wise union. The result of superposition is going to be:  . This string contains new information. Namely, b is happening during a, which is equivalent to one of Allen's relations. Had we not introduced any stutter at all, we would get  which indicates that a and b are equal.The intuition here is that by introducing arbitrary stutter we can consider all possible temporal configurations that may occur between input timelines. In our case, both input timelines only have information that a and b took place and they are finite, meaning that any one of the 13 Allen's relations can hold between a and b. In fact, if we continue the superposition process for strings s_1 and s_2 we will find out that its result is precisely the set of strings representing each of Allen's relations. This set is illustrated in Table 2.3.

Relation name	Finite temporality string
Equals	$\boxed{\boxed{a,b}}$
Before	$\boxed{\boxed{a}\boxed{b}}$
Before inverse	$\boxed{\boxed{b}\boxed{a}}$
During	$\boxed{\boxed{b}\boxed{b,a}\boxed{b}}$
During inverse	$\boxed{\boxed{a}\boxed{a,b}\boxed{a}}$
Overlap	$\boxed{\boxed{a}\boxed{a,b}\boxed{b}}$
Overlap inverse	$\boxed{\boxed{b}\boxed{a,b}\boxed{a}}$
Meets	$\boxed{\boxed{a}\boxed{b}}$
Meets inverse	$\boxed{\boxed{b}\boxed{a}}$
Starts	$\boxed{\boxed{a,b}\boxed{b}}$
Starts inverse	$\boxed{\boxed{a,b}\boxed{a}}$
Finishes	$\boxed{\boxed{b}\boxed{a,b}}$
Finishes inverse	$\boxed{\boxed{a}\boxed{a,b}}$

Table 2.3: Finite temporality strings representing Allen's relations. Based on table presented in Section 5 of Woods et al. (2017).

The key idea here is that we could introduce a set of initial timelines as strings and superpose them to get all possible timelines that are consistent with our initial assumptions. For example, if one of our initial assumptions is a before b i.e. $\boxed{\boxed{a}\boxed{b}}$, the generated timelines will never contradict this. The intuition here is that regardless of the structure of other strings and the stutter introduced the order fluents a and b will be preserved.

However, if strings share the same fluents, which is a common case when we want to do reasoning about timelines, the consistency may be lost. For example, if we have strings $\boxed{\boxed{x}\boxed{y}}$ and $\boxed{\boxed{y}\boxed{z}}$, Woods and Fernando (2018) suggests that the results of $\&_*$ may contain strings that do not correspond to initial assumptions.

In order to map from the superposed string to its initial assumptions, Woods and Fernando (2018) discusses a projection operation somewhat similar to the one we looked at in the section on prior probabilities of Allen's relations. This operation is also based on the notion of reduction:

$$\rho_A(\alpha_1\alpha_2...\alpha_n) := (\alpha_1 \cap A)...(\alpha_n \cap A)$$

We refer to ρ_A as the A-reduct of $\alpha_1\alpha_2...\alpha_n$. String s is said to project to another string s' if $bc(\rho_{voc(s')}(s)) = s'$, where $voc(s')$ is the set of all fluents in $s' = \alpha'_1...\alpha'_n$ or $\bigcup_i^n \alpha'_i$. Now we can formally define consistency as the fact that the superposed string should project into its initial assumptions.

For example, if our initial timelines were $\boxed{\boxed{x}\boxed{y}}$ and $\boxed{\boxed{y}\boxed{z}}$, some of the possible superpositions according to the definition of $\&_*$ may include $\boxed{\boxed{x}\boxed{y}\boxed{z}}$ and $\boxed{\boxed{y}\boxed{x}\boxed{z}\boxed{y}}$. The first superposition is actually consistent, because it projects back into our initial timelines as well as $\boxed{\boxed{x}\boxed{z}}$ which arises due to transitivity. However, the second one is invalid. Some of the strings it projects into are $\boxed{\boxed{y}\boxed{x}\boxed{y}}$ $\boxed{\boxed{y}\boxed{z}\boxed{y}}$, which are contradictory since they include fluent y twice suggesting that y happened before y.

In order to tackle this problem, Woods et al. (2017) imposes constraints on the generated strings, while Woods and Fernando (2018) introduces the notion of vocabulary constraint superposition. For brevity, we are going to omit its details here.

Suliman (2021) develops a Python framework which allows for manipulating finite temporality strings including the operation of vocabulary constrainted superposition. They utilize this framework to attempt to find the probabilities of timelines given some set of events. In doing so, they resort to stochastically generating well formed strings using a Markovian process. Markovian here means that the future of the timeline only depends on its current state and nothing else. This results in the same limitation as we saw in the previous section. For example, if we end up in a state when both creatures are alive they are going to have the same probability of dying despite one of them being born earlier than the other. At each given state, the transitions in the Markov Chain are given by one of possible valid continuations of the timeline. If we put aside probabilities, such construction can be thought of as a finite state machine. In the next subsection, we are going to explain this construction by looking at timelines for simple events.

2.2.3 Finite state machine representation

For each simple event with no underlying structure, there are fundamentally 3 possible states: event is not started, event is happening and event is finished. In finite temporarlity strings, each symbol corresponds to the fluents which are happening at its time point. Therefore, Suliman (2021) suggests a finite state machine construction where each state represents the fluents that are happening at the current time point and also the fluents

which have already concluded. The latter is required to ensure that events are not re-introduced in the further states. Let's define each state as a triple (NS, S, F), where NS indicates the set of all fluents that are not started, S indicates fluents that started by not yet concluded and F indicates fluents that have finished. A careful reader might notice that NS is redundant if we know the set of all possible fluents A since $NS = A \setminus (S \cup F)$. Transitions in each state are defined as possible changes to the events. Each transition symbol is represented by sets of fluents S_t and F_t, where represents S_t represents the fluents from NS that are going to start and F_t represents the fluents from S that are going to end. Formally, we have a transition:

$$(NS, S, F) \rightarrow_{S_t, F_t} (NS \setminus S_t, (S \cup S_t) \setminus F_t, F \cup F_t)$$

Suliman (2021) illustrates this construction by looking at lifespans of two creatures a and b. We start in a state when both creatures are unborn i.e. u_a and u_b. At any given point, unborn creatures may become living by following transitions l_a or l_b (live a and live b) respectively. Additionally, any of the living creatures may die which is represented by transitions r_a and r_b (rest a and rest b). Figure 2.3 presents the state machine we just described. Please note this definition is slightly different from the initial one we gave above, because we just keep the current state of each of the creatures as opposed to keeping a list of unborn, living and dead creatures in each state of the finite automata. This makes the notation easier to follow.

If we look at Figure 2.3 carefully, we can see that the 13 paths from the starting state to the final state i.e. the 13 accepted words represent each of 13 Allen's relations. This is expected, since the finite state machine construction mimicked the structure of finite temporality strings defined by superposition of two simple events such as $\boxed{\boxed{a}}$ and $\boxed{\boxed{b}}$. The words accepted by automata, however, are closer to the representation presented in Table 2.1, since the transitions are concerned with the start and finish of the intervals. For example, the path for equals is la, lb followed by ra, rb, which can be converted to string notation as follows: $\boxed{la, lb}\boxed{ra, rb}$. At the same time, if we look at the states along the path, we can get finite temporality strings. For example, the "equals" path goes through states (u_a, u_b), (li_a, li_b) and (d_a, d_b), which in order correspond to $\boxed{\boxed{a, b}}$ i.e. no event happening, both a and b happening, no event happening. Table 2.4 lists each of the paths in the finite state automata from Figure 2.3 and their corresponding Allen's relations.

The correspondence of finite temporality strings to finite state machines is an important

foundation of our work. It allows us to reason about timelines in a probabilistic manner, because we can assign probabilities to the transitions. For example, one of the approaches here is running Monte-Carlo simulations on the finite automata that represent the timelines of interest. Moreover, finite automata representation allows us to consider events with underlying structure. For example, we might want to consider lifespans of creatures in terms of their age which requires a more sophisticated event model rather than defining events as simple intervals.

In finite temporality strings, we use the notion of superposition to derive timelines for events or more generally other timelines. It is natural to pose a question of whether we can define events using finite state machines and perform a similar superposition operation but directly with the finite automata. We are going to discuss this problem in the next chapter.

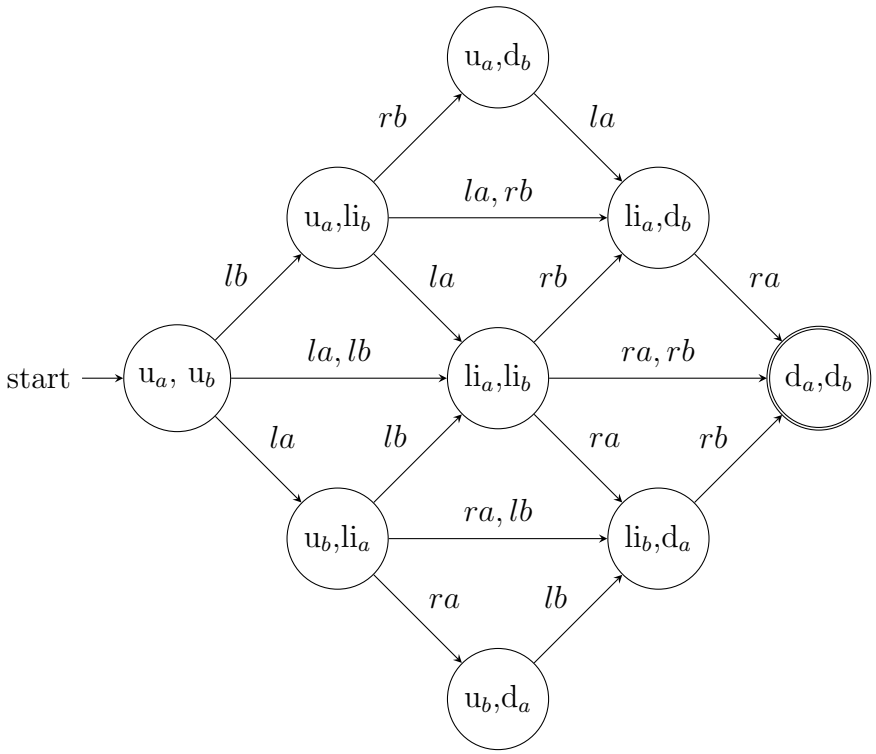


Figure 2.3: Finite state machine representing timelines of fluents for two living creatures. Based on the diagram presented in Section 4.2 of Suliman (2021)

Relation name	Path in the finite state machine in Figure 2.3
Equals	la, lb | ra, rb
Before	la | ra | lb | rb
Before inverse	lb | rb | la | ra
During	lb | la | ra | rb
During inverse	la | lb | rb | ra
Overlap	la | lb | ra | rb
Overlap inverse	lb | la | rb | ra
Meets	la | ra, lb | rb
Meets inverse	lb | rb, la | ra
Starts	la, lb | ra | rb
Starts inverse	la, lb | rb | ra
Finishes	la | lb | ra, rb
Finishes inverse	lb | la | rb, ra

Table 2.4: Illustration of how paths in finite state machine in Figure 2.3 correspond to Allen's relations.

2.3 Summary

We looked at several approaches for representing Allen's relations and timelines. The work of Fernando and Vogel (2019) used strings of sets, where elements in the sets represented starting and finishing points of the intervals. This paper classified Allen's relations as short, medium and long. The prior probabilities of Allen's relations over n ordered points and n interval names were found to be conditioned by the length of the relations. We presented some of the limitations of this approach that arise due to its combinatorial treatment of the problem.

Another commonly used approach was finite temporality strings. We explored the common operations on these strings leading up to the operation of superposition that allows us to reason about timelines. This operation was thoroughly discussed in Woods et al. (2017) and Woods and Fernando (2018). After that, we looked at the work of Suliman (2021) which developed a finite temporality reasoning framework in Python and, as part of that, linked finite temporality strings to finite state machines in order to simulate timelines using a Markovian process. In the case of finite temporality strings and their representation as automata, we observed the same limitation of the future only depending

on the current state as in Fernando and Vogel (2019). We also proposed that finite state machines might be a useful tool in modelling events with complex structure and that the notion of superposition should be easy to extend directly to automata which will allow us to reason about timelines of complex events from a more probabilistic perspective via assigning probabilities to transitions.

Overall, this section helped us find gaps and areas of interest in the existing work which will help us formulate a concrete problem within the context of the literature in the next chapter.

Chapter 3

Design

In this chapter, we are going to outline our research problem by defining several research directions based on the challenges identified during literature review. After that, we will introduce the notion of automata superposition and, hence, perform an initial analysis of the probabilities for the during and overlap relations. Based on this analysis, we will define the software that needs to be developed. The chapter will end with a detailed discussion of the key algorithms and data structures that are required to build the proposed software system.

3.1 Problem Formulation

This section is going to summarize the challenges that we identified during the literature review. It will then present concrete research directions based on the challenges and discuss how we approached them.

3.1.1 Identified Challenges

As we saw in the previous chapter, one of the disadvantages of existing approaches is that the timeline's future is only dependent on its current state and not on its prior history. This may cause unintuitive probability results such as Allen's relation overlap having the same probability as during. We discussed why this is unintuitive in the previous chapter by looking at a thought experiment consisting of lifespans of two living creatures.

Additionally, we saw that we can represent timelines using finite state machines and that such representation allows us to reason about them in a probabilistic manner by assigning probabilities to transitions. We proposed that finite state machines might be

a useful tool for modelling probabilities of timelines consisting of complex events. To achieve this, we highlighted the need to expand the notion of superposition to the domain of automata.

3.1.2 Proposed Work

Based on the identified challenges, we propose 3 research directions (RD_1, RD_2, RD_3):

- Extend the notion of superposition to the domain of automata - RD_1
- Define a timeline representation model where future is dependent on the current state and the past. Evaluate this model by looking at probabilities of Allen's relations, namely overlap and during. We should expect that overlap is more probable than during - RD_2
- Propose a model for reasoning about probabilities of timelines of complex events using finite state machines - RD_3

In this chapter, we are going to discuss RD_1 in detail. We are also going to analyze probabilities of overlap and during in the context of RD_2 which will help us propose the design of the software component that will enable us to approach RD_2 and RD_3. The discussion of RD_2 and RD_3, however, is not possible without explaining the software implementation, so we will postpone this discussion to Chapter 5.

3.2 Overview of the Design

This section will begin with the initial analysis we performed on the probabilities of Allen's relations. We are going to show that the unintuitive probabilities of overlap and during relations cannot be fixed via assignment of prior probabilities. Therefore, we are going to propose two alternative approaches based on the automata superposition. It will become clear that we require a software system in order to evaluate these approaches. This section will conclude with a detailed analysis of the algorithms and data structures required to develop the aforementioned software.

3.2.1 Defining superposition for finite state machines

We start by extending the notion of superposition to finite state machines. This is a required step in understanding how finite state machines for timelines are generated from the automata representing individual events.

Let us assume we are given finite automata $M = \langle \rightarrow, F, q_0 \rangle$ and $M' = \langle \rightarrow', F', q_0' \rangle$, where $\rightarrow$ and $\rightarrow'$ are transition functions, F and F' are the sets of final states and q_0 and q_o' are the initial states. We define their superposition $M \& M' = \langle \rightsquigarrow, F \times F', (q_0, q_0') \rangle$ using the three rules listed below. In these rules, A and A' indicate the alphabets for the corresponding automata, while α and α' are the individual transition symbols coming from these alphabets[1].

$$\frac{P(q,q') \quad q \xrightarrow{\alpha} r \quad q' \overset{\alpha'}{\rightarrow'} r' \quad \alpha \cap A' \subseteq \alpha' \quad \alpha' \cap A \subseteq \alpha}{P(r,r') \quad (q,q') \overset{\alpha \cup \alpha'}{\rightsquigarrow} (r,r')}, \text{ rule 1}$$

$$\frac{P(q,q') \quad q \xrightarrow{\alpha} r \quad \alpha \cap A' = \varnothing}{P(r,q') \quad (q,q') \overset{\alpha}{\rightsquigarrow} (r,q')}, \text{ rule 2}$$

$$\frac{P(q,q') \quad q' \overset{\alpha'}{\rightarrow'} r' \quad \alpha' \cap A = \varnothing}{P(q,r') \quad (q,q') \overset{\alpha'}{\rightsquigarrow} (q,r')}, \text{ rule 3}$$

The idea of this superposition construction is to let the original finite state machines execute side by side. Let's look at each of the rules in detail to understand the definition more intuitively. Rule 1 suggests that if state q is followed by state r via transition α in M and state q' is followed by state r' via transition α' in M' then the superposed automata is going to have a transition from state (q, q') to state (r, r') with symbol $\alpha \cup \alpha'$. To put this into context of events, if the two original automata had transitions which indicated the start of some events e_1 and e_2, the superposed automata will have a transition corresponding to both events starting at the same time. Please note that events e_1 and e_2 starting at the same time is only one of the timeline possibilities. Rules 2 and 3 are symmetrical. Each of them consider the cases when a transition happens in one of the automata but not in the other. In case of events, this could mean that one of the events starts or finishes without any change to the other event. Additionally, there are constraints that are imposed on the symbols α and α' from the alphabets. These constraints are imposed to remove potential ambiguity when the alphabets of the initial automata have shared symbols.

Let's now observe how this definition works in practice. Suppose we have two simple automata representing the lifespans of creatures. Figure 3.1 illustrates these automata. The starting state for the superposed automata is going to be (u_a, u_b) where both creatures are unborn. Rule 1 suggests that we can go to (li_a, li_b) from this state, while rules 2 and

[1]The rules are courtesy of Dr. Tim Fernando

3 suggest that we can go to states (li_a, u_b) and (u_a, li_b) respectively. The corresponding transition symbols for these states are going to be $\{la, lb\}$, la and lb respectively. If we apply the rules further based on the new states that we found, we are going to construct a finite state machine equivalent to one from Figure 2.3. This highlights that the proposed definition of finite automata superposition is reasonable and behaves similarly to the superposition of strings of sets.

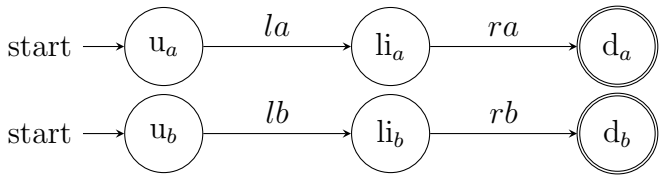


Figure 3.1: Basic automata representing lifespans of creatures a and b. Labels u_a and u_b represent unborn states, li_a and li_b represent living states, and d_a and db represent the dead states. Transitions la and lb mean start to live, while transitions ra and rb mean rest.

Being able to construct finite state representations of timelines from the automata representing the individual events is an important first step because it allows us to assign probabilities to the derived transitions based on the probabilities of transitions in the original finite state machines. We are going to explore this further in the next subsection, where we present an initial discussion of probabilities of Allen's interval relations based on the automata from Figure 2.3.

3.2.2 Initial analysis of probabilities for overlap and during

The goal of this subsection is to explore probabilities of Allen's interval relations by looking at various ways in which we can assign transition probabilities to the finite state automata from Figure 2.3.

The most trivial choice is to assign probabilities uniformly to each of the transitions in a given state. This means if we have N transitions the probability of each one of then is going to be $\frac{1}{N}$. Such assignment was first introduced in Suliman (2021). Figure 3.2 illustrates the assignment. In this case we can see that probabilities of both overlap and during are $\frac{1}{27}$ since they have to move through four transitions where the first three transitions have probability of $\frac{1}{3}$, while the final one is deterministic with probability of 1. The probabilities of during and overlap are the same because there is no distinction between prior paths once we reach state (li_a, li_b). In other words, we do not know which event started first and therefore the probabilities of them concluding are the same.

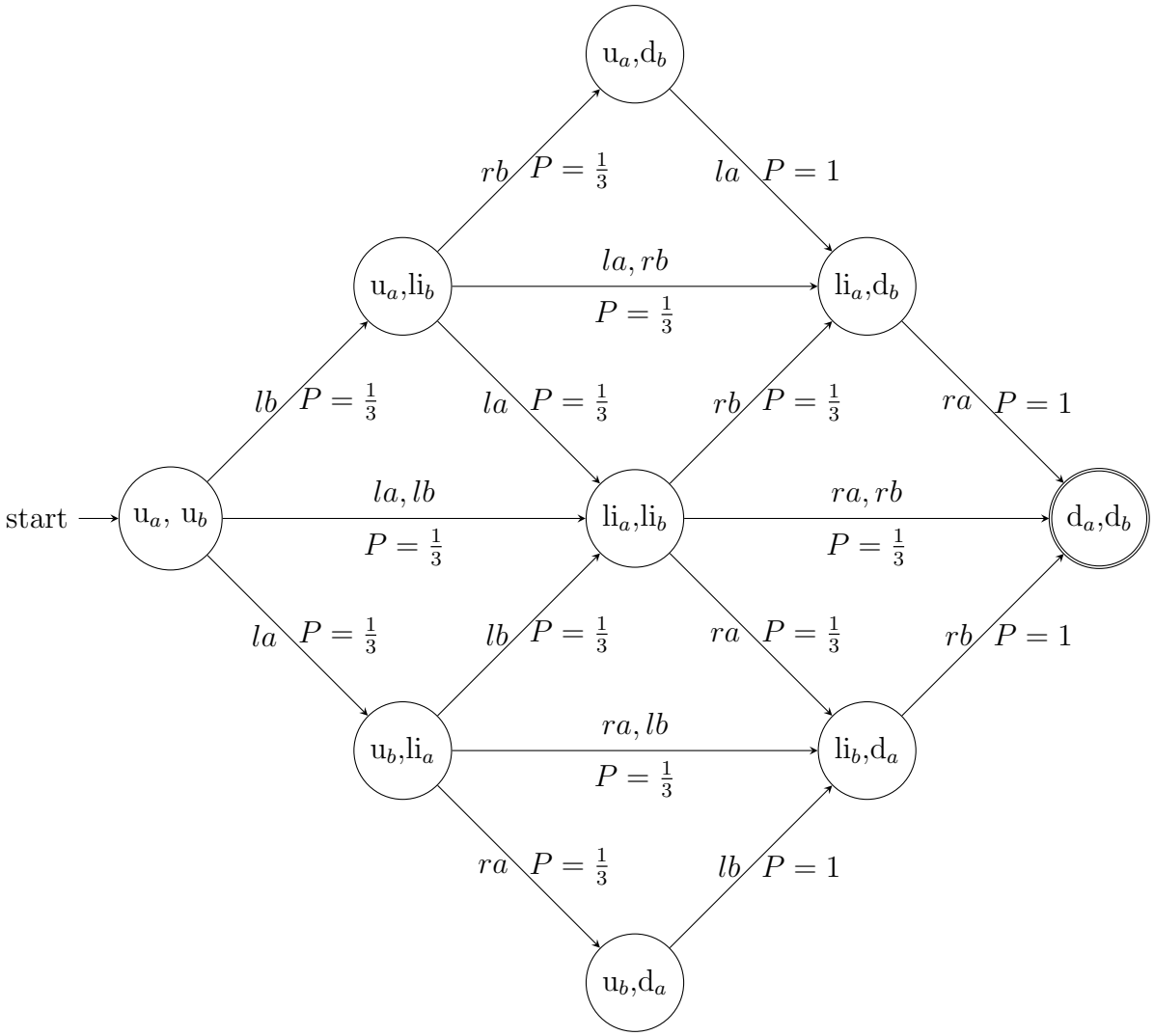


Figure 3.2: Finite state machine representing Allen's interval relations with uniformly assigned probabilities. The diagram is based on Section 4.2 from Suliman (2021)

Another possible assignment of probabilities arises when we introduce transitions probabilities to the initial automata from Figure 3.1. Figure 3.3 shows the initial automata with transitions probabilities introduced. When we perform superposition of these automata in each state we are presented with several possibilities: both automata transition (rule 1), one of them transitions (rule 2 or 3) or none of them transition. Let's consider (u_a, u_b) as an example. As we saw before, we can arrive to states (li_a, li_b), (li_a, u_b) and (u_a, li_b) directly from (u_a, u_b).

Assuming that events are independent of one another, we can compute probabilities of transitions in the superposed automata by multiplying the corresponding probabilities from the initial automata. For example, transition to state (li_a, li_b) will have the probability of $p_a p_b$, while transition to (li_a, u_b) will have the probability of $p_a(1 - p_b)$. If we assign probabilities in this way, it will quickly become clear that probabilities of transitions for a given state do not add up to 1. The reason for this is that we omit the self loops which take up the remaining portion of the probability i.e. $(1 - p_a)(1 - p_b)$ for state (u_a, u_b). The probabilities of transitions in the original automata did not add up to 1 for the same reason. One way to fix this in the superposed automata is to normalize probabilities i.e. divide each transition probability by the sum of all transitions coming from a given state. If we normalize probability, we are essentially disallowing stalling which is a notion similar to stuttering in finite temporality strings. In Figure 3.4, we illustrate how the probabilities of initial automata allow us to derive the probabilities for the superposed finite state machine.

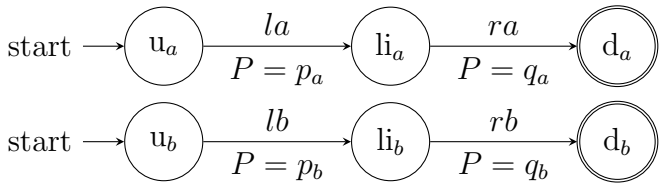


Figure 3.3: Basic automata representing lifespans of creatures from Figure 3.1 with transition probabilities introduced. Probabilities of becoming alive (event starting) are p_a and p_b, while probabilities of dying (event finishing) are q_a and q_b.

We already have the intuition that overlap and during have the same probability due to the indifference to history that arises in state (li_a, li_b) and not due to the assignment of probabilities to transitions. However, it is still useful to verify that for an arbitrary assignment of probabilities we see the same situation as for the uniform assignment we described previously. We are going to perform the analysis both for the not normalized case which allows stalling and the normalized case.

Probability of overlap and during for automata from Figure 3.4

We are going to analyze the probabilities of the following relations: A overlap B and B during A. In other words, we are going to compare probabilities of the following paths in the automata from Figure 3.4: $\boxed{la}\boxed{lb}\boxed{ra}\boxed{rb}$ and $\boxed{la}\boxed{lb}\boxed{rb}\boxed{ra}$.

Probability of initial la transition is $p_a(1-p_b)$ i.e. a starts and b does not. For transition lb, it is $p_b(1-q_a)$ i.e. b starts and a does not end. Transition ra after la and lb has a probability of $q_a(1-q_b)$ which is a ends and b does not. Similarly, for rb after la and lb the probability is $q_b(1-q_a)$. The final transitions have probabilities of q_a and q_b respectively.

Assuming independence, we can multiply the probabilities at each symbol in each of the strings representing the relations. For overlap, we get $p_a(1-p_b)p_b(1-q_a)q_a(1-q_b)q_b$, which can be rearranged as $p_ap_bq_aq_b(1-p_b)(1-q_a)(1-q_b)$. Similarly, for duration the result is going to be $p_a(1-p_b)p_b(1-q_a)q_b(1-q_a)q_a$ which is equal to $p_ap_bq_aq_b(1-p_b)(1-q_a)^2$.

We also need to take potential stalls in each of the states in the path into account. In state, (u_a, u_b), the probability of stalling is $(1-p_a)(1-p_b)$. In state, (li_a, u_b) it is $(1-q_a)(1-p_b)$. In state, (li_a, li_b) it is $(1-q_a)(1-q_b)$. In state (li_a, d_b) it is $1-q_a$. In state, (d_a, li_b) it is $1-q_b$.

For overlap, we need to take stalling in states (u_a, u_b), (li_a, u_b), (li_a, li_b) and (d_a, li_b) into account. Assuming stalling occurs k_1, k_2, k_3 and k_4 number of times respectively for each of the states. The probability is going to be $((1-p_a)(1-p_b))^{k_1}((1-q_a)(1-p_b))^{k_2}((1-q_a)(1-q_b))^{k_3}(1-q_b)^{k_4}$

For duration, it is similar, but the last state is li_a, d_b, so the stalling probability is going to be $((1-p_a)(1-p_b))^{n_1}((1-q_a)(1-p_b))^{n_2}((1-q_a)(1-q_b))^{n_3}(1-q_a)^{n_4}$. Using n_1, n_2, n_3 and n_4 to highlight difference from the k's.

We can now express the probabilities of duration and overlap given the number of stalls defined by k's and n's:

$$P_{overlap}(k_1,k_2,k_3,k_4) = p_ap_bq_aq_b(1-p_b)(1-q_a)(1-q_b) \\ ((1-p_a)(1-p_b))^{k_1}((1-q_a)(1-p_b))^{k_2}((1-q_a)(1-q_b))^{k_3}(1-q_b)^{k_4}$$

$$P_{duration}(n_1, n_2, n_3, n_4) = p_a p_b q_a q_b (1-p_b)(1-q_a)^2 \\ ((1-p_a)(1-p_b))^{n_1}((1-q_a)(1-p_b))^{n_2}((1-q_a)(1-q_b))^{n_3}(1-q_a)^{n_4}$$

The overall probabilities of duration and overlap can be found by summing over all possible values for the stalling counts. Let us start with the probability of overlap:

$$P_{overlap} = \sum_{k_1=0}^{\infty}\sum_{k_2=0}^{\infty}\sum_{k_3=0}^{\infty}\sum_{k_4=0}^{\infty} P_{overlap}(k_1, k_2, k_3, k_4)$$

$$P_{overlap} = p_a p_b q_a q_b (1-p_b)(1-q_a)(1-q_b) \\ \sum_{k_1=0}^{\infty}\sum_{k_2=0}^{\infty}\sum_{k_3=0}^{\infty}\sum_{k_4=0}^{\infty}((1-p_a)(1-p_b))^{k_1}((1-q_a)(1-p_b))^{k_2}((1-q_a)(1-q_b))^{k_3}(1-q_b)^{k_4}$$

Noting that for $0 < x < 1$ the sum of infinite geometric series has a closed form i.e.

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x},$$

we can simplify $P_{overlap}$ by using the distributivity of sum over products:

$$P_{overlap} = p_a p_b q_a q_b (1-p_b)(1-q_a)(1-q_b) \\ \frac{1}{1-(1-p_a)(1-p_b)}\frac{1}{1-(1-q_a)(1-p_b)}\frac{1}{1-(1-q_a)(1-q_b)}\frac{1}{q_b}$$

Using the same simplification procedure, we can obtain $P_{duration}$

$$P_{duration} = p_a p_b q_a q_b (1-p_b)(1-q_a)^2 \\ \frac{1}{1-(1-p_a)(1-p_b)}\frac{1}{1-(1-q_a)(1-p_b)}\frac{1}{1-(1-q_a)(1-q_b)}\frac{1}{q_a}$$

If $q_a = q_b = q$, these probabilities are the same. This means that if probabilities of a and

b ending are the same, the probabilities of overlap and duration Allen relations are also going to be the same.

Now, let's consider the case when $q_b < q_a$. This similar to assuming that if event a starts earlier than b we would expect it to have a higher probability to end than that of b.

Let's compare probabilities of duration and overlap in this case. We can remove the common factor of $p_a p_b (1-p_b) \frac{1}{1-(1-p_a)(1-p_b)} \frac{1}{1-(1-q_a)(1-p_b)} \frac{1}{1-(1-q_a)(1-q_b)}$ which would leave us with 2 quantities to compare $q_a(1-q_b)$ and $q_b(1-q_a)$.

Since $q_a > q_b$, we know that $1-q_b > 1-q_a$. We can multiply these inequalities, since all numbers are positive probabilities. Hence, we get $q_a(1-q_b) > q_b(1-q_a)$. This means that when $q_a > q_b$ overlap is more probable than duration.

This, however, is a rather artificial way to derive that overlap is more probable than duration. For example, if the two events are of the same nature, we might expect that their prior probabilities of starting and ending are the same. At the same time, if one event starts before another, we expect that there is a higher chance that the first event concludes before the second one. A simple example could be expecting that if someone is born significantly earlier than another person, they would be expected to die earlier as well.

Additionally, if we consider relations B overlap A and A during B the assumption of $q_a > q_b$ is going to have a reverse effect, resulting in probability of duration being higher than the probability of overlap. This is unintuitive, since in case of the latter two relations event B starts first and not event A. In order to fix that problem we would have to set $q_b > q_a$ which is obviously incompatible with our initial assumption.

Probability of overlap and during with normalized transition probabilities

The case of normalized probability does not involve stalling, so it is simpler to consider. The probability of overlap is going to be $\frac{p_a p_b q_a (1-p_b)(1-q_a)(1-q_b) q_b}{N_{(u_a,u_b)} N_{(u_b,li_a)} N_{(li_a,li_b)} N_{(d_a,li_b)}}$ where $N(s)$ indicates the normalization factor for state s which is equal to the sum of transition probabilities from that state. Similarly, duration probability will be equal to $\frac{p_a p_b q_b (1-p_b)(1-q_a)^2 q_a}{N_{(u_a,u_b)} N_{(u_b,li_a)} N_{(li_a,li_b)} N_{(li_a,d_b)}}$. $N_{(d_a,li_b)}$ is equal to q_b while $N_{(li_a,d_b)}$ is equal to q_a. Hence, we can simplify both equations to get:

$$P_{overlap} = \frac{p_a p_b q_a (1-p_b)(1-q_a)(1-q_b)}{N_{(u_a,u_b)} N_{(u_b,li_a)} N_{(li_a,li_b)}}$$

$$P_{duration} = \frac{p_a p_b q_b (1-p_b)(1-q_a)^2}{N_{(u_a,u_b)} N_{(u_b,li_a)} N_{(li_a,li_b)}}$$

If $q_a = q_b$ the probabilities of overlap and during are going to be the same. If we assume $q_a > q_b$, we are going to get a result similar to the previous case where the probability of overlap was higher than the probability of during for relations A overlap B and B during A.

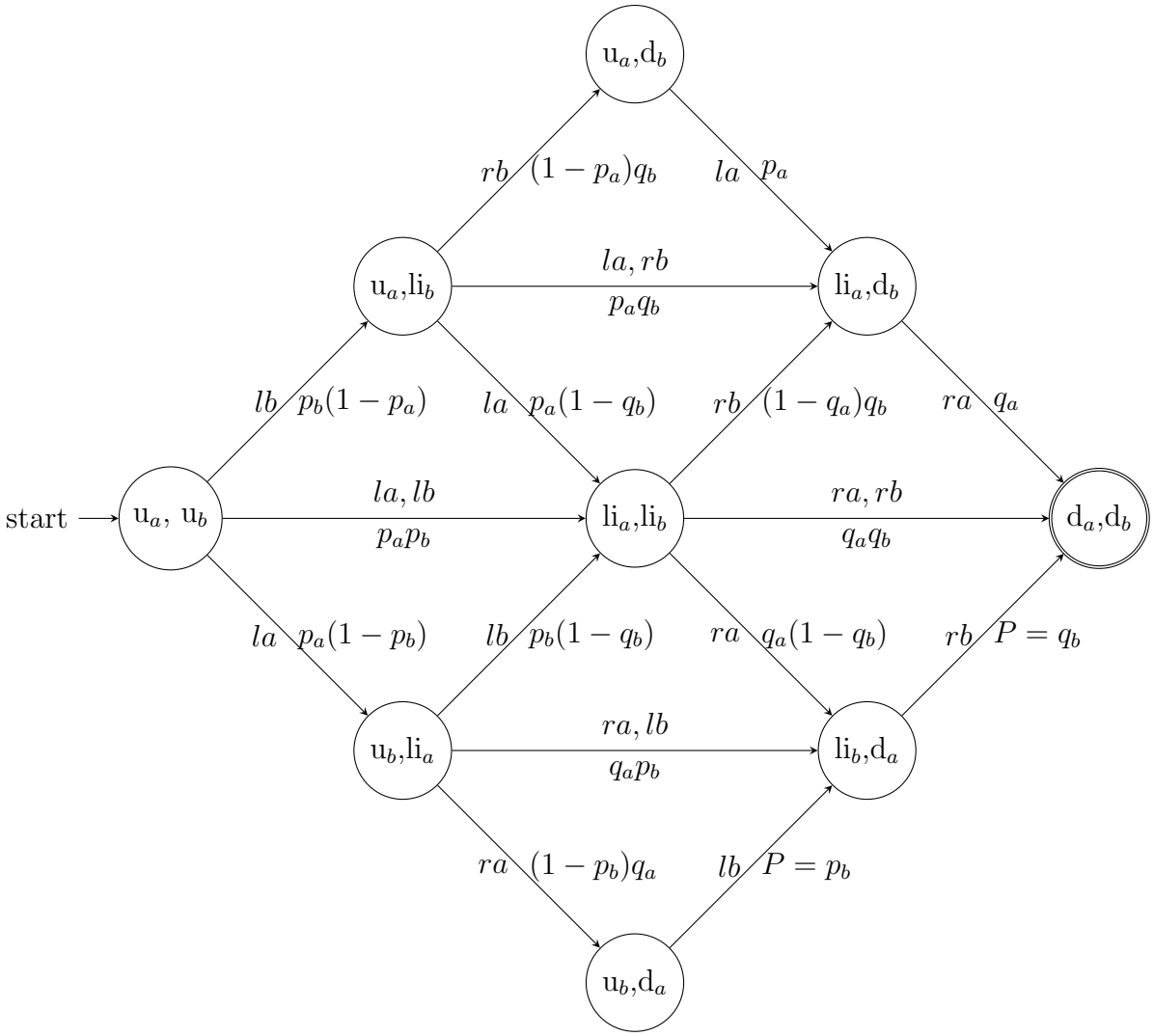


Figure 3.4: Finite state machine representing Allen's interval relations with not normalized transition probabilities derived from the transition probabilities of initial automata from Figure 3.3.

Analysis of the probability assignments

We saw that our intuition was right, meaning that assignment of probabilities does not influence the relationship between probabilities of during and overlap unless we make artificial assumptions about the probabilities of dying (events finishing). This suggests that solving the problem will require changing the structure of the finite state machine so that the history of the path has some impact on the probabilities of relations.

We propose two approaches to this challenge. The first one is based on extending the notion of states via superposing the automata from Figure 3.4 with another automata. For example, we could superpose the Allen's relation automata with a finite state machine representing a clock which will allow us to measure how much time passed before the current state was reached. The second one involves introducing granularity to the initial events i.e. multiple living states corresponding to being "young" and "old".

Regardless of the approach taken, we realize that we require a tool for defining and superposing arbitrary automata. Therefore, in the next subsection we are going to discuss the design of the software we developed for working with finite state machines and their superposition.

3.2.3 Software design

This subsection will being with us describing the data model required to represent finite state machines efficiently. After that, we are going to introduce the algorithms for superposing automata and for simulating automata in order to generate timelines. The subsection will conclude with a discussion on visualizing automata and user interface design.

Defining automata in software

Finite state machines are defined using four concepts:

- Starting state q_o
- Set of final states F
- Alphabet A
- Transition function $T : s \times \alpha \to s'$, where s is the current state, α is a symbol from A and s' is the next state

We choose to distinguish between states using string labels. This means that the starting state label is going to be represented as a single variable of type string which can only

be assigned once at the initialization stage of the state machine. Another possible way is to differentiate states using indices. However, this makes human interpretation difficult because during superposition the states become more complicated. For example, having state named (li_a, li_b) is a lot more informative than using notation $(2, 2)$. One might argue that we could translate indices to human readable names at visualization stage. While this is a reasonable approach, it will most certainly introduce unnecessary complexity to the code and make it a lot harder to maintain. Given that there is no significant performance gain from using indices if we consider modern computational power, we chose to use strings as labels for the states.

In terms of the set of final states, the most reasonable choice here is going to be a hash set of strings which stores the labels of all final states. Hash sets are hash tables where key and value are the same. Hash tables allow for amortized $O(1)$ lookup and add operations as described in a well-known algorithms textbook by Sedgewick and Wayne (2024), which makes them really useful for our application. The reason for this is that the most frequent operation for the set of final states is checking whether a given state label is final or not.

Similar to state labels, we are going to represent symbols in the alphabet using strings. In our design, alphabet does not represent all allowed symbols for the finite state machine. It is actually a hash set that contains all of the symbols that are currently used in some transition within the automata. Alphabets do not have much use for our further algorithms and we choose to maintain them for completeness and easier debugging.

Maintaining information about transitions is closely tied to how we are storing the states. As discussed before each state is identified by a string label. However, states also require other information to be stored with them, for example, the transition function. We choose to represent state as a custom datatype i.e. class in object-oriented paradigm. The states are stored in a hash table which maps each state label to the corresponding state object. Each state object has its label and a transition function. The transition function is another hash table which maps each symbol to the state label of its corresponding next state. Such representation allows for quick simulation operations because looking up the next state can be done in $O(1)$. Additionally, it makes it easy to access each of the states based on its label for visualization purposes. An schematic example of this data structure is shown in Figure 3.5, while a visualization of the represented state machine is presented in 3.6. Please note that we are not concerned with which state is starting or which states are final in this representation because this information is maintained in separate data structures that we discussed above.

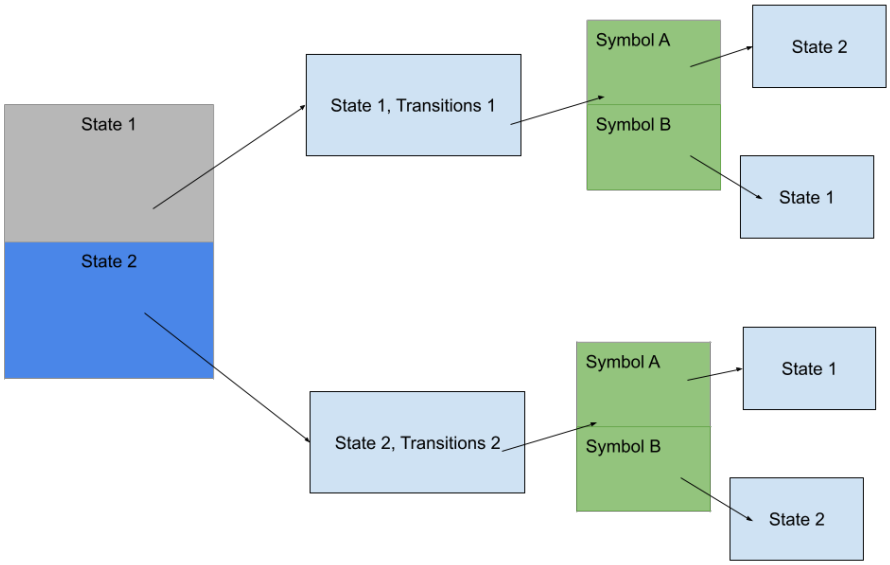


Figure 3.5: Design of state and transitions data structure.

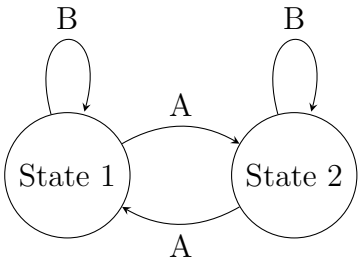


Figure 3.6: State machine that is represented by the data structure in Figure 3.5.

Superposing finite state machines

Now that we defined the representation of finite state machines, we can start defining the algorithm for their superposition. The idea of our algorithm is based on Breadth-First Search (BFS) where nodes of the graph are the states in the superposed automata. Suppose we are given finite automata $M = \langle \rightarrow, F, q_0 \rangle$ and $M' = \langle \rightarrow', F', q_0' \rangle$. We build the superposed automata starting from its initial state (q_0, q_0'). As we saw previously, at any state in the superposed automata we are presented with 3 choices:

- Let automata 1 transition
- Let automata 2 transition
- Let both automata transition

We have to consider all of the choices. Hence, we look at all possible transitions that automata 1 and automata 2 could make on their own as well as each possible combination of transitions for automata 1 and 2. The procedure of finding all possible next nodes given the current node while doing some processing on it is often referred to as node expansion. Figure 3.7 provides our expansion procedure. Please note that the probability of transitions in the superposed automata is computed in terms of stall probabilities p_{stall} and p'_{stall} when only one of the initial automata transitions. The stall probabilities are the probabilities that the automata is not going to transition in its current state. If both automata transition, we assign the probability in the superposed automata to the product of the probabilities of transitions in the initial automata. Such definition of probabilities does not consider normalization, although the probabilities could be optionally normalized after the superposed automata $M\&M'$ is generated.

With the expansion procedure defined, we can finally present the superposition algorithm. Figure 3.8 presents the pseudo code. As mentioned before, the idea is to traverse the superposed automata states in a BFS manner until all of the superposed states were generated. A careful reader might notice that if at least one of the automata has loops, this algorithm will end up in an infinite execution cycle. In our analysis in Chapter 5, we are only considering automata that have no loops. However, it is trivial to extend this algorithm to work for automata with loops. We need to introduce a visited array data structure. Everytime a given state in the superposed automata is fully expanded i.e. we have generated all of its transitions and considered all of its next nodes, we are going to add it to the visited array. When we are performing expansion we will ignore the node completely if it was already visited.

For example, if some state has a self loop in the superposed automata, it is still going to be re-added to the queue after it is expanded because the new nodes it generates contain itself. However, when we are taking it out of the queue for the second time, we are going to see that it was already expanded since it is part of the visited array and therefore are going to ignore it.

Algorithm 1 Node expansion procedure

1: **procedure** EXPAND($M, M', M\&M', (q_{current}, q'_{current})$)
2: $next \leftarrow [\,]$
3: $p'_{stall} \leftarrow 1 - \sum_{t' \in M'.T[q'_{current}]} t'.probability$
4: $p_{stall} \leftarrow 1 - \sum_{t \in M.T[q_{current}]} t.probability$
5:
6: **for all** $s_{next}, \alpha, p_{next}$ in $M.T[q_{current}]$ **do** ▷ T indicates transition function
7: $next.append((s_{next}, q'_{current}))$
8: $M\&M'.T.add((q_{current}, q'_{current}), (s_{next}, q'_{current}), \alpha, p_{next}p'_{stall})$
9: **end for**
10:
11: **for all** $s'_{next}, \alpha', p'_{next}$ in $M'.T[q'_{current}]$ **do**
12: $next.append((q_{current}, s'_{next}))$
13: $M\&M'.T.add((q_{current}, q'_{current}), (q_{current}, s'_{next}), \alpha', p'_{next}p_{stall})$
14: **end for**
15:
16: **for all** $s_{next}, \alpha, p_{next}$ in $M.T[q_{current}]$ **do**
17: **for all** $s'_{next}, \alpha', p'_{next}$ in $M'.T[q'_{current}]$ **do**
18: $next.append((s_{next}, s'_{next}))$
19: $M\&M'.T.add((q_{current}, q'_{current}), (s_{next}, s'_{next}), \alpha \cup \alpha', p_{next}p'_{next})$
20: **end for**
21: **end for**
22: **return** $next$ ▷ Next nodes to consider
23: **end procedure**

Figure 3.7: Pseudo code for the expansion procedure. Please note we use $\alpha \cup \alpha'$ as a symbol when both automata transition. Union operation is not defined on strings, so we have to use a custom implementation which will be discussed in Chapter 4

Algorithm 2 Superposition procedure

```
1: procedure SUPERPOSE(M, M')
2:     M&M' ← new automata
3:     M&M'.start = (q_0, q'_0)
4:
5:     Q ← empty queue
6:     enqueue(Q, (q_0, q'_0))
7:     while Q is not empty do
8:         (q_current, q'_current) ← dequeue(Q)
9:
10:        if q_current in M.final and q'_current in M'.final then
11:            M&M'.final.add((q_current, q'_current))
12:        end if
13:
14:        next ← EXPAND(M, M', M&M', (q_current, q'_current))
15:        enqueueMany(Q, next)
16:    end while
17:    return M&M'
18: end procedure
```

Figure 3.8: Pseudo code for the superposition procedure. Automata M and M' are assumed to have no loops.

Simulating timelines with automata

Suppose we have an automata M which is the result of superposition of some finite state machines that represent events. We would want to sample the strings that this automata generates, since each string represents a certain version of the timeline. We can do so by simulating the finite state machine starting from its initial state q_0 and transitioning all the way to one of its final states. The algorithm for this procedure is given in Figure 3.9. The unknown part here is the transition selection procedure. Since transitions in the automata are probabilistic, we need to introduce a way to select these transitions according to the probabilities assigned to them.

In order to achieve this, we are going to assign each transition an index from 0 to $K - 1$ where K is the total number of transitions from the state under consideration. After that, we are going to compute a prefix sum based on the transition probabilities. The next step is uniformly generating a random real number from 0 to 1. Let us denote this number as x. We are going to attempt to find an index i in the prefix sum such that $prefixSum[i + 1] \geq x > prefixSum[i]$. If such index exists we are going to proceed with transition corresponding to i. Otherwise, we are going to stall in the current state. Stalling is only possible if probabilities across the state transitions do not add up to 1 i.e. not normalized case. Figure 3.10 gives the pseudo code for the algorithm described above.

Algorithm 3 Simulation of automata

```
1: procedure SIMULATE(M)
2:     path ← [ ]
3:     currentState ← q_0
4:     while currentState is not in M.final do
5:         (nextState, α) ← TRANSITION(M, currentState)
6:         path.append(α)
7:         currentState ← nextState
8:     end while
9:     return path
10: end procedure
```

Figure 3.9: Pseudocode for simulating a finite state machine.

Algorithm 4 Transition Selection Algorithm

```
1: procedure TRANSITION(M, currentState)
2:     transitions ← M.T[currentState]
3:     prefixSum[0] ← 0
4:     for i ← 0 to |transitions| − 1 do
5:         prefixSum[i + 1] ← prefixSum[i] + transitions[i].probability
6:     end for
7:     x ← random number between 0 and 1
8:     i ← 0
9:     while i < |transitions| and prefixSum[i + 1] < x do
10:        i ← i + 1
11:    end while
12:    if i < |transitions| then
13:        return transitions[i]
14:    else
15:        return stall
16:    end if
17: end procedure
```

Figure 3.10: Transition Selection Algorithm

Visualizing automata & User interface (UI) design

We require a visualization technique in order to present the finite state machines to the user. Our initial approach involved generating a static image based on the representation of the automata. Generating data for such visualization is quite simple because we store all states by their labels. Hence, we needed to iterate over all states to extract their labels and the labels of states that they are connected to. This information could then be passed to the visualization framework of choice.

However, it quickly became apparent that superposition produces finite state machines with a lot of states and transitions. Therefore, it was hard to visualize the entirety of such automata in a single image. As a result of this, we decided to implement a user interface which would allow the user to look at the automata in an interactive manner. Additionally, we wanted the user interface to provide ways to define automata and to perform predefined experiments of interest. More details about the user interface design and implementation will be provided in Chapter 4, since the design and implementation of UIs are tightly coupled.

3.3 Summary

In this chapter, we provided a solution for the first research direction (RD_1) by extending the notion of superposition to the domain of finite state machines. After that we discussed the probabilities of during and overlap in the context of the finite state machine representing Allen's relations. We considered various schemes for assigning probabilities and showed that no matter the scheme the probabilities of overlap and during are going to be the same. This led us to theorise two alternative approaches based on the operation of automata superposition. One of them proposed introducing the notion of clock and the other suggested making the events more granular by making the models of initial events more complex. We realized that evaluating these approaches would require a software system that can superpose and simulate arbitrary automata. At the end of this chapter, the key algorithms and data structures for this software system were discussed in detail. In the next chapter, we are going to look at the implementation of the aforementioned software that will help us contribute to RD_2 and RD_3 in Chapter 5.

Chapter 4

Implementation

In this chapter, we are going to discuss the implementation of the software system outlined in Chapter 3. Two iterations of the software system will be presented. During the first iteration, we created a proof of concept Python script and made an attempt to extend the Python script to a web app by defining a Flask [1] web server and an HTML frontend based on Jinja [2] templates. It quickly became clear that this approach was not flexible enough and did not provide the interactivity features required. Therefore, we switched to a React [3] app written in Javascript which was deployed as a static web page using a Github Actions [4] pipeline for Github Pages [5].

4.1 Initial approach - Python app

In this section, we are going to present a detailed discussion of the first implementation iteration. We are going to explain how the developed Python app implemented the algorithms we discussed in Chapter 3. Additionally, we are going to present the visualization techniques that the first iteration utilized, including static images and a simple web page. This section will conclude by discussing the limitations of these visualization techniques which served as a motivation for our second implementation iteration. The full code for our initial system is provided as a supplementary material in initial_system.zip.

[1] Flask Documentation (2024)
[2] Jinja Documentation (2024)
[3] React (2024)
[4] GitHub Actions (2024)
[5] GitHub Pages (2024)

4.1.1 Overview of the Solution

The Python script follows the structure we outlined in Chapter 3. It starts by defining a data model which consists of the State class that represents state labels and their transitions and the FSM (Finite State Machine) class for representing automata. The FSM class also defines various operations on automata such as superposition and simulation. Additionally, it has a function which can generate a static image visualization of the automata. Finally, the FSM and State classes are integrated into a proof of concept Flask web application which has a simple input form for defining finite automata and allows the user to superpose automata they defined.

4.1.2 Data model

Listing 4.1 describes the initial implementation of the data model. As discussed in the previous chapter, the State class consists of a label which uniquely identifies it and a transition map implemented using a hash map (in Python this data structure is called dictionary) where key is the symbol and value is the next state. Additionally, we define a separate map to keep track of transition probabilities. It mimics the structure of the transition map except it maps each symbol to the probability of its corresponding transition. The State class allows us to add a transition by specifying the next state, symbol and probability.

The FSM class is initialized with its start state. It has a dictionary which maps state labels to their corresponding State objects. The set of final states and the current alphabet are mainted using Python sets. Adding a state is done by supplying the State object which is then stored in the states dictionary by its label. States can be marked as final, this is done by supplying the state's label which is then added to the set of final state labels. Finally, we can also add transition to the FSM by specifying the previous and next states as well as the corresponding symbol and transition probability. We first add the symbol to the set representing the alphabet and then utilize the State's method for adding transitions by using the state labels supplied and the state map.

```
1  class State:
2      def __init__(self, label):
3          self.label = label
4          self.transitions = {}
5          self.probabilities = {}
6
7      def add_transition(self, other, symbol, probability):
8          self.transitions[symbol] = other
9          self.probabilities[symbol] = probability
10
11
```

```
class FSM:
    def __init__(self, start):
        self.start = start
        self.states = {}
        self.final_states = set()
        self.states[self.start.label] = self.start
        self.alphabet = set()

    def add_state(self, state):
        self.states[state.label] = state

    def mark_state_as_final(self, state_label):
        self.final_states.add(state_label)

    def add_transition(self,
                prev_state_label,
                next_state_label,
                symbol: str,
                probability):
        self.alphabet.add(symbol)
        self.states[prev_state_label].add_transition(
            self.states[next_state_label], symbol, probability)
```

Listing 4.1: This snippet provides the State class and a part of the FSM class responsible for its data model.

4.1.3 Superposition

The code for the initial implementation of superposition algorithm is given in Listing 4.2. The superposition procedure is defined on the FSM class as an instance method and accepts another FSM object denoted as other. It also accepts an optional flag is_clock. This flag ensures that other automata always transitions. This feature is required because we want the clock automata to always proceed forward to keep track of "time". More analysis on this will be provided in Chapter 5.

The procedure starts with defining the start state for the superposed automata. In order to do so, the labels of start states from both automata are combined using concatenation to form the start label for the superposed automata. Additionally, we also save the component labels as part of the state tuple since these will be required when performing the BFS traversal. After that the superposed automata object is instantiated using the start state object.

The next part of the script performs a BFS traversal over the states in the superposed automata. It starts by creating an empty queue and adding the start state to it. We proceed until the queue is empty. On each iteration of the while loop the current state to consider is taken from the queue. Since we recorded the component labels for the

initial FSMs we can now extract them from the tuple. After that we proceed with three cases, namely, current automata transitions and the other does not, the other automata transitions and the current does not, and both automata transition. The first case is only possible if other automata is not a clock.

For the cases when only one of the initial automata transitions we have to compute stall probabilities as $1 - p_{total}$ where p_{total} is the sum of probabilities of all transitions from the current state of the automata that stalls.

Each of the cases starts by defining a new state, which is a tuple consisting of the new State object and the component labels from both automata. The new state object is added to the superposed FSM. The new state is marked as final if both of the component states are final in the original automata. However, there is an exception to this for clock automata. Since clock automata only count the "time" that elapsed, we are not concerned with them reaching their final state. Finally, the new transition is added to the superposed automata and the new state tuple is pushed to the queue for further consideration.

Please note that when both automata transition we define the transition symbol using concatenation of the original symbols. This approach makes it difficult to process the transition symbols in the superposed automata since we need to split the string to see its components. In our second iteration, we eliminated this issue by representing symbols as JSON [6] strings.

```
def superpose(self, other, is_clock = False):
    start_state = (State(self.start.label + "," + other.start.label),
                   self.start.label,
                   other.start.label)

    super_fsm = FSM(start_state[0])

    state_queue = []
    state_queue.append(start_state)

    while len(state_queue) > 0:
        current_state = state_queue.pop(0)

        fsm1_cur_state_label = current_state[1]
        fsm2_cur_state_label = current_state[2]
        super_fsm_label = current_state[0].label

        if not is_clock:
            for symbol, next_state in self.states[
                                    fsm1_cur_state_label].transitions.items():
                new_state = State(next_state.label+","+fsm2_cur_state_label)
                super_fsm.add_state(new_state)

```

[6] JSON Documentation (2024)

```
                if (next_state.label in self.final_states and
                                    fsm2_cur_state_label in other.final_states):
                    super_fsm.mark_state_as_final(new_state.label)

                probability_fsm1 = self.states[
                                     fsm1_cur_state_label].probabilities[symbol]
                probability_fsm2 = 1 - sum(
                    [x[1] for x in other.states[
                                     fsm2_cur_state_label].probabilities.items()])
                probability = probability_fsm1 * probability_fsm2

                super_fsm.add_transition(
                    super_fsm_label, new_state.label, symbol, probability)
                state_queue.append((new_state,
                                    next_state.label,
                                        fsm2_cur_state_label))

        for symbol, next_state in other.states[
                                    fsm2_cur_state_label].transitions.items():
            new_state = State(fsm1_cur_state_label+","+next_state.label)
            super_fsm.add_state(new_state)
            if (is_clock and fsm1_cur_state_label in self.final_states):
                super_fsm.mark_state_as_final(new_state.label)
            elif (next_state.label in other.final_states and
                                            fsm1_cur_state_label in self.final_states)
    :
                super_fsm.mark_state_as_final(new_state.label)

            probability_fsm1 = 1 - sum(
                [x[1] for x in self.states[
                                fsm1_cur_state_label].probabilities.items()])
            probability_fsm2 = other.states[
                                fsm2_cur_state_label].probabilities[symbol]
            probability = probability_fsm1 * probability_fsm2

            super_fsm.add_transition(super_fsm_label,
                                    new_state.label,
                                      symbol,
                                        probability)
            state_queue.append((new_state,
                                fsm1_cur_state_label,
                                        next_state.label))


        for symbol_1, next_state_1 in self.states[
                                fsm1_cur_state_label].transitions.items():
            for symbol_2, next_state_2 in other.states[
                                fsm2_cur_state_label].transitions.items():
                new_state = State(next_state_1.label+","+next_state_2.label)
                super_fsm.add_state(new_state)
                if is_clock and next_state_1.label in self.final_states:
                    super_fsm.mark_state_as_final(new_state.label)
                elif (next_state_1.label in self.final_states
                                    and next_state_2.label in other.final_states):
                    super_fsm.mark_state_as_final(new_state.label)

```

```
                probability_fsm1 = self.states[
                                fsm1_cur_state_label].probabilities[symbol_1]
                probability_fsm2 = other.states[
                                fsm2_cur_state_label].probabilities[symbol_2]
                probability = probability_fsm1 * probability_fsm2

                super_fsm.add_transition(
                    super_fsm_label,
                    new_state.label,
                    symbol_1 + "," + symbol_2,
                    probability)
                state_queue.append((new_state, next_state_1.label, next_state_2.label))

    return super_fsm
```

Listing 4.2: Initial implementation of the algorithm defined in Figure 3.8. The procedure is an instance method in the FSM class.

4.1.4 Simulation

Listing 4.3 provides the initial implementation of the simulation procedure. This implementation did not consider stalling as a possibility which is a limitation that we fixed in the second iteration.

The procedure starts in the current state and continues until we reach one of the final states. The list of symbols encountered is recorded in the path variable. Additionally, invalid flag indicates whether we reached a non-final state with no transitions at any point.

Each iteration starts with checking whether the current state has transitions. After that, a random number between 0 and 1 is generated and the total probability of transitions from the current state is computed. The transition is selected by comparing the ratio between current cumulative probability and the total probability to the random number.

Once the transition was found, the current state is updated and the symbol encountered is recorded in the path variable. At the end of the method, the path array and the invalid flag are returned to the caller.

```
def simulate(self):
  current_state = self.states[self.start.label]
  path = []
  invalid = False
  while current_state.label not in self.final_states:

    if len(current_state.transitions) <= 0:
      invalid = True
      break

```

```
    random_number = random.random()
    sum_of_prob = sum(
                [x[1] for x in current_state.probabilities.items()])

    current_sum_of_prob = 0.0
    i = 0

    probabilities = list(current_state.probabilities.items())

    while current_sum_of_prob / sum_of_prob < random_number:
      current_sum_of_prob += probabilities[i][1]
      i+=1

    decision = probabilities[i-1][0]
    next_state_label = current_state.transitions[decision].label
    current_state = self.states[next_state_label]
    path.append(decision)

  return (path, invalid)
```

Listing 4.3: Initial implementation of the algorithm defined in Figure 3.9. The procedure is an instance method in the FSM class.

4.1.5 Visualization as static image

At the start, we used the Visual Automata [7] Python library. We chose this library because it provided an easy to use interface for visualizing simple automata. In order to visualize the FSM objects with this library, we had to convert our data model to be compatible with the library. Listing 4.4 provides the code that converts the FSM into the library's data model.

The procedure is pretty simple, since the only major modification required is changing the format of how transitions are stored. For each transition we record its state label along with its symbol. The library does not support adding extra labels for transitions, so we create an effective symbol which also contains the transition's probability. The visualize method has an option which controls if we want to see probabilities in the visualization result. We use the NFA (Nondeterminstic finite automata) class from the visualization library because it provides richer visualization options than the DFA (Determistic finite automata) class even though our automata is deterministic in a sense that each transition symbol corresponds to exactly one next state.

The created visual_nfa object could be used to produce a static image representation of the FSM object. Listing 4.5 shows a simple snippet which creates a static PNG image representing the the FSM object.

[7] Visual Automata Python Package (2021)

```
def visualize(self, add_prob = False) -> NFA:
  states = set(self.states.keys())

  transitions = {}
  input_symbols = set()
  for state_label, state in self.states.items():
    transitions[state_label] = {}

    for symbol, next_state in state.transitions.items():
      probability = state.probabilities[symbol]
      symbol_eff = (symbol
          if not add_prob else symbol + f"_{probability}")
      input_symbols.add(symbol_eff)
      transitions[state_label][symbol_eff] = set()
      transitions[state_label][symbol_eff].add(next_state.label)

  initial_state = self.start.label
  final_states = self.final_states

  visual_dfa = NFA(
    states=states,
    input_symbols=input_symbols,
    transitions=transitions,
    initial_state=initial_state,
    final_states=final_states,
  )

  return visual_dfa
```

Listing 4.4: Listing representing the conversion procedure between our data model and the data model of the visualization library. The procedure is an instance method in the FSM class.

```
visual_nfa = fsm.visualize(add_prob=True)
visual_nfa.show_diagram(path="./fsm_static_image.png")
```

Listing 4.5: An example script that shows how visualize method can be used to produce a static image representing the FSM.

Let us look at how the FSM class can be used to create a finite state machine representing the 13 Allen's relations. Listing 4.6 provides the example code. We start by defining finite state machines for two simple events following the unborn, living and dead notation we discussed in the previous chapters. After that, we superpose the automata and visualize the result of superposition as static image. The image produced is given in Figure 4.1.

```
start_1 = State("u_a")
fsm_1 = FSM(start_1)
fsm_1.add_state(State("li_a"))
fsm_1.add_state(State("d_a"))
fsm_1.mark_state_as_final("d_a")
```

```
fsm_1.add_transition("u_a", "li_a", "la", 0.5)
fsm_1.add_transition("li_a", "d_a", "ra", 0.5)

start_2 = State("u_b")
fsm_2 = FSM(start_2)
fsm_2.add_state(State("li_b"))
fsm_2.add_state(State("d_b"))
fsm_2.mark_state_as_final("d_b")
fsm_2.add_transition("u_b", "li_b", "lb", 0.5)
fsm_2.add_transition("li_b", "d_b", "rb", 0.5)

superpose_fsm = fsm_1.superpose(fsm_2)
superpose_fsm.visualize(add_prob=True).show_diagram(path='./allen_fsm.png')
```

Listing 4.6: Example usage of the State and FSM classes to produce an FSM representing Allen's relations.

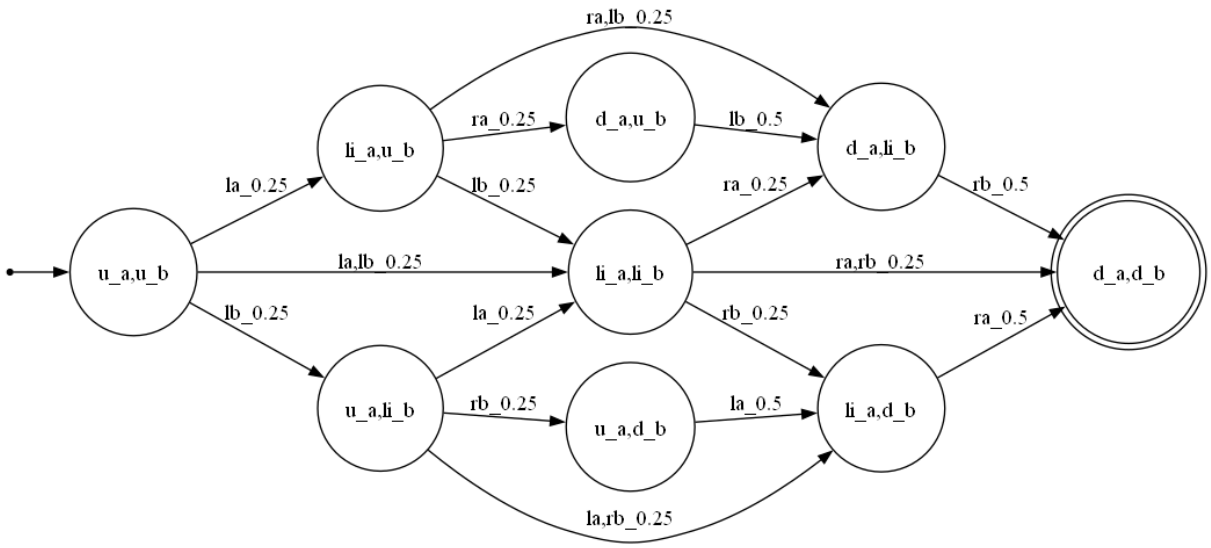


Figure 4.1: Example visualization of the FSM representing Allen's relations produced by code from Listing 4.6.

After we used the system for several experiments, it became clear that the usability of software will benefit from a user interface. We decided to implement a proof of concept user interface using Python Flask web server combined with a frontend based on HTML templates. The next section will discuss its implementation details and present an example visualization.

4.1.6 Simple Python Flask web app

We utilized a Python Flask web server as a backend for our system. The web server was designed using a representational state transfer (REST) [8] paradigm. Table 4.1 lists the

[8]REST APIs - IBM (2024)

endpoints that were defined.

Endpoint name	Request type	Input parameters
Start Page	GET	None
Create FSM	POST	FSM name, Start state label
Edit FSM Page	GET	FSM name
Add transition	POST	FSM name, source, target, probability, symbol
Mark state as final	POST	FSM name, Start name
Superposed FSM Page	GET	FSM 1 name, FSM 2 name

Table 4.1: List of web server endpoints.

For brevity, we are not going to discuss each of the endpoints here, since they are simply wrapping the methods of the FSM class so that they can be used in a web context. Let us look at the superposed FSM page endpoint as an example. Listing 4.7 provides the code for the endpoint. The endpoint begins with extracting arguments representing the FSM names from the request. After that, it tries to find the corresponding FSM objects in a shared dictionary. We use a simple dictionary here as a data storage since the aim of this web app was to create a proof of concept system, which we planned to later improve. The automata are then superposed and the result is stored in the dictionary. The image of the superposed automata is created and is used to render an HTML web page that is returned to the user.

For the web app, we changed the visualization library from Visual Automata to NetworkX [9]. The reason for this is that NetworkX library provides more features such as controlling the node color and the automated positioning of nodes using various graph layout schemes. For brevity, we are not going to discuss the NetworkX implementation here, but instead we will present an example visualization that it produced. Figure 4.2 shows the result of superposing automata from Figure 3.3 using the web application. The initial automata were specified using the input forms provided by the web app. The start state is highlighted with green color, while the final state is presented in red.

```
@app.route('/superpose_fsm', methods=['GET'])
def superpose_fsm():
    fsm_name1 = request.args.get('fsm1')
    fsm_name2 = request.args.get('fsm2')

    fsm1 = finite_state_machines.get(fsm_name1)
    fsm2 = finite_state_machines.get(fsm_name2)

    if fsm1 and fsm2:
        superposed_fsm = fsm1.superpose(fsm2)
        print(superposed_fsm.format_as_string())
```

[9] NetworkX Documentation (2024)

```
        finite_state_machines[fsm_name1 + " x " + fsm_name2] = superposed_fsm

        superposed_fsm_image = superposed_fsm.visualize()
        return render_template('superpose_fsm.html',
                fsm_name1=fsm_name1, fsm_name2=fsm_name2,
                    superposed_fsm_image=superposed_fsm_image)
    else:
        return "One or both FSMs not found."
```

Listing 4.7: Superposed FSM Page endpoint.

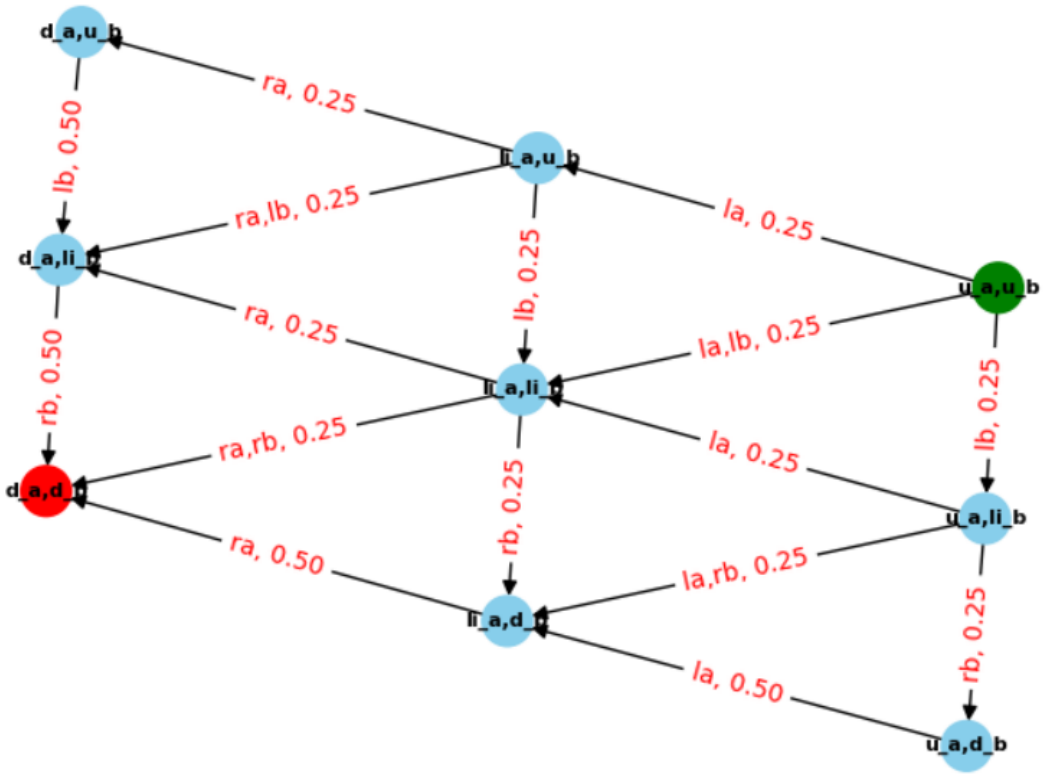


Figure 4.2: Example visualization of the FSM representing Allen's relations produced by the proof of concept web app. The initial FSMs were created using simple input forms.

4.1.7 Analysis of the initial solution

Our initial Python script provided the functionality for superposing and simulating arbitrary automata. It also provided means of visualizing FSMs as static images. However, it was difficult to use because setting up various experiments required writing code to create the initial automata from scratch which is a tedious process. Additionally, the visualization did not work well when the superposed automata had a lot of states, since it became

too cluttered. The proof of concept web app helped us improve the visualization by using a lower level drawing library while also allowing the user to input the initial finite state machines using forms. However, it quickly became apparent that using static template rendering is not a viable approach for building a fully fledged web app because it lacked the modern interactivity features available in component based fronted frameworks such as React. Therefore, we decided to leave the Python web app implementation in the proof of concept state and switched to using React and Javascript. The next section is going to present the second iteration of our system which was developed using the aforementioned technologies and deployed using Github Pages.

4.2 Improved system - React web app

This section is going to give an overview of the final system that we developed. We are going to start with describing the Javascript implementation of algorithms from Chapter 3, emphasizing the key improvements to the superposition and simulation logic. After that, we are going to discuss the design of the user interface, while providing screenshots of user views. Finally, the deployment and testing strategy will be presented. The source code for the improved system is submitted as a supplementary material in a zip file named improved_system.zip.

4.2.1 Overview of the Solution

The implementation has two main parts. The first part defines all of the logic behind superposition and simulation of the automata. The logic is implemented using JavaScript for easy integration with frontend user interface. The second part is a frontend React application, which was built using component based paradigm. The React application provides several pages, including:

- pages for running several predefined experiments
- a page for defining finite state machines
- a page for superposing finite state machines
- a page for simulating finite state machines
- a page for estimating the length of timelines represented by finite state machines

4.2.2 Data model

One of the limitations of the initial data model was the fact that transition symbols for superposed automata were derived using string concatenation. We realized that a more convenient way is to represent the transition symbols using arrays so that we can simply merge the arrays corresponding to symbols from initial automata during superposition.

However, we also need to use symbols as a key to the hash maps storing the transitions and probabilities. Arrays are a reference type which means that they are going to be compared using their memory addresses when stored in a hash map. This can lead to undesired behavior when we have two arrays consisting of the same values but stored at different memory addresses. In order to fix this problem, we decided to extend the standard Javascript Map class to work with reference types. The idea behind this was to maintain data in a regular Map where key is a JSON string representation of the array, while exposing the array based interface to the user. Listing 4.8 provides the implementation.

```
class DeepDict {
  constructor() {
    this.map = new Map();
  }

  set(key, value) {
    const stringifiedKey = JSON.stringify(key);
    this.map.set(stringifiedKey, value);
  }

  get(key) {
    const stringifiedKey = JSON.stringify(key);
    return this.map.get(stringifiedKey);
  }

  has(key) {
    const stringifiedKey = JSON.stringify(key);
    return this.map.has(stringifiedKey);
  }

  delete(key) {
    const stringifiedKey = JSON.stringify(key);
    return this.map.delete(stringifiedKey);
  }

  keys() {
    return [...this.map.keys()].map((key) => JSON.parse(key));
  }

  values() {
    return [...this.map.values()];
  }
```

```
33
34    entries() {
35      return [...this.map.entries()].map(([key, value]) => [
36        JSON.parse(key),
37        value,
38      ]);
39    }
40
41    clear() {
42      this.map.clear();
43    }
44
45    size() {
46      return this.map.size;
47    }
48  }
49
50  module.exports = DeepDict;
```

Listing 4.8: Deep dictionary implementation based on the standard Map class. JSON representation of arrays is used as the key in standard Map class, while the deep dictionary class exposes the interface based on arrays to the user.

Additionally, we created a similar DeepSet class which is implemented as a DeepDict where keys and values are the same. We are going to omit its description for brevity.

The classes representing states and finite state machines were similar to our initial implementation. Listing 4.9 provides the JavaScript implementation for the State class. Additional error handling was introduced and an option to normalize probabilities i.e. eliminate the chance of stalling was added.

```
1  const DeepDict = require("./dict.js");
2
3  class State {
4    constructor(label) {
5      this.label = label;
6      this.transitions = new DeepDict();
7      this.probabilities = new DeepDict();
8    }
9
10   normalizeProbabilities() {
11     let sum = 0;
12     // eslint-disable-next-line no-unused-vars
13     for (let [_, probability] of this.probabilities.entries()) {
14       sum += probability;
15     }
16
17     if (sum > 0) {
18       for (let [symbol, probability] of this.probabilities.entries()) {
19         this.probabilities.set(symbol, probability / sum);
20       }
```

```
21      }
22    }
23
24    addTransition(symbol, nextState, probability) {
25      if (this.transitions.has(symbol)) {
26        return;
27      }
28
29      if (probability < 0 || probability > 1) {
30        throw new Error(
31          `Probability has to be between 0.0 and 1.0, supplied: ${probability}!`
32        );
33      }
34
35      this.transitions.set(symbol, nextState);
36      this.probabilities.set(symbol, probability);
37    }
38  }
39
40  module.exports = State;
```

Listing 4.9: JavaScript class for representing states of finite automata. A function for normalizing probabilities was added compared to the initial implementation. Additional error handling was also introduced.

The data model for the FSM class is very similar to the original one we proposed so we are not going to discuss it here. Please check fsm.js file from the supplementaty material (improved_system.zip) for details.

4.2.3 Superposition

With the introduction of DeepDict and DeepSet, we could implement the superposition procedure in a more elegant way. The key difference here is that there is no need to maintain the initial labels separately, since the labels in a superposed automata are now defined as two element arrays as opposed to strings in the initial implementation. Each element in the label array represents the corresponding state label from the initial automata. Additionally, we refactored the code so that different transition possiblities are handled in separate functions. Listing 4.10 provides the updated superposition procedure implemented using JavaScript.

Moreover, we removed the notion of clock from the superposition procedure. The reason for this was that introducing clocks did not help us solve the problem with probabilities of overlap and during. The experiments with clocks were done using our initial system and therefore there was no reason to include the clock functionality into the final software. More discussion on this will be given in Chapter 5.

```
superpose(otherFsm) {
    const superposedFsm = new Fsm();
    const thisStartState = this.statesByLabel.get(
      this.startStateLabels.values()[0]
    );
    const otherStartState = otherFsm.statesByLabel.get(
      otherFsm.startStateLabels.values()[0]
    );

    const superFsmStartStateLabel = [
      thisStartState.label,
      otherStartState.label,
    ];

    const superFsmStartState = new State(superFsmStartStateLabel);
    superposedFsm.addState(superFsmStartState);
    superposedFsm.markAsStart(superFsmStartStateLabel);

    const stateLabelQueue = [superFsmStartStateLabel];

    while (stateLabelQueue.length > 0) {
      const currentStateLabel = stateLabelQueue.pop();

      const thisFsmNewLabels = this.generateThisFsmTransitions(
        otherFsm,
        superposedFsm,
        currentStateLabel
      );

      stateLabelQueue.push(...thisFsmNewLabels);

      const otherFsmNewLabels = this.generateOtherFsmTransitions(
        otherFsm,
        superposedFsm,
        currentStateLabel
      );

      stateLabelQueue.push(...otherFsmNewLabels);

      const bothFsmsTransitionNewLabels = this.generateBothFsmsTransitions(
        otherFsm,
        superposedFsm,
        currentStateLabel
      );

      stateLabelQueue.push(...bothFsmsTransitionNewLabels);
    }

    return superposedFsm;
}
```

Listing 4.10: Improved superposition procedure. Functions generateThisFsmTransitions, generateOtherFsmTransitions and generateBothFsmsTransitions consider the cases when only the current automata transitions, only the other automata transitions and both automata transition respectively.

4.2.4 Simulation

As mentioned in the previous subsection, the simulation implementation did not take into account the possibility of stalling. In the improved system, we eliminated this problem by comparing cumulative probability directly with the generated random value. Listing 4.11 presents the updated simulation procedure.

```
simulate() {
    let currentStateLabel = this.startStateLabels.values()[0];

    const symbols = [];
    const states = [];

    while (true) {
      const currentState = this.statesByLabel.get(currentStateLabel);

      if (this.finalStateLabels.has(currentStateLabel)) {
        return { symbols: symbols, states: states };
      }

      const transitionProbabilities = currentState.probabilities;

      let cumulativeProbability = 0;
      const randomValue = Math.random();

      for (const [symbol, probability] of transitionProbabilities.entries()) {
        cumulativeProbability += probability;

        if (randomValue < cumulativeProbability) {
          states.push(currentStateLabel);
          symbols.push(symbol);
          currentStateLabel = currentState.transitions.get(symbol).label;
          break;
        }
      }
    }
}
```

Listing 4.11: Improved simulation procedure. The cumulative probability is directly compared with the random value allowing for the automata to stall if the sum of probabilities of transitions in the current state is less than 1.

4.2.5 Visualization

The main issue with previous visualization appproaches was the lack of interactivity. The superposed automata can contain a lot of states and transitions, which makes the global automata picture impractical. Therefore, the key requirements for the updated version of the visualization are:

- ability to arbitrarily zoom in / out on the automata
- ability to move through automata state by state, while only seeing the previous and next states
- ability to move the automata states around the canvas in case the automated layout is not easily readable

With these requirements in mind, we considered several JavaScript visualization libraries which integrate with React, including Vis.js, ReactFlow, and react-d3-graph. While Vis.js and react-d3-graph offer extensive features for creating interactive network graphs within React applications, such as various layouts and customizable styling, they may pose a challenge due to their complex APIs and usage patterns. In contrast, ReactFlow provides a streamlined approach tailored specifically for building interactive flow diagrams within React. Leveraging React's component-based architecture, ReactFlow ensures seamless integration and efficient development, offering a more intuitive and consistent experience for React developers. Thus, considering its simplicity, compatibility, and efficiency, ReactFlow turns out to be the most suitable choice for our use case.

ReactFlow provides a component which accepts a list of nodes and a list of edges connecting the nodes. Each node specifies attributes such as position, style (color, type of arrows for edges) and id. Each edge specifies the target and sources nodes along with a string label, an edge id and several style attributes. This component provides zoom in / out functionality out of the box as well as the ability to move nodes around the canvas as necessary.

Now that the layout i.e. node positions are not generated automatically we had to come up with a way to position the nodes on the canvas. We considered two algorithms for this.

The first algorithm used a topological sort procedure to find the topological order of the states in the finite state machine. A topological order in a graph is an ordering of the nodes where every node appears only after all the nodes pointing to it have appeared. We utilized the property that topological order of the graph is a reverse of its post-order

traversal using a Depth-First Search (DFS) algorithm. Once we found the topological ordering of the states in the finite automata, we positioned all states in vertical layers of size K, where K is a predefined parameter. The vertical layers were spaced out from one another using a predefined horizontal step. For brevity, we are going to omit our topological sort implementation here since it is a well-known algorithm. However, we are going to present our positioning procedure in Listing 4.12. Figure 4.3 provides an example positioning generated by this algorithm for the automata representing Allen's relations.

```
generateNodesAndEdgesForReactFlowTopologicalSort(
    startNodeX,
    startNodeY,
    stepX,
    stepY,
    nodesPerVerticalLayer = 3
  ) {
    const [nodes, edges] = this.generateConnectionsForReactFlow();

    const topologicalOrder = this.topologicalSort().map(
      (x) => `${JSON.stringify(x)}`
    );

    for (const node of nodes) {
      const order = topologicalOrder.indexOf(node.id);
      const shiftX = Math.floor(order / nodesPerVerticalLayer);
      const shiftY = order % nodesPerVerticalLayer;
      node.position = {
        x: startNodeX + stepX * shiftX,
        y: startNodeY + stepY * shiftY,
      };
    }

    return {
      nodes: nodes,
      edges: edges,
    };
  }
```

Listing 4.12: Node positioning using topological sort. The generateConnectionsForReactFlow method produces nodes and edges based on the states and transitions in the finite state machine. After that, the topological ordering algorithm is run. For each of the nodes the position is computed by evaluating which layer it belongs to based on its index in the topological ordering. The position within the layer is identified using a corresponding modulo operation.

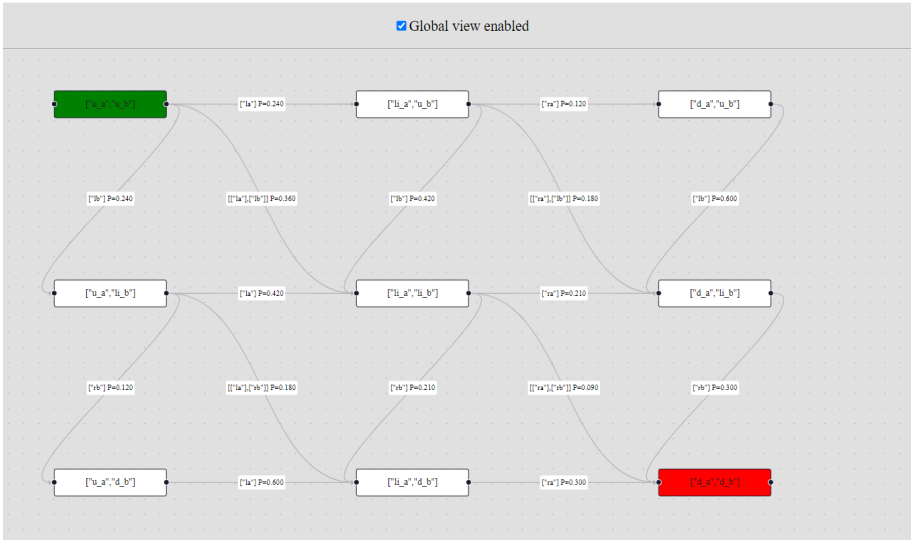


Figure 4.3: Visualization of Allen's relations FSM generated using a topological sort positioning algorithm with 3 nodes per vertical layer.

The second algorithm we considered was based on the length of longest path from the start state to a given state. In this positioning algorithm, nodes are also positioned into vertical layers, however, the vertical layer is now determined by the length of the longest path from the start state to the state in question. The intuition behind this algorithm is that every transition in the automata represents some change in the events and hence grouping the states by the maximum number of changes that occurred before them will allow us to reduce clutter while maintaining the chronological order.

The algorithm for finding longest paths works by performing a topological sort and iterating through the states in a topological order. It initializes a dictionary to store the distances from each state label to the starting state. The distance for starting state is set to zero. For each state in topological order, the algorithm iterates through its transitions, updating the distance for each adjacent state if a longer path is found. The code for the algorithm is given in Listing 4.13. The positioning procedure is similar to that of the previous algorithm we considered, except the index of the vertical layer is determined by the length of the longest path. Hence, we are going to omit its discussion. Figure 4.4 shows the visualization generated by this position algorithm for the finite automata representing Allen's relations.

The key difference between the two positioning algorithms considered is that the topological sort algorithm requires a predefined number of nodes per vertical layer to be supplied, while the longest path algorithm does not have this limitation. After experimenting with several automata we concluded that the longest path algorithm is a more generic approach since it produces decent visualizations regardless of the finite automata supplied without

the need of manually configuring number of nodes per vertical layer.

```
findLongestPaths() {
    const topologicalSort = this.topologicalSort();
    const distances = new DeepDict();

    for (const stateLabel of this.statesByLabel.keys()) {
      distances.set(`${JSON.stringify(stateLabel)}`, 0);
    }

    for (const stateLabel of topologicalSort) {
      // eslint-disable-next-line no-unused-vars
      for (const [_, nextState] of this.statesByLabel
        .get(stateLabel)
        .transitions.entries()) {
        const newDistance = distances.get(`${JSON.stringify(stateLabel)}`) + 1;
        if (newDistance > distances.get(`${JSON.stringify(nextState.label)}`)) {
          distances.set(`${JSON.stringify(nextState.label)}`, newDistance);
        }
      }
    }

    return distances;
}
```

Listing 4.13: Algorithm for finding the longest path length from the starting state to all states in the FSM. The first step is a topological sorting. After that, the longest paths are computed in bottom up manner by traversing the nodes in topological order.

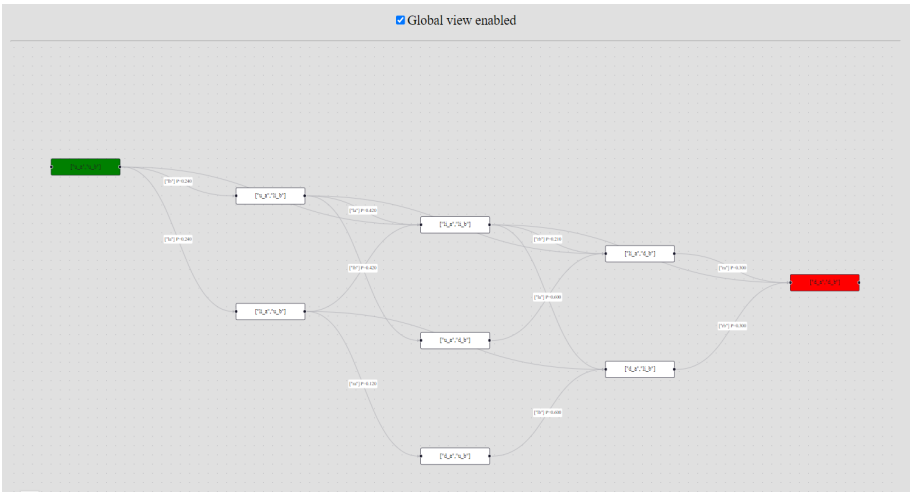


Figure 4.4: Visualization of Allen's relations FSM generated using a longest paths positioning algorithm.

With the positioning algorithm developed, the final part was to allow the user to step through individual states in the automata. For this feature, we had to find all previous and next states for a given state in the automata and then restrict the visualization to

only show these states. Additionally, we had to develop a user interface component which provides the user with options to move to one of the next or previous states. All of the next states were already available as part of the transition mapping, while finding all previous states involved a simple traversal over the transition maps of all states in the automata. Figure 4.5 provides an example of this feature for the finite automata representing Allen's relations.

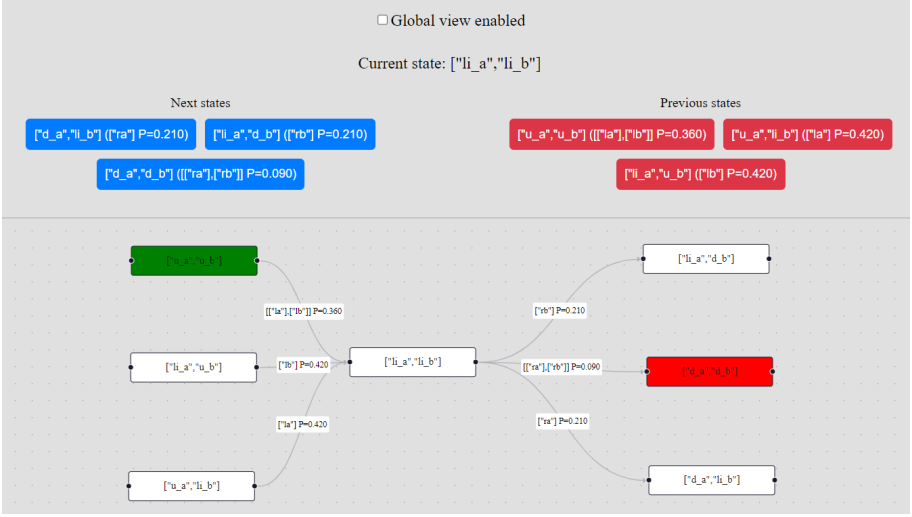


Figure 4.5: Example of step by step visualization for the Allen's relations automata. The current state is (li_a, li_b).

4.2.6 User interface and features

This subsection is going to discuss the various pages that are available to the user. It is going to explain the purpose of each page and how it operates. The pages were built using a component based React paradigm. This means that pages are comprised of smaller components which may be re-used across multiple views. We are not going to go into the details of the components implementation here since the frontend JavaScript code is quite verbose. The full user interface implementation is available as part of the improved_system.zip supplementary material.

Allen's relations predefined experiment

This page allows the user to perform experiments on the automata from Figure 3.4. It allows the user to specify the living and dying probabilities for the initial automata representing simple events (lifespans of creatures) in order to generate various transition probabilities for the superposed automata representing Allen's relations. It also provides the options to normalize all probabilities and to override probabilities of individual tran-

sitions as needed. Finally, it lets the user run simulations on the automata by specifying the number of runs. The simulation results are specified as a table listing the probabilities of each of Allen's relations. The user has an option to look at the results of individual runs if necessary by ticking a checkbox.

Another option presented to the user is generating a heat map of probabilities of a given relation based on the living and dying probability. Such heat maps were first presented in Suliman (2021). The user has the option to specify the number of iterations and probability step for the heatmap which control how accurate it is. Figures 4.6 and 4.7 provide the screenshots of the main view of the page and the heat map view respectively.

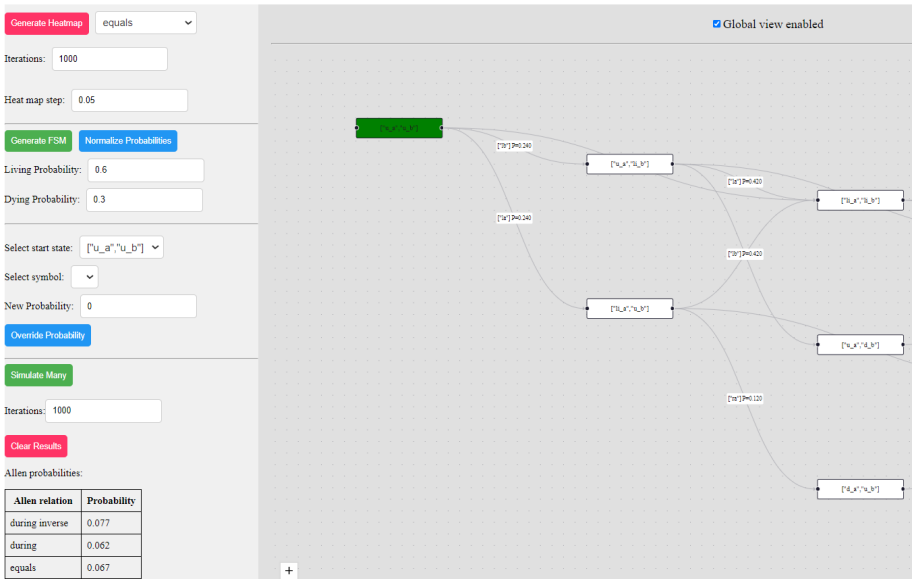


Figure 4.6: Page for predefined experiments on Allen's relations probabilities.

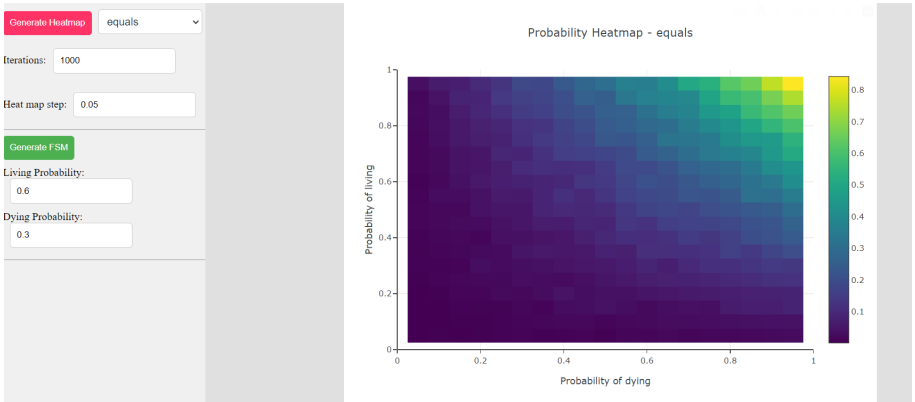


Figure 4.7: Heat map generated for the probability of equals relation.

Allen's relations based on granular initial automata experiment

This page is very similar to the previous one we discussed, except it allows the user to perform experiments on the superposed automata that is based on initial automata of higher granularity. By granularity, we mean that the initial automata contains multiple living states. Figure 4.8 provides a visualization of a granular automata where the living state is split between being young and old. The superposed automata in this case has 16 states, because the initial automata have 4 states each. In order to generate heatmaps, it is necessary to fix 2 out of the 4 parameters (α, p_{start}, p' and p). Figures 4.9 and 4.10 provide a general view and the heat map view respectively.

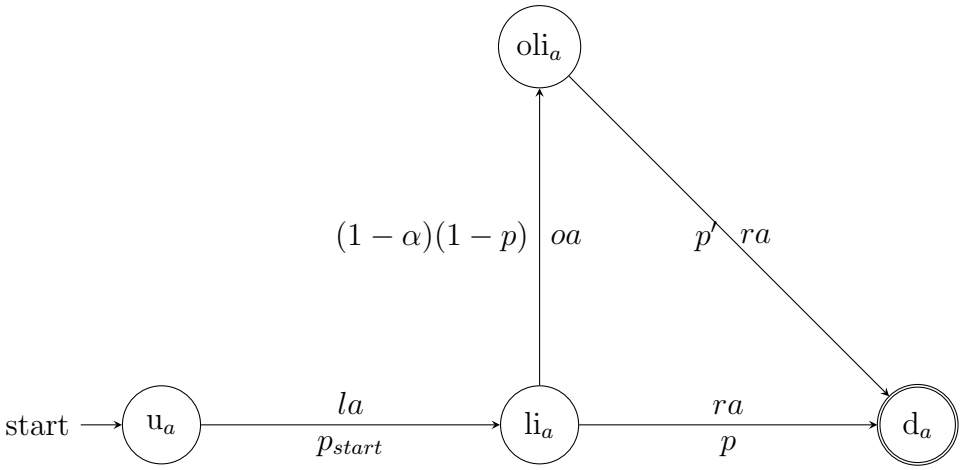


Figure 4.8: Automata representing the lifespan of a creature with multiple living states i.e. young living li_a and old living oli_a. Probability of becoming alive is p_{start}. Probability of dying while being young is p. Prior probability of becoming old is α, hence the probability of becoming old is $(1-\alpha)(1-p)$ i.e prior probability of not dying times the probability of becoming old. Probability of dying while being old is p'.

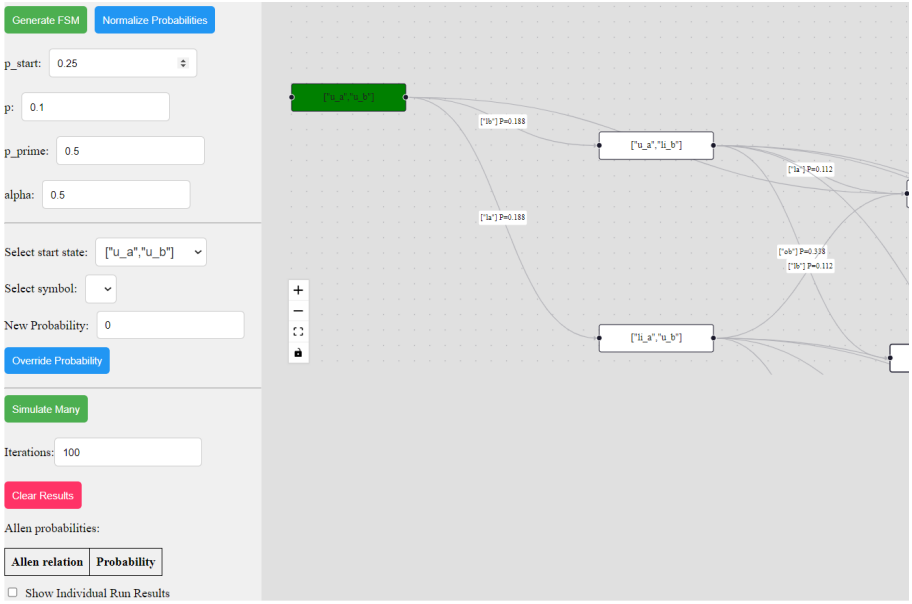


Figure 4.9: Page for predefined experiments on Allen's relations probabilities for granular initial automata.

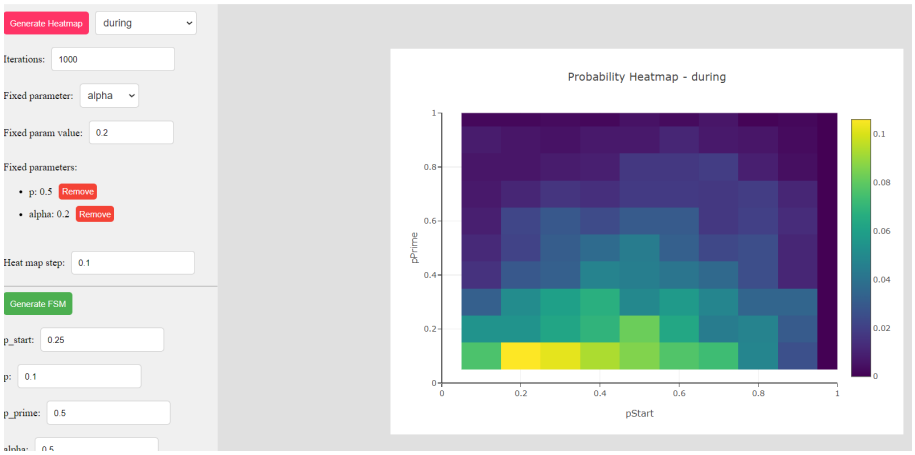


Figure 4.10: Heat map generated for the probability of during relation for granular initial automata.

Page for defining finite state machines

This page allows the user to input finite state machine using a form while examining its visualization. After defining the automata, the user can download the automata representation in JSON format in order to save it and use in the other pages. Figure 4.11 provides a screenshot of this page.

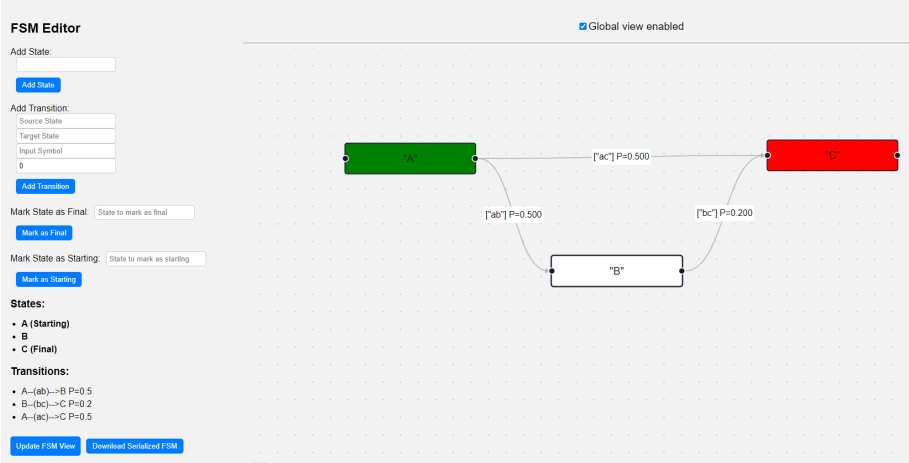


Figure 4.11: FSM input page.

Page for superposing finite state machines

This page allows the user to superpose arbitrary finite state machines by uploading their JSON configurations into the form (Figure 4.13). After the superposition, the user can view the result and download it in JSON format (Figure 4.12).

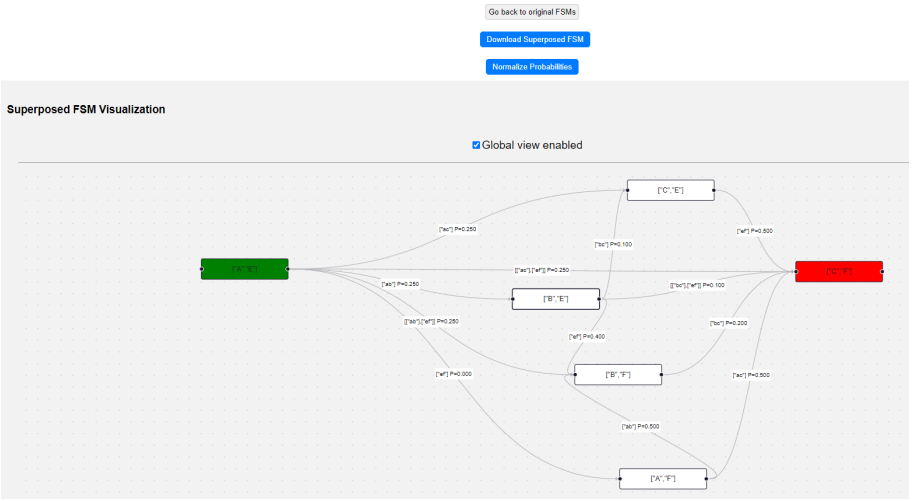


Figure 4.12: Superposition result view.

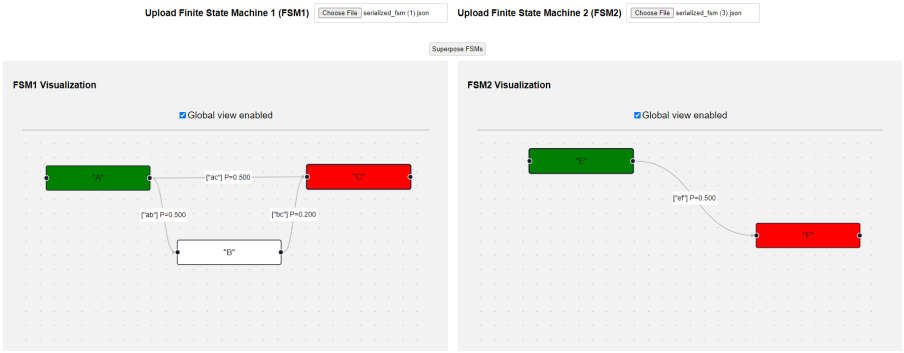


Figure 4.13: FSM superpose input view.

Page for simulating finite state machines

This page allows the user to upload a JSON representation of an arbitrary automata and run simulations for it. The strings generated by the automata are presented in a histogram. Figure 4.14 shows the simulate view.

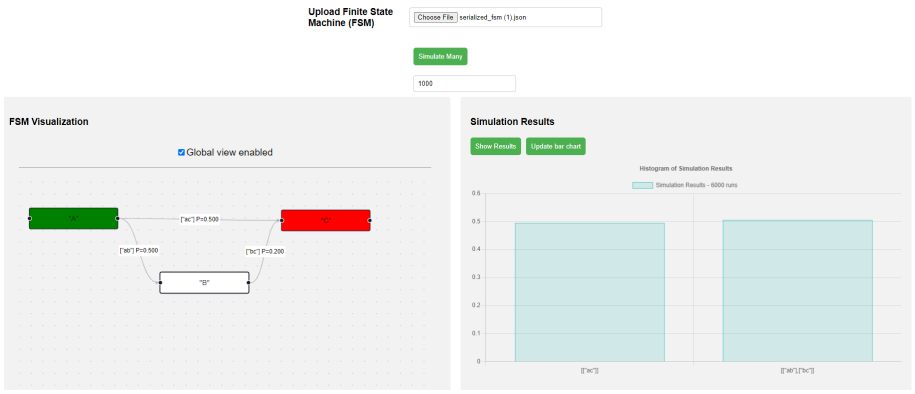


Figure 4.14: FSM simulate view. Please note the probabilities of both strings are the same because 0.5 is the same as $0.5 \times \frac{1}{1-0.8} \times 0.2 = 0.5$. Formula $0.5 \times \frac{1}{1-0.8} \times 0.2$ arises from the fact that we can stall in state B with probability of 0.8.

Page for estimating the number of transitions to end up in the final state

This page was not a part of the original design. We decided to introduce it in order to see how many transitions are required for the majority of density in state distribution to end up in the final state. This computation is done by creating a transition probability matrix based on the finite state machine. After that, we repeatedly multiply the vector representing the initial state distribution with the transition matrix. The initial state distribution is a vector of probabilities where the element corresponding to the starting state is 1 and the rest of the elements are 0. In other words, we are executing a power

method for finding the final distribution of the Markov Chain corresponding to the finite automata given an initial distribution. Figure 4.15 provides the screenshot of the view.

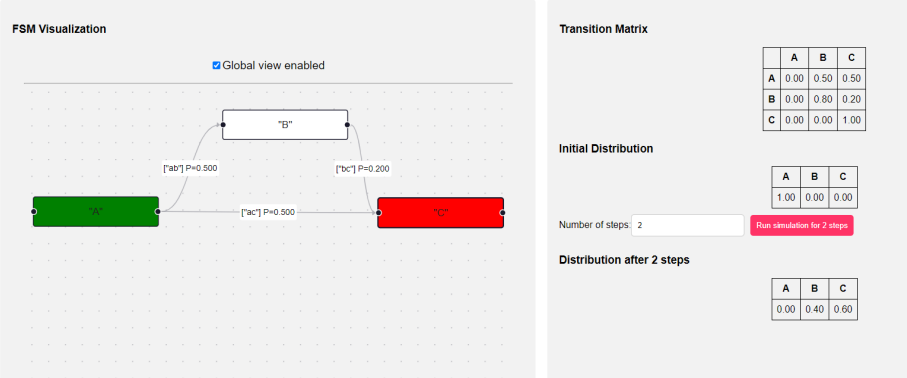


Figure 4.15: FSM matrix view, which allows the user to observe the expected state distribution after a given number of steps, assuming that we began in the starting state.

4.2.7 Testing & Deployment

The testing strategy for the app involved defining 45 automated unit tests using Jest framework[10]. The automated unit tests helped us ensure that the code for the logic behaved as expected.

We deployed the application using Github Actions pipeline for Github Pages. Github pages is a platform that allows hosting static web pages for free. In order to deploy our application to Github Pages, we had to package it as a static website by following the instructions in Nelson Michael (2024). Our app is available at `https://cppavel.github.io/fsm-website/`.

4.3 Summary

In this chapter, we presented a detailed discussion of the two iterations of our software system. The chapter began with a description of the initial python script we developed, including the superposition and simulation algorithms as well as the visualization techniques. After that, we presented the proof of concept Python Flask web app. We outlined the reasons why a Python web app based on static pages was not the best approach for

[10] Jest - JavaScript Testing Framework (2024)

our problem and proposed creating a JavaScript React web application as a second iteration. The key improvements that were introduced in the second iteration of the software system were then presented, including the enhanced versions of the algorithms and new techniques for visualization. We concluded the chapter with a thorough analysis of the user interface and its features followed by a short section about testing the code and deploying the software to Github Pages.

With the software system ready, we can finally proceed to the evaluation stage. In the next chapter, we will use our software to run several experiments which will contribute to research directions 2 and 3 (RD_2 and RD_3).

Chapter 5

Evaluation

This chapter will begin with a sanity check experiment which will compare the Allen's relations probabilities produced by our software based on the automata from Figure 3.4 to the analytical formulas we derived in Chapter 3. As an additional sanity check, we will compare our results to the work of Suliman (2021), since they also performed simulations on automata from Figure 3.4. After that we are going to perform two novel experiments. The first experiment will involve superposing the Allen's relations automata with a clock automata and analyzing the resulting probabilities. This experiment is conducted using the initial version of the software, since it was our first attempt at resolving the problem with probabilities of overlap and during. The second experiment will look at the probabilities of Allen's relations generated by the superposition of initial automata with increased granularity (Figure 4.8). This experiment is done using the web app that we developed.

5.1 Sanity checks

This section discusses the sanity checks that we perform in order to ensure that our models are correct. The first sanity check is based on the analytical formulas we derived earlier, while the second one compares our results to existing literature.

5.1.1 Analytical formula sanity check

From Chapter 3, we know that probabilities of overlap and during for the automata in Figure 3.4 can be computed using the below formulas. We assume that events are of the same nature and hence $p_a = p_b$ and $q_a = q_b$, which yields $P_{overlap} = P_{duration}$.

$$P_{overlap} = p_a p_b q_a (1-p_b)(1-q_a)(1-q_b) \frac{1}{1-(1-p_a)(1-p_b)} \frac{1}{1-(1-q_a)(1-p_b)} \frac{1}{1-(1-q_a)(1-q_b)}$$

$$P_{duration} = p_a p_b q_b (1-p_b)(1-q_a)^2 \frac{1}{1-(1-p_a)(1-p_b)} \frac{1}{1-(1-q_a)(1-p_b)} \frac{1}{1-(1-q_a)(1-q_b)}$$

Therefore, we can simplify the above expressions to

$$P_{duration} = P_{overlap} = p^2 q (1-p)(1-q)^2 \frac{1}{1-(1-p)^2} \frac{1}{1-(1-q)(1-p)} \frac{1}{1-(1-q)^2}$$

Table 5.1 summarizes the results for various configurations of p and q. We can see that our software is very close to the analytical formula indicating its correctness.

p	**q**	**Software simulation (100000 iterations)**	**Analytical formula**
0.5	0.5	0.037	0.037037
0.9	0.1	0.038	0.0383301
0.3	0.9	0.001	0.00120752
0.1	0.1	0.106	0.106284

Table 5.1: Sanity check comparing the probabilities of duration and overlap produced by our software (`https://cppavel.github.io/fsm-website/fsmexample-allen`) to results based on the above analytical formula.

5.1.2 Sanity check based on previous work

An experiment consisting of 500000 samples is performed in Suliman (2021) for $p = q = 0.5$. We run the same experiment and compare the results in Table 5.2. We can see that our results closely align with Suliman (2021).

Heat maps were also introduced in Suliman (2021) to look at the surfaces describing how probabilities of relations change depending on p and q. Figure 5.1 presents the

Relation name	Suliman (2021)	Our result (500000 iterations)
Equals	0.110	0.111
Starts	0.111	0.111
Starts inverse	0.111	0.111
Meets	0.111	0.110
Before	0.112	0.111
Finishes inverse	0.037	0.037
Overlap	0.037	0.037
During inverse	0.037	0.037
Meets inverse	0.111	0.112
Finishes	0.038	0.037
During	0.037	0.037
Overlap inverse	0.037	0.037
Before inverse	0.111	0.111

Table 5.2: Comparison of results obtained in Suliman (2021) for $p = q = 0.5$ with 500000 samples and the results produced by our software (`https://cppavel.github.io/fsm-website/fsmexample-allen`).

results obtained in Suliman (2021). We can see that the probability of equals relation gradually increases as p and q increase. Figure 5.2 presents the heat map for equals relation generated using our software, we can see that it follows the same pattern as the heat map presented in Suliman (2021).

5.1.3 Overview of sanity checks

We can conclude that our software passed all of the sanity checks considered. It matched the probabilities for during and overlap computed using the analytical formula. Additionally, it was able to reproduce the results obtained in Suliman (2021).

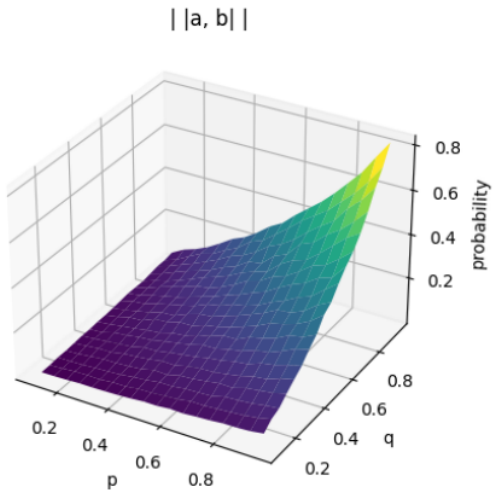


Figure 5.1: Figure 8 from Suliman (2021). 3D heat map representing the probability of equals relation depending on p and q.

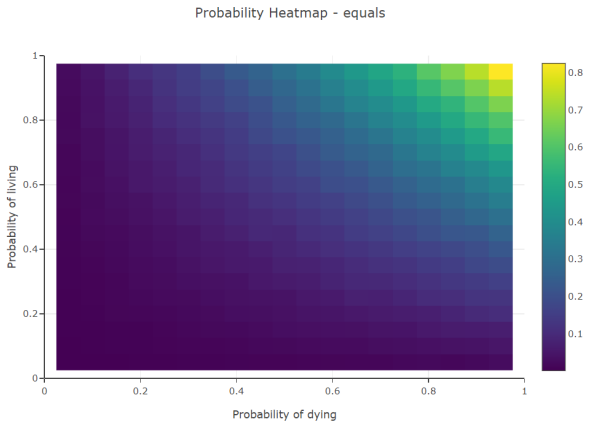


Figure 5.2: 2D Heat map for equals relation generated using `https://cppavel.github.io/fsm-website/fsmexample-allen`. We can see that it is very similar to the results of Suliman (2021).

5.2 Superposing Allen's relations automata with a clock automata

In this section, we are going to look at our first attempt at fixing the issue with probabilities of duration and overlap. Namely, we wanted to create a model which suggests that overlap is more probable than duration for events of the same nature (same probabilities of starting and finishing). Please note we are going to use the initial version of our software (Python script) to perform this simulation. The reason for this is that we

performed this experiment at the start of our research before the improved software was developed. The experiment showed that this approach did not help us solve the problem and therefore the functionality for this simulation was not added to the final version of the software system. The Python script utilized is available as part of the initial_system.zip supplementary material.

We proposed that superposing the automata from Figure 3.4 with a clock FSM would help us resolve the problem with probabilities of duration and overlap, since each state would be augmented with the time that passed so far. Clock FSM is a finite automata with N states with each state representing a tick of the clock. The starting state is labelled as zero and the finishing state is labelled as $N - 1$. The transitions happen between adjacent states and have probability of 1. Figure 5.3 provides an example of clock FSM for $N = 5$.

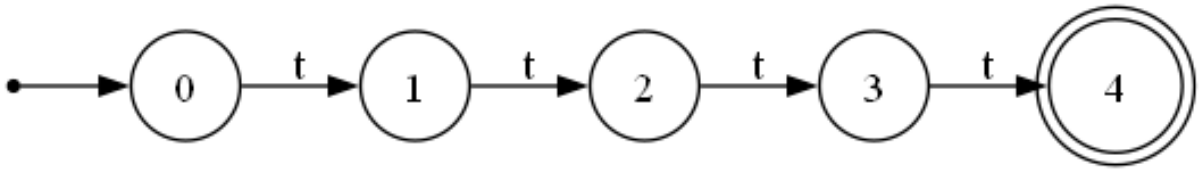


Figure 5.3: Clock FSM for $N = 5$

Superposing the automata from Figure 3.4 with a clock automata produces a lot of states. Therefore, the automata becomes difficult to visualize, especially using the simple visualization techniques that the initial version of the software utilized. Nevertheless, we present the superposed automata for $N = 7$ in Figure 5.4. From this diagram, we can notice that some states have no transitions out of them but are not final. Such states exist because the clock automata has a finite length and we require the clock transitions during the superposition. Additionally, we can see that there are multiple final states. Some of the final states have outgoing transitions to other final states. These transitions simply increase the clock by 1 while staying in the (d_a, d_b) state for the automata from Figure 3.4. During simulations, we exit immediately after reaching the first final state, so this does not impact our results.

Table 5.3 provides results for various values of clock length N. We run 500000 simulations with $p = q = 0.5$. The assignment of p and q is not essential, since we want our model to yield that overlap is more probable than during regardless of the initial probability assignments.

We can notice that for all values of N the number of observed during and overlap relations are relatively the same. This suggests that introducing the clock does not help resolve

our problem. The simple intuition here is that instead of having a single (li_a, li_b) state we now have a collection of states (li_a, li_b, t) where t indicates the current time. However, the problem is still the same. Namely, once we reach a state in which both a and b are alive, we have no way of knowing who was born first leading to the probability of overlap being the same as the probability of during.

The only impact of the clock is that it limits the number of transitions that the original FSM is allowed to make. We can notice this by comparing the counts for overlap and during to the total number of simulations. When N is small the counts are small because there is a lower chance that the original automata reaches one of the final states compared to the situation when it is allowed to make more steps. As N increases the ratio between counts of during/overlap and the total number of simulations approaches 0.037. For example, $\frac{18828}{500000} = 0.037656$. This is expected, since when we allow the automata to transition for a large number of steps, it will start approaching the original case when we did not have any bound on the number of steps at all.

Overall, we can see that introducing the notion of clock does not help us solve the problem of overlap and during having the same probability.

Total simulations	**N (clock length)**	**Count during**	**Count overlap**	**p**	**q**
500000	5	3862	3949	0.5	0.5
500000	10	17565	17549	0.5	0.5
500000	15	18439	18594	0.5	0.5
500000	25	18828	18411	0.5	0.5

Table 5.3: Counts of overlap and during for simulations of clocked automata. The initial automata representing Allen's relations is assumed to have $p = q = 0.5$. The choice of p and q can be arbitrary since our goal is to show that overlap is more probable than during for all initial configurations.

5.3 Superposing granular automata

The next attempt we made at making the probability of overlap be greater than during was based on the idea of using more complex initial automata. Since we introduced an algorithm for superposing arbitrary automata, we had no limitation on which automata to use initially.

We had an intuition that making the living state more granular could help with our problem. In the context of lifespans of creatures, granular means splitting the the single living state into several states such as "young living" and "old living". Figure 4.8 provides

the automata we used as our granular event model. This automata has 4 parameters: prior probability of staying young α, probability of dying when being young p, probability of dying when being old p' and probability of being born p_{start}.

We utilized the web application we developed to run several experiments for the superposition of granular automata. The visualization of this automata is given in Figure 5.5. Allen's relations were counted in the same way as previously. We ignored all of the information related to the old state and only considered the symbols related to creatures being born and dying. We present several experiments for various assignments of the 4 parameters in Table 5.4.

Experiment 1 is a simple sanity check. We set $\alpha = 1$ to ensure that we can never transition to the old state. In that case, the granular automata behaves the same as the initial automata with a single living state. We can see that we get the same probabilties for overlaps and during. These probabilities also match our results from the previous sanity checks.

Experiments 2-4 ensure that the probability of dying when being old p' is higher than the probability of dying when being young p. We experiment with several assignments of α but in every case we can see that overlap is more probable than during.

Experiment 5 is yet another sanity check. We set $p' < p$ and observe that probability of during becomes higher than the probability of overlap. The reason for this is that young creatures have a higher probability of dying than the old creatures which is an unreasonable assumption. Hence, the result is expected.

The reason why such construction yields the desired outcome is rooted in the fact that we are able to record the history of events by tracking whether they are young or old. The probabilities of transitions can be hence assigned depending on the event's history, allowing us to achieve the required behavior. Our result suggests that using complex models for representing events might be a better approach then just using simple intervals. While earlier timeline modelling approaches did not integrate with complex events models well, the superposition of automata seamlessly allows us to utilize various event models as long as they can be repsented via finite state machines.

Experiment number	**P(Overlaps)**	**P(During)**	α	p	p'	p_{start}
1	0.036	0.037	1.0	0.5	0.5	0.5
2	0.077	0.011	0.2	0.3	0.9	0.5
3	0.097	0.069	0.8	0.2	0.8	0.5
4	0.131	0.106	0.9	0.1	0.6	0.5
5	0.002	0.005	0.9	0.9	0.1	0.5

Table 5.4: Probabilities of overlap and during for various configurations of α, p_{start}, p and p'. The experiments were run for 100000 iterations using `https://cppavel.github.io/fsm-website/fsmexample-granular`.

5.4 Summary

This chapter began with two sanity checks which enabled us to conclude that our model behaved correctly. After that, we explored the idea of superposition with clock automata and explained why adding the notion of clock to the automata does not help us solve the problem of overlap and during having the same probability. We concluded this chapter by discussing the concept of event granularity based on the example of Allen's interval relations. We showed that having increased granularity allows us to fix the problem with the probabilities of overlap and during. Finally, this led us to suggest that using finite state machine superposition to represent timelines of complex is a very flexible and powerful approach.

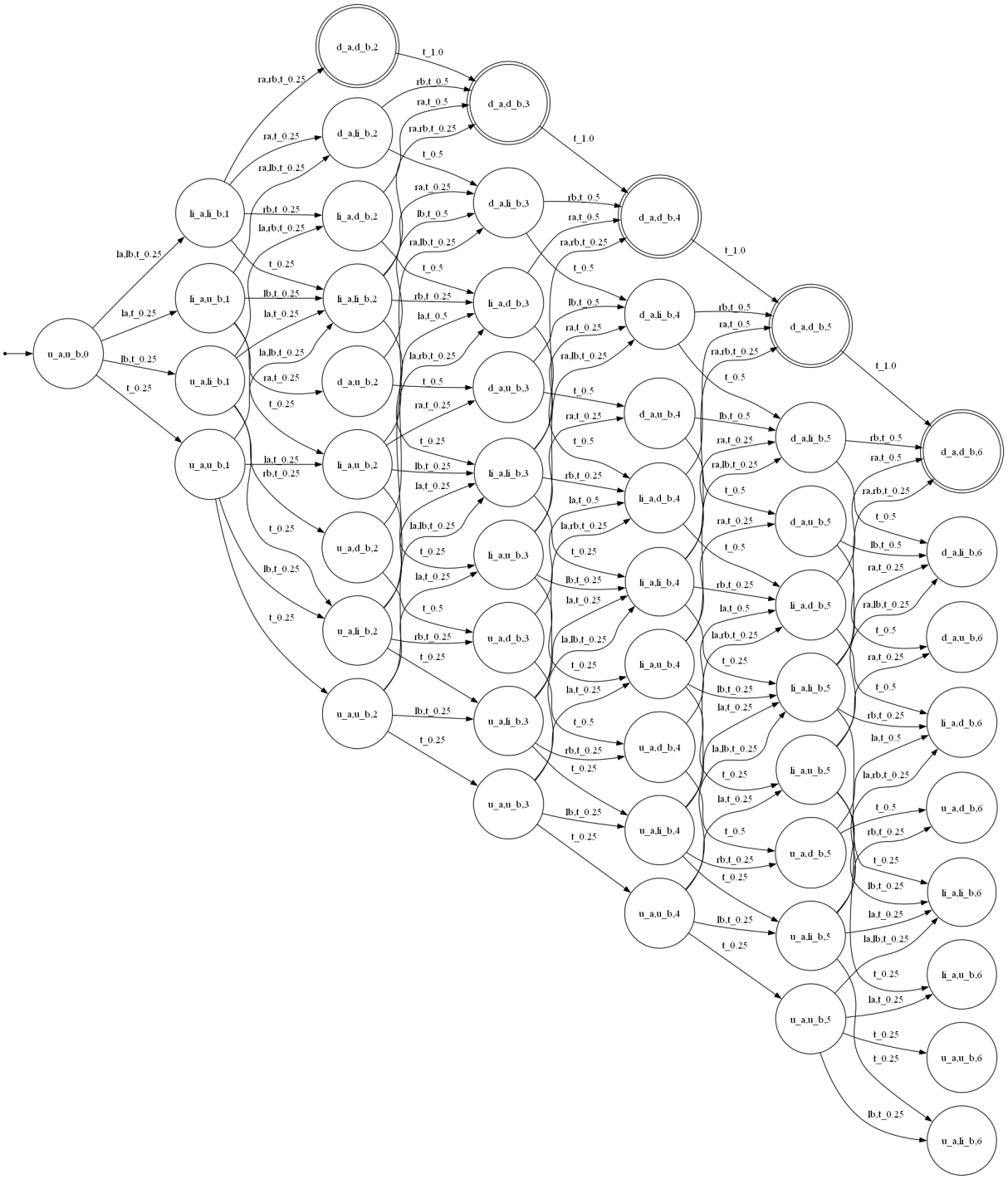


Figure 5.4: Automata from Figure 3.4 superposed with a clock automata for $N = 7$.

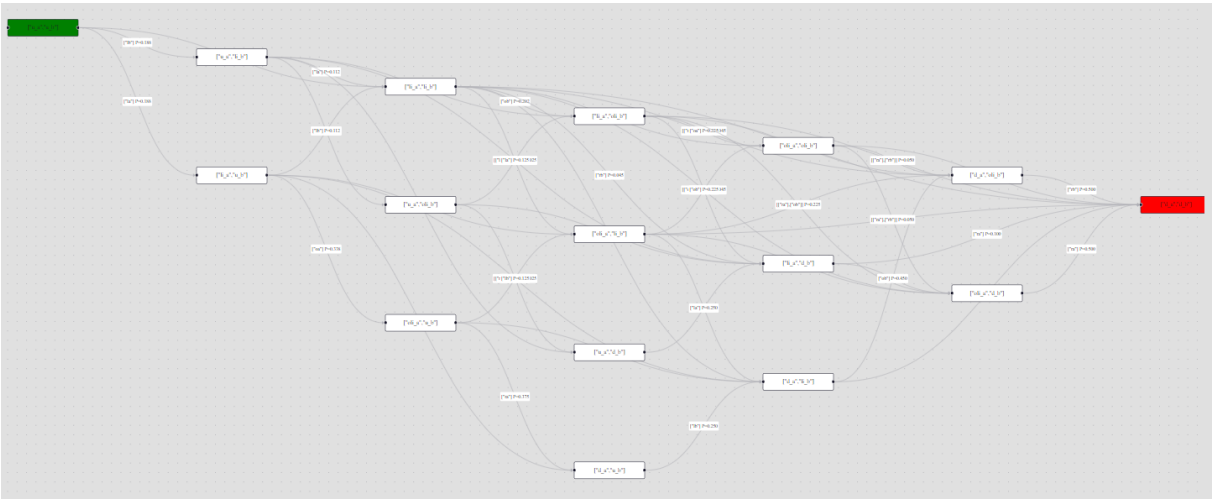


Figure 5.5: Result of superposition of two granular automata for events (creatures) a and b respectively. Each of the initial automata has 4 states and hence the superposed automata consists of 16 states.

Chapter 6

Conclusions & Future Work

We started by exploring the existing literature in the area of timeline representation and prior probabilities of Allen's interval relations. This led us to the work of Suliman (2021) which introduced the idea of using finite state machines to represent timelines. We noted two limitations about existing techniques. Firstly, the future of the timeline was considered completely independent of its past in all of the methods. Secondly, the methods did not consider events that may have a more complicated structure than just starting and finishing. With this in mind we outlined 3 research directions: RD_1, RD_2 and RD_3. We contributed to RD_1 by extending the notion of superposition to the domain of finite state machines, which served as a foundational framework for the rest of our work. In order to contribute to RD_2 and RD_3 we required a software system for superposing, simulating, defining and visualizing arbitrary automata. We created two versions of the system. Our initial system was implemented in Python and had several drawbacks that were eliminated in the second iteration which was a React web app.

With the web app ready, we proceeded to conducting several experiments. We started by performing various sanity checks to verify the correctness of our system. After that, the idea of introducing clocks to automata via superposition was considered. We found that it did not help us resolve the unintuitive probabilities of the overlap and during relations. Hence, we proposed an alternative approach which was based on increasing the granularity of initial events. This approach proved to be successful since it was able to keep the history of the timeline and allowed us to assign probabilities to transitions based on it. Due to this, we concluded that automata superposition is as a powerful framework for reasoning about probabilities of timelines of complex events.

Conducting the experiments allowed us to contribute to RD_2 through creating an exam-

ple timeline representation model where future depended both on the current state and the past. We contributed to RD_3 by providing an example of how finite state machine superposition can be used to model timelines consisting of complex events.

6.1 Future Work

The main direction for future work is applying the proposed timeline probability framework on real life problems. Through real applications one might be able to notice limitations in the framework that were not evident during the design stage. Additionally, the current software system could be extended to work with automata containing loops, which will allow our model to consider the notion of events that may happen multiple times during the timeline. The final further direction of work could involve exploring how temporal reasoning could be incorporated into automata. For example, if we already know that event A must happen before event B we may not need to generate/consider some of the transitions in the automata that contradict this. The starting point for this research direction might lie in exploring how inference over finite temporality strings could be translated into the domain of finite state machines.

Bibliography

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

Allen, J. F. (1984). Towards a general theory of action and time. *Artificial Intelligence*, 23(2):123–154.

Durand, I. and Schwer, S. R. (2008). A tool for reasoning about qualitative temporal information: the theory of s-languages with a lisp implementation. *J. Univers. Comput. Sci.*, 14(20):3282–3306.

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

Fernando, T. and Vogel, C. (2019). Prior probabilities of allen interval relations over finite orders. In *ICAART (2)*, pages 952–961.

Flask Documentation (2024). Flask documentation.

Georgala, K., Sherif, M. A., and Ngonga Ngomo, A.-C. (2016). An efficient approach for the generation of allen relations. In *ECAI 2016*, pages 948–956. IOS Press.

GitHub Actions (2024). Github actions.

GitHub Pages (2024). Github pages.

Gooneratne, N., Tari, Z., and Harland, J. (2008). Vgc: Generating valid global communication models of composite services using temporal reasoning. In Bouguettaya, A., Krueger, I., and Margaria, T., editors, *Service-Oriented Computing – ICSOC 2008*, pages 585–591, Berlin, Heidelberg. Springer Berlin Heidelberg.

Guesgen, H. W. (1989). *Spatial reasoning based on Allen's temporal logic*. International Computer Science Institute Berkeley.

Haimowitz, I. J., Farley, J., Fields, G. S., Stillman, J., and Vivier, B. (1996). Temporal reasoning for automated workflow in health care enterprises. In Adam, N. R. and

Yesha, Y., editors, *Electronic Commerce*, pages 87–113, Berlin, Heidelberg. Springer Berlin Heidelberg.

Jest - JavaScript Testing Framework (Accessed 2024). Jest - javascript testing framework.

Jinja Documentation (2024). Jinja documentation.

JSON Documentation (2024). The json data interchange standard.

Jurafsky, D. and Martin, J. (2024). *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. 3rd ed. draft edition.

Kong, X., Wei, Q., and Chen, G. (2010). An approach to discovering multi-temporal patterns and its application to financial databases. *Inf. Sci.*, 180:873–885.

Lai, F., Chen, G., Gan, W., and Sun, M. (2024). Mining frequent temporal duration-based patterns on time interval sequential database. *Information Sciences*, 665:120421.

Manna, Z. and Pnueli, A. (2012). *Temporal verification of reactive systems: safety*. Springer Science & Business Media.

Morales, A. and Sciavicco, G. (2006). Using temporal logic for spatial reasoning: Spatial propositional neighborhood logic. In *Thirteenth International Symposium on Temporal Representation and Reasoning (TIME'06)*, pages 50–60.

Nelson Michael (Accessed 2024). Deploying react apps to github pages.

NetworkX Documentation (2024). Networkx documentation.

Patel, D., Hsu, W., and Lee, M. L. (2008). Mining relationships among interval-based events for classification. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, page 393–404, New York, NY, USA. Association for Computing Machinery.

Pnueli, A. (1977). The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 46–57.

Prior, A. N. (1957). *Time and modality*. OUP Oxford.

React (2024). React documentation.

REST APIs - IBM (2024). Rest apis - ibm.

Rost, C., Gomez, K., Täschner, M., Fritzsche, P., Schons, L., Christ, L., Adameit, T.,

Junghanns, M., and Rahm, E. (2022). Distributed temporal graph analytics with gradoop. *The VLDB journal*, 31(2):375–401.

Sedgewick, R. and Wayne, K. (2024). *Algorithms*. Fourth edition edition.

Suliman, M. (2021). Timeline probabilities.

Van Beek, P. and Manchak, D. W. (1996). The design and experimental analysis of algorithms for temporal reasoning. *Journal of Artificial Intelligence Research*, 4:1–18.

Visual Automata Python Package (2021). Visual automata python package.

Woods, D. and Fernando, T. (2018). Improving string processing for temporal relations. In *Proceedings of the 14th Joint ACL-ISO Workshop on Interoperable Semantic Annotation*, 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)*.

Zhou, B., Richardson, K., Ning, Q., Khot, T., Sabharwal, A., and Roth, D. (2020). Temporal reasoning on implicit events from distant supervision. *arXiv preprint arXiv:2010.12753*.