

**Checking for Language Inclusion  
with Kurshan's  
Quasi-Complement Construction**

*Adam McDevitt*

Undergraduate Dissertation  
Computer Science  
School of Informatics  
University of Edinburgh

2020



# Abstract

Kurshan's algorithm for language inclusion checks whether  $\mathcal{L}(A) \subseteq \mathcal{L}(B)$  holds for Büchi automata  $A$  and  $B$ . When  $B$  is deterministic the algorithm is complete, but otherwise the algorithm can only witness inclusion some of the time and can never witness non-inclusion. This dissertation describes my implementation of Kurshan's algorithm for checking inclusion between Büchi automata, based on the RABIT framework, written in Java. Work is evaluated and experimental results analysed to come to a conclusion on the applicability of Kurshan's algorithm.

## **Acknowledgements**

I'd like to thank for supervisor Richard Mayr for his guidance and enthusiasm, as well as my project group, and group convener Paul Jackson, for providing an outside perspective on the project. I'd also like to thank my brother Matthew for sanity-checking my crudely sketched automaton diagrams when I was testing error-prone inputs by hand.

# Table of Contents

<b>1</b>	<b>Introduction</b>	<b>7</b>
1.1	Motivation . . . . .	7
1.2	Goals . . . . .	7
1.3	Achievements . . . . .	8
1.4	Structure of the Report . . . . .	8
<b>2</b>	<b>Background</b>	<b>9</b>
2.1	Büchi Automata . . . . .	9
2.1.1	Edge-recurring Büchi Automata . . . . .	10
2.2	Kurshan’s Quasi-Complement, and its Applications to Language Inclusion . . .	10
2.3	The RABIT Framework . . . . .	12
2.3.1	Random Generation of Automata in RABIT . . . . .	12
<b>3</b>	<b>Design, Implementation and a High-Level Description of Kurshan’s Algorithm</b>	<b>13</b>
3.1	A Data Structure for Representing Edge-Recurring Büchi Automata . . . . .	13
3.1.1	Overall Data-Structure . . . . .	14
3.1.2	Representation of $Q$ , the States . . . . .	14
3.1.3	Representation of $\Delta$ , the Transition Function . . . . .	15
3.1.4	Representation of $\Sigma$ , $I$ and $R$ : The Alphabet, Initial States and Recurring Edges . . . . .	15
3.2	Conversion Between State-Recurring and Edge-Recurring Automata . . . . .	16
3.2.1	State-Recurring to Edge-Recurring . . . . .	16
3.2.2	Edge-Recurring to State-Recurring . . . . .	16
3.3	Computing the Quasi-Complement . . . . .	17
3.4	Emptiness of the Intersection of Two Büchi Automata . . . . .	20
3.5	Bringing the Components Together . . . . .	23
<b>4</b>	<b>Testing and Verification</b>	<b>25</b>
4.1	Unit Testing . . . . .	25
4.2	Cross-checking With Pre-Existing RABIT Algorithms . . . . .	25
<b>5</b>	<b>Results and Analysis</b>	<b>27</b>
5.1	Initial Experiment . . . . .	27
5.1.1	Setup . . . . .	27
5.1.2	Hardware . . . . .	28
5.1.3	100 State Automata Results . . . . .	29

5.1.4	1000 State Automata Results . . . . .	33
5.2	Execution Time of Each Stage of the Algorithm . . . . .	37
5.3	Performance on Previously Known “Nasty” Automata . . . . .	37
5.4	Integration Into RABIT . . . . .	38
5.4.1	Motivation . . . . .	38
5.4.2	Results . . . . .	38
5.4.3	Lookahead 12 Minimisation Before Attempting Complete Ramsey Procedure . . . . .	41
5.5	Conclusions: Applicability of Kurshan’s Algorithm . . . . .	42
<b>6</b>	<b>Future Work</b>	<b>45</b>
6.1	Development . . . . .	45
6.2	Experiments . . . . .	45
6.3	Theory . . . . .	46
6.4	Integration Into RABIT . . . . .	46

# Chapter 1

## Introduction

### 1.1 Motivation

Büchi automata have applications in formal verification, particularly model checking of static systems that run for an indefinite amount of time. Such systems might include, for example, operating systems, as they should be ready to accept user input at any moment and do not terminate at any set time. We can model them as running “infinitely” [GTW02]. We might represent such a system with LTL (Linear Temporal Logic). Any LTL system can be converted to an equivalent Büchi automaton, and similarly any LTL property can be converted to a Büchi automaton. If all input words accepted by the system automaton are accepted by the property automaton then the property holds for the system. This is a language inclusion problem. [HR04] [HHK96]

The RABIT framework has various inclusion-checking algorithms implemented already. The most used inclusion checking method, `inclusion_Buchi`, attempts various algorithms on the input automaton pair, both in parallel and sequentially, while minimising the inputs along the way. Should it be found that Kurshan’s algorithm is effective, RABIT could be modified and Kurshan’s algorithm called at an appropriate point in `inclusion_Buchi`.

### 1.2 Goals

My primary aim for this project was to implement, using the RABIT framework, Kurshan’s algorithm for checking language inclusion between Büchi automata. The effectiveness of the algorithm and implementation was then to be evaluated, with key questions including:

- We know that Kurshan’s algorithm *can* return non-null answers when automaton  $B$  is non-deterministic, but do so with any reasonable frequency for non-deterministic  $B$ ? How often does it return null results and how is this affected by properties of the input automata?
- Is Kurshan’s algorithm faster than `inclusion_Buchi`, already implemented in RABIT? How much faster? Is it only faster on certain classes of input?

- How effective is this particular implementation of the algorithm? Could it be improved? Was it developed with good engineering practises?

Should the implementation of Kurshan’s algorithm be deemed useful, it could be integrated into the RABIT framework and perhaps employed by `inclusion_Buchi` in certain situations.

The description of Kurshan’s algorithm in [Kur87] is not intuitive, and so I also aim in this dissertation to give some intuition to the algorithm and its data structures through high-level textual descriptions and diagrams, neither of which were present in [Kur87].

## 1.3 Achievements

Throughout the project I

- implemented Kurshan’s algorithm in RABIT;
- tested Kurshan’s algorithm with both handwritten tests and randomly-generated automata on which Kurshan’s algorithm could be cross-checked with RABIT’s `inclusion_Buchi` for consistency;
- evaluated the effectiveness of Kurshan’s algorithm on its own compared to RABIT;
- implemented an integration of Kurshan’s algorithm into the main RABIT inclusion-checking method and evaluated its effectiveness;
- and wrote more intuitive descriptions of Kurshan’s algorithm, via higher-level descriptions and diagrams.

We will see in Chapter 5 that Kurshan’s algorithm seemed at first to outperform RABIT for some classes of automata, but then see that this was really only due to a greater effort at minimising the input automata before checking for inclusion, rather than anything to do with Kurshan’s algorithm itself. If Kurshan were integrated into RABIT it would be useful only rarely.

## 1.4 Structure of the Report

I will start by providing the necessary background to the project in Chapter 2, including a definition of Büchi automaton and an informal summary of Kurshan’s algorithm for checking language inclusion. In Chapter 3 I will describe this project’s implementation of Kurshan’s algorithm, written in Java, and, to better appreciate each component of the implementation, I explain each step of Kurshan’s algorithm in a more intuitive style than [Kur87]. The design problems that occurred throughout the project, and the decisions made to solve them, shall be discussed here also. I will present data I gathered by recording results of the implementation on randomly generated automata, as well as some previously-known “nasty” automata, in Chapter 5. Results will be analysed and conclusions drawn from them. In Chapter 6 I will draw on the preceding chapters to evaluate the overall effectiveness of Kurshan’s algorithm and my implementation of it. I will also discuss potential future work in this final chapter.



# Chapter 2

## Background

### 2.1 Büchi Automata

Büchi automata are similar to finite automata, but with an important difference. A finite automaton takes as input a finite-length input word and accepts or rejects, and each finite automaton recognises a regular language. A Büchi automaton, however, takes as input an *infinite*-length input word. The language recognised by a Büchi automaton is called an  $\omega$ -regular language. More formally, a Büchi automaton is a tuple  $A = (Q, \Sigma, \Delta, I, R)$  where:

- $Q$  is a finite set of states
- $\Sigma$  is a finite set of symbols, normally referred to as the alphabet
- $\Delta : Q \times \Sigma \rightarrow 2^Q$  is the transition function, as with finite automata, where  $2^Q$  is the power set of  $Q$
- $I \subseteq Q$  is the set of initial states
- $R \subseteq Q$  is the set of recurring states

Rather than being defined with a set of accepting states, a Büchi automaton is defined with a set of “recurring states”  $R$ . [Wika] For a given Büchi automaton, a chain is an infinite sequence of states  $\mathbf{v} = v_0, v_1, \dots$  where  $v_i \in Q$  and  $|\Delta(v_i, v_{i+1})| > 0$  for all  $i$ . An acceptance chain is a chain where for infinitely many values of  $i$ ,  $v_i \in R$  holds. [Kur87].

When we have  $|\Delta(v, u)| > 0$ , we say  $A$  has an edge  $(v, u)$  with associated symbols  $\Delta(v, u)$ .

A sequence  $\mathbf{t} = t_0, t_1, \dots$  where  $t_i \in \Sigma$  for all  $i$ , called an “input word”, follows a chain  $\mathbf{v} = v_0, v_1, \dots$  if for all  $i$  we have  $t_i \in \Delta(v_i, v_{i+1})$ .  $\mathbf{t}$  is accepted by automaton  $A$  if it follows an acceptance chain. The language of  $A$ ,  $\mathcal{L}(A)$ , is the set of all  $\mathbf{t} \in \Sigma^\omega$  (where  $\Sigma^\omega$  is the set of infinitely long strings of symbols in  $\Sigma$ ) such that  $\mathbf{t}$  follows some acceptance chain  $\mathbf{v}$ . If an input word  $\mathbf{t}$  is in  $\mathcal{L}(A)$ , we say  $\mathbf{t}$  is “accepted” by  $A$ .

When we say “Büchi automaton” without qualification we generally mean a non-deterministic Büchi automaton. A deterministic Büchi automaton is one where  $|I| = 1$  and for all  $q \in Q$ ,  $a \in \Sigma$  we have  $|\Delta(q, a)| = 1$ .

For a Büchi automaton  $A = (Q, \Sigma, \Delta, I, R)$ , let  $D_A = (V, E)$  be the directed graph where:

- $V = Q$
- $(v, u) \in E$  if  $|\Delta(v, u)| > 0$

### 2.1.1 Edge-recurring Büchi Automata

Kurshan defines edge-recurring Büchi automata, a variant on the above definition of Büchi automata that is useful for his algorithm of overestimating the complement of a Büchi automaton. In an edge-recurring Büchi automaton,  $R$  is a subset of  $Q \times Q$  and we consider recurring *edges* rather than recurring states. An acceptance chain for an edge-recurring Büchi automaton is a chain  $\mathbf{v} = v_0, v_1, \dots$  such that for infinitely many  $i$ ,  $(v_i, v_{i+1}) \in R$  holds. When it is unclear from context which variant is being discussed, we refer to the standard variant of Büchi automata as “state-recurring Büchi automata”. [Kur87] Chains, input words and the language of an edge-recurring automaton in the same way as for a state-recurring automaton.

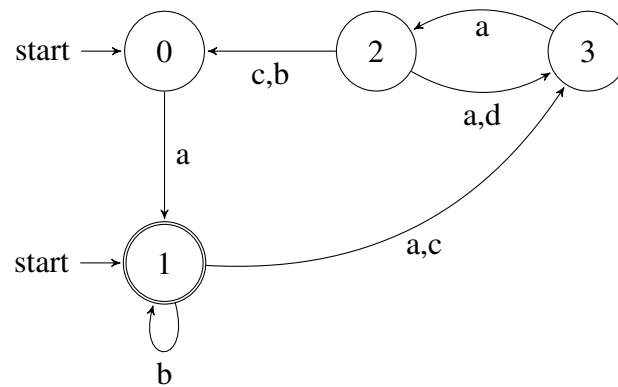
Edge-recurring automata and state-recurring automata can represent exactly the same languages, and conversion between the two representations is straightforward. See [Kur87] for a formal description of conversion between the representations, and Section 3.2 for a more high-level description.

## 2.2 Kurshan’s Quasi-Complement, and its Applications to Language Inclusion

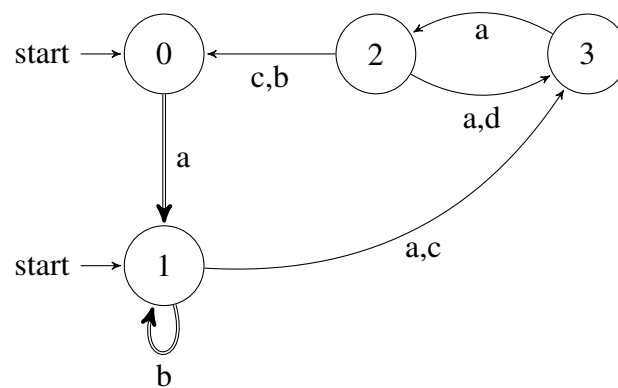
Given Büchi automata  $A$  and  $B$ , we may wish to know whether  $\mathcal{L}(A) \subseteq \mathcal{L}(B)$  holds. To answer this, we can use the method described in [Kur87] to construct an overestimation of the complement of  $B$ ; call this quasi-complement  $\tilde{B}$ . If  $B$  has  $n$  states then  $\tilde{B}$  can be constructed with  $O(n)$  states. We have  $\mathcal{L}(\tilde{B}) \subseteq \mathcal{L}(B)$ , and, if  $B$  is deterministic,  $\mathcal{L}(\tilde{B}) = \mathcal{L}(B)$ . Thus if  $\mathcal{L}(A) \cap \mathcal{L}(\tilde{B}) = \emptyset$  we can conclude that  $\mathcal{L}(A) \cap \mathcal{L}(B) = \emptyset$  and thus  $\mathcal{L}(A) \subseteq \mathcal{L}(B)$ . The reverse implication does not hold unless  $B$  is deterministic. Full details of this procedure can be found in [Kur87]. Details on the procedure will be described throughout Chapter 3, at a higher level than the original paper, in order to better explain the corresponding components of my implementation.

This provides a relatively fast algorithm of checking for inclusion -the previously best known algorithm for obtaining the complement of  $B$  constructed an automaton with  $2^{4n^2}$  states - but does not guarantee an answer unless  $B$  is deterministic.

Unlike finite automata, there exist non-deterministic Büchi automaton cannot be converted



(a) State-recurring Büchi automaton



(b) Edge-recurring Büchi automaton

Figure 2.1: Büchi automata can be represented diagrammatically in a similar way to finite automata. Recurring states of state-recurring Büchi automata are drawn as two concentric circles, and recurring edges of edge-recurring Büchi automata are drawn with double line. I will use this representation throughout the dissertation.

to an equivalent deterministic Büchi automaton.<sup>1</sup> Thus we cannot rely on conversion to a deterministic Büchi automaton to ensure that Kurshan’s algorithm returns a definite answer. A deterministic equivalent of a non-deterministic Büchi automaton can sometimes be found by minimisation, however, as we will see in Chapter 5. This is an important fact and crucial to the applicability of Kurshan’s algorithm in domains with any amount of non-determinism.

In my implementation, a null value will returned when an answer cannot be produced.

## 2.3 The RABIT Framework

RABIT (RAMsey-based Büchi automata Inclusion Testing) is a framework for checking language-inclusion and other properties of both Büchi automata and finite automata. Several inclusion-checking algorithms have been implemented in RABIT, but Kurshan’s algorithm is not among them. The main method for checking inclusion in RABIT, `inclusion_Buchi`, makes use of several algorithms and techniques, trying first the least computationally expensive algorithms that may or may not produce a result, and later resorting to complete, yet expensive, algorithms. RABIT is written in Java. [May]

### 2.3.1 Random Generation of Automata in RABIT

The RABIT framework can also randomly generate automata using the Tabakov-Vardi model. [TV05] This feature takes the following parameters

- Size - Size of  $Q$
- Alphabet Size - Size of  $\Sigma$
- Transition Density - For each symbol  $s \in \Sigma$  the number of transitions labelled with  $s$  is equal to  $d_t \cdot |Q|$ , where  $d_t$  is the transition density
- Acceptance Density - Number of recurring states is equal to  $d_a \cdot |Q|$ , where  $d_a$  is the acceptance density

This project makes use of randomly generated automata for gathering experimental results, and for cross-checking my implementation with inclusion-checking algorithms already in RABIT.

---

<sup>1</sup>This result is folkloric. Here is a proof: Take the language  $L$  of alphabet  $\Sigma = \{a, b\}$  that contains only those strings with finitely many  $bs$ . This is accepted by non-deterministic Büchi automaton  $A = (\{q_0, q_0\}, \{a, b\}, \Delta, \{q_0\}, \{q_1\})$  where  $\Delta(q_0, a) = \{q_0, q_1\}$ ,  $\Delta(q_0, b) = \{q_0\}$ ,  $\Delta(q_1, a) = \{q_1\}$ ,  $\Delta(q_1, b) = \{\}$ . Suppose  $L$  were also accepted by some *deterministic* Büchi automaton  $B$  with recurring states  $R$ .  $B$  accepts  $a^0$  where  $a^0$  is an infinitely long string of  $as$ ,  $B$  must enter  $R$  after reading some finite prefix of  $a^0$ , say at the the  $n_0$ th symbol.  $B$  also accepts  $a^{n_0}ba^0$ , similarly entering  $R$  after the  $n_1$ th symbol for some  $n_1$ . We can continue this construction and build the word  $a^{n_0}ba^{n_1}ba^{n_2}...$  which contains infinitely many  $bs$ , but is accepted by  $B$ . This is a contradiction, and thus  $B$  cannot exist.  $L$  is a language accepted by a non-deterministic Büchi automaton that is not accepted by any deterministic Büchi automata.

## Chapter 3

# Design, Implementation and a High-Level Description of Kurshan's Algorithm

### 3.1 A Data Structure for Representing Edge-Recurring Büchi Automata

The RABIT framework already has graph-like data structures for representing Büchi automata, but these are not suitable for Kurshan's edge-recurring variant. I had to write a new data structure to represent edge-recurring automata. It was important to be able to convert automata between the state-recurring and edge-recurring data structures, as algorithms already featured RABIT - which would be useful for this project - can be used only on the pre-existing state-recurring data structure.

I considered extending the class `FiniteAutomaton`, already present in RABIT. The rationale behind this was that edge-recurring automata differ little in their definition from state-recurring automata - the only difference being a set of recurring edges rather than recurring states. However, any subclass of `FiniteAutomaton` could be used in any context in which the parent class is usable. This is problematic, as it is not safe to assume that all methods that work on state-recurring automaton (e.g. RABIT's pre-existing minimisation methods) would work correctly on edge-recurring automata. Such a class extension would thus be in violation of the Liskov Substitution Principle, an Object Oriented principle that aims to prevent errors and increase maintainability.<sup>1</sup>

Additionally, the description of Kurshan's algorithm to calculate the quasi-complement uses a non-standard matrix-based representation of Büchi automata, rather than the traditional "states and transition function" representation given in Chapter 2. By writing a new data structure, I could structure it in a way that allows for closer correspondence with Kurshan's matrix-based descriptions of the data structure and algorithms and thus implementation was more straightforward - there was no need to translate the descriptions between different representations.

---

<sup>1</sup>The Liskov Substitution principle states that if class *S* is a subclass of class *T* then it should be possible to replace all instances of type *T* with an instance of type *S* without altering the desirable behavior of the program. *S* should require no more than *T*, and promise no less. [Lis87]

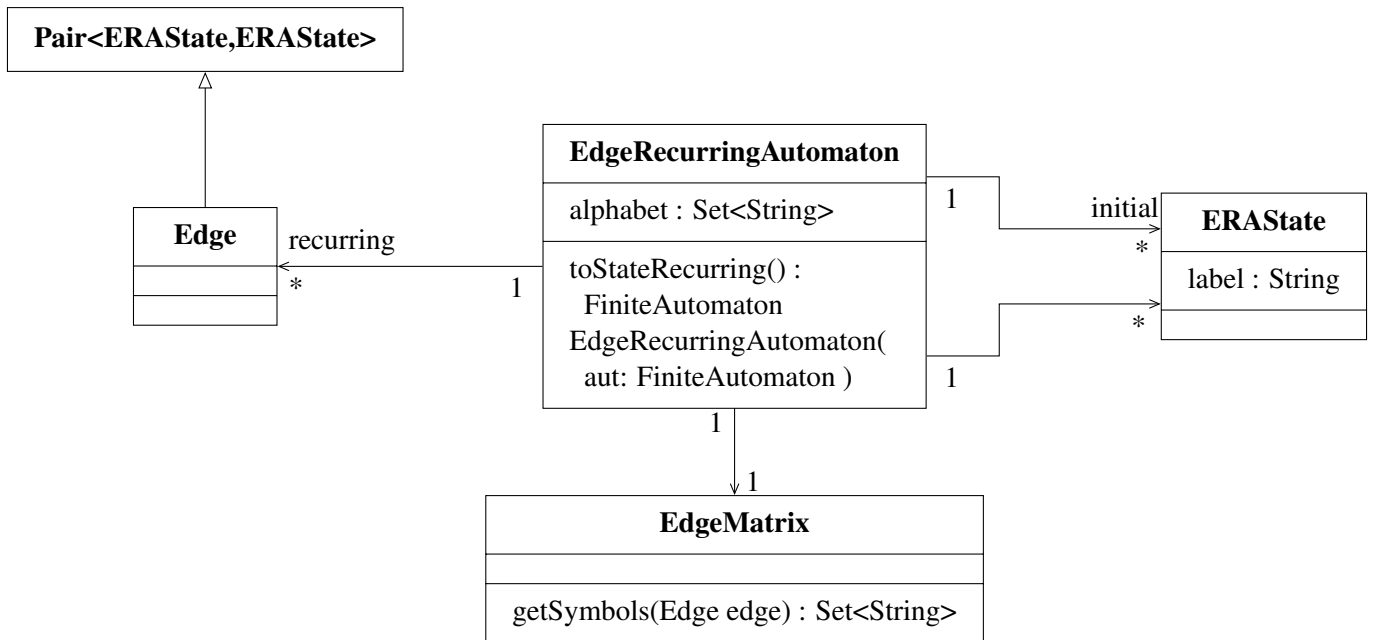


Figure 3.1: Class diagram of the edge-recurring automaton data structures

Conversion between state-recurring automata and edge-recurring automata is relatively computationally inexpensive, compared to other parts of Kurshan’s algorithm, as can be seen in Section 5.2, so if we desire to run, for example, RABIT’s minimisation algorithms on an edge-recurring automaton, we can convert it to state-recurring, minimise, then convert it back again.

Thus I decided to make a whole new data structure to represent edge-recurring Büchi automata.

### 3.1.1 Overall Data-Structure

See Figure 3.1 for a class diagram of the data structures used to represent edge-recurring automata.

An instance of `EdgeRecurringAutomaton` can be constructed by passing a `FiniteAutomaton` instance. This encapsulates the conversion from state-recurring to edge-recurring. It also has a more “manual” constructor where each attribute is passed separately. This second constructor is useful in unit testing, and is also used in the implementation of Kurshan’s algorithm itself. The instance method `toStateRecurring` encapsulates the conversion from edge-recurring to state-recurring.

### 3.1.2 Representation of $Q$ , the States

In an earlier implementation of the data structure, I represented the states of an edge-recurring automata only by integers. I chose to do so as it corresponds more closely with the description of edge-recurring automata in [Kur87], so felt natural. However, this posed problems in implementing Kurshan’s algorithm when the “duplicating” of states is required, as there was no simple way to distinguish a state from its duplicate while both were represented by the same integer. For this reason, I wrote the basic `ERASState` class, whose only attribute is a `String` label

```

public Set<String> getSymbols(ERASState src , ERASState dst) {
    if (srcToDsts.containsKey(src)) {
        Map<ERASState , Set<String>> srcDsts = srcToDsts.get(src);
        if (srcDsts.containsKey(dst))
            return srcDsts.get(dst);
    }
    return emptySet;
}

```

Figure 3.2: The `getSymbols` method of `EdgeMatrix`

that, in this implementation, is used for `toString` methods and nothing else. The use of objects rather than primitives means we can compare `ERASState` instances by checking only for object reference equality and thus it is impossible for two states to “accidentally” be equal.

The set  $Q$  is represented in `EdgeRecurringAutomaton` by a `Set` of `ERASState` instances.

### 3.1.3 Representation of $\Delta$ , the Transition Function

To represent the transitions between states, I wrote a `HashMap`-based abstraction of an adjacency matrix. This is closer to the representations of automata used in [Kur87] and so implementation of later algorithms was more straightforward. I chose this `HashMap` implementation over a traditional adjacency matrix as it allows for the `ERASState` object IDs to be used as keys, and because Java’s `HashMap` interface provides useful auxiliary methods.

I implemented this in the class `EdgeMatrix`, which is essentially a 2-key `HashMap` which takes as keys two instances of `ERASState` and returns a `Set` of `Strings`. The instance method `getSymbols` returns the set  $\Delta(src, dst)$  of symbols. It returns the empty set if there is no transition from `src` to `dst`. See Figure 3.2 for a code snippet.

Before constructing the quasi-complement of an automaton, Kurshan’s algorithm requires that  $|\Delta(q, s)| > 0$  for all  $q \in Q$  and  $s \in \Sigma$ . This is, in fact, baked into Kurshan’s definition of both state-recurring and edge-recurring automata. [Kur87] We call an automaton  $A$  “lockup-free” if it meets this condition. This necessarily-lockup-free definition is non-standard [Wikb] [Wika], and I overlooked it in early implementations of `EdgeRecurringAutomaton`, only to discover it when troubleshooting bugs further down the line. The solution I chose to solve this problem is discussed in Subsection 3.2.1.

### 3.1.4 Representation of $\Sigma$ , $I$ and $R$ : The Alphabet, Initial States and Recurring Edges

$\Sigma$  is represented by a `Set` of strings,  $I$  by a `Set` of `ERASState` instances and  $R$  by a `Set` of `Edge` instances. I chose `Sets` over, say, `Lists` or `arrays`, so that there is no need to worry about duplication of data.

## 3.2 Conversion Between State-Recurring and Edge-Recurring Automata

### 3.2.1 State-Recurring to Edge-Recurring

Constructing an edge-recurring automaton from a state-recurring automaton is straightforward. For a state-recurring automaton  $A = (Q, \Sigma, \Delta, I, R)$  we construct edge-recurring automaton  $A' = (Q', \Sigma, \Delta', I, R')$  where:

- $Q' = Q \cup \{Sink\}$  where *Sink* is a special “sink state”.
- $R'$  contains all edges  $(v, u)$  where  $u \in R$
- The transition function is essentially the same, but “missing transitions” from all vertices are directed to the sink state. We have:
  - $\Delta'(v, u) = \Delta(v, u)$  for all  $v, u \in Q$
  - For any  $v \in Q$ , have  $\Delta'(v, Sink) = \{s \in \Sigma \mid \nexists u \in Q \text{ such that } s \in \Delta(v, u)\}$
  - $\Delta'(Sink, Sink) = \Sigma$

The construction of  $R'$  from  $R$  is intuitive. It is easy to see that an input word is accepted by  $A'$  if and only if it is accepted by  $A$ . What is less immediately clear is the addition of the sink state. This state is added (and transitions to and from *Sink* are added) to ensure that the `EdgeMatrix` attribute of all `EdgeRecurringAutomaton` instances are lockup free, as discussed in Section 3.1.3.

The “state-recurring to edge-recurring” transformation is encapsulated in the constructor of `EdgeRecurringAutomaton`. Its signature is `EdgeRecurringAutomaton(FiniteAutomaton aut)`.

As, initially, I overlooked the lockup-free requirement, the requirement is ensured only when using the constructor with the above signature. The “manual” constructor described in Subsection 3.1.1 allows for the `EdgeMatrix` to be non-lockup-free. Future versions of this implementation might aim to ameliorate this.

When constructing an edge-recurring automaton from state-recurring automaton  $A$ , a “new” alphabet  $\Sigma' \supseteq \Sigma$  can optionally be passed to the constructor. This will result in extra sink-state transitions being added to  $A'$  for symbols in  $\Sigma' \setminus \Sigma$ . This ensures the intersection between  $A$  and  $\tilde{B}$  is constructed correctly, as this implementation requires that the alphabet of  $A$  be a subset of the alphabet of  $\tilde{B}$ , and  $\tilde{B}$  is constructed using this edge-recurring automaton data structure. This augmented constructor has signature `EdgeRecurringAutomaton(FiniteAutomaton aut, Set<String> alphabet)`.

### 3.2.2 Edge-Recurring to State-Recurring

This transformation is more involved than the inverse above.

The full algorithm is described in [Kur87], but without much intuition. I shall give more intuition to the algorithm in the following description. For each  $q \in Q$  that has both an incoming



recurring edge and an incoming *non*-recurring edge it must be split into two states:  $q_{NR}$  and  $q_R$ . We can view  $q_R$  as the “recurring version” of the  $q$  and  $q_{NR}$  as the “non-recurring version”.  $q_{NR}$  and  $q_R$  have the same outgoing non-self-looping edges as  $q$ , but for incoming edges,  $q_{NR}$  is constructed with the incoming *non-recurring* non-self-looping edges and  $q_R$  with the incoming *recurring* non-self-looping edges. For self-looping edges on  $q$ , if recurring, the edge is represented as edges  $(q_{NR}, q_R)$  and  $(q_R, q_R)$ . If a self-loop on  $q$  is non-recurring, it is represented as edges  $(q_R, q_{NR})$  and  $(q_{NR}, q_{NR})$ . Symbols associated with the edges are unchanged. See Figures 3.3a and 3.3b for an example. To intuit: we have split  $q$  into a “recurring” and a “non-recurring” state, to make up for the loss of ability to express distinctly recurring edges.

Ideally I would have captured this transformation in a constructor for `FiniteAutomaton`, so as to mirror the constructor `EdgeRecurringAutomaton(FiniteAutomaton aut)`, but altering the pre-existing code of RABIT was off-limits for this project and writing a subclass `FiniteAutomaton` that differs only in additional constructor would be overkill. Thus I encapsulated the transformation in `EdgeRecurringAutomaton.toStateRecurring`. If Kurshan’s algorithm were to be integrated fully into the RABIT framework, this transformation could easily be relocated to a new constructor for `FiniteAutomaton`.

A complication arose in that Kurshan’s description of edge-recurring Büchi automata allows for multiple initial states, whereas, for legacy reasons, the class `FiniteAutomaton` (used to represent state-recurring automata) in RABIT supports only one initial state. A Büchi automaton with start state set  $I$  can be transformed into a Büchi automaton with a single start state by adding a new state  $q_I$ , setting it as the only initial state, and setting  $\Delta(q_I, s) = \bigcup_{q \in I} \Delta(q, s)$ . See Figures 3.3b and 3.3c for an example. The new start state  $q_i$  is never entered again after reading the first symbol from the input word.

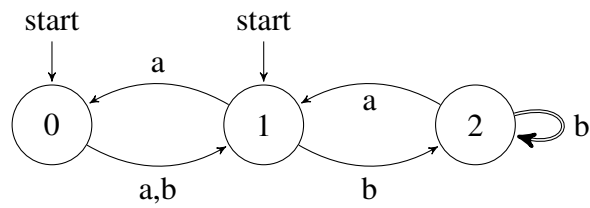
Note that, due to the addition of the sink state, converting an state-recurring automaton  $A$  to edge-recurring then back again does not guarantee an identically structured automaton, but language is preserved by these transformations so it *will* accept exactly the same language as  $A$ . Space complexity is not a concern as  $O(n)$  states and  $O(n)$  edges (where  $n = |Q|$ ) are added.

### 3.3 Computing the Quasi-Complement

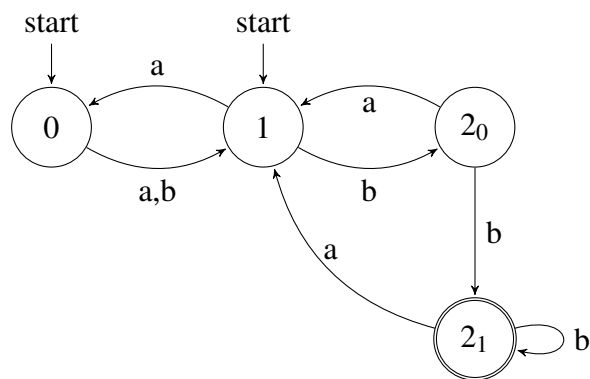
[Kur87] provides an algorithm to convert a *lookup-free* edge-recurring automaton  $B$  into an (incidentally, also *lookup-free*) edge-recurring automaton  $\tilde{B}$  such that  $\tilde{B}$  is an overestimation of the complement of  $B$  - a quasi-complement. In this section I describe that algorithm, give some intuition to its correctness, and present my implementation of the construction.

Given an automaton  $B = (Q_B, \Sigma_B, \Delta_B, I_B, R_B)$ , the basic idea of the first step is to construct an automaton  $M = (Q_B, \Sigma_B, \Delta_M, I_B, R_M)$  such that for all *non*-acceptance chains  $\mathbf{t}$  on  $B$ , there exists some *acceptance* chain  $\mathbf{t}'$  for  $M$  such that  $\mathbf{t}'$  is a suffix of  $\mathbf{t}$  i.e.  $\mathbf{t} = t_0, t_1, \dots, t_k, t_{k+1}, \dots$  and  $\mathbf{t}' = t_k, t_{k+1}, \dots$ . See sub-figures 3.5a and 3.5b for an example. The chain  $\mathbf{t} = a, b, b, a, b, b, a, (b, b, \dots)$  is not accepted by  $B$ , while the chain  $\mathbf{t}' = b, b, \dots$  (a suffix of  $\mathbf{t}$ ) is accepted by  $M$ .

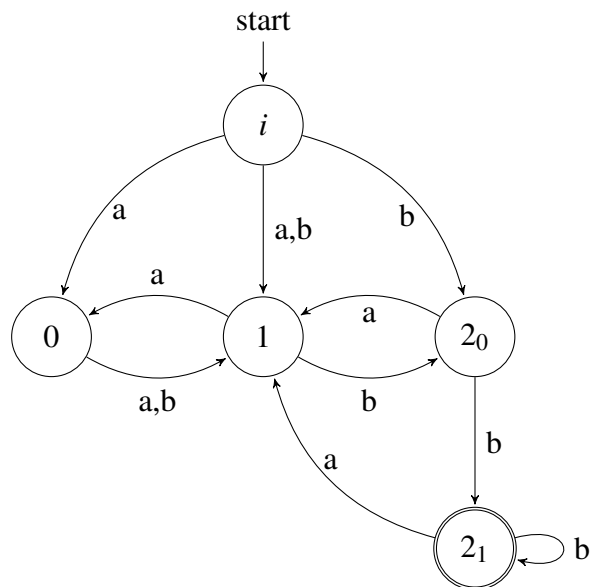
$M$  is constructed from  $B$  by creating a copy of  $B$  with all  $(u, v) \in R_B$  removed. The recurring edges of  $M$  are chosen by finding a maximal spanning forest  $F$  of graph  $D_M$  and, as all cycles in  $D_M$  must contain one of the back-edges, we can ensure all chains in  $M$  are accepting chains by setting  $R_M$  to the set of back-edges. [Kur87] In this implementation, the set of back-edges



(a) Edge-recurring Büchi automaton



(b) Equivalent state-recurring Büchi automaton



(c) Equivalent state-recurring Büchi automaton with one initial state

Figure 3.3: Conversion from edge-recurring Büchi automaton to state-recurring Büchi automaton

```

public static Set<Edge> backEdges(EdgeMatrix edgeMatrix) {
    Set<ERASState> examined = new HashSet<>();
    Set<ERASState> hasParent = new HashSet<>();
    Set<Edge> forest = new HashSet<>();
    Set<Edge> backEdges = new HashSet<>();
    edgeMatrix.getVertices().forEach(src -> {
        if (!examined.contains(src))
            search(src, edgeMatrix, examined, hasParent, forest, backEdges);
    });
    return backEdges;
}

private static void search(ERASState src, EdgeMatrix edgeMatrix, Set<
    ERASState> hasParent, Set<ERASState> ancestors, Set<Edge> forest, Set<Edge
    > backEdges) {
    edgeMatrix.getOut(src).keySet().forEach(dst -> {
        ancestors.add(src);
        if (ancestors.contains(dst))
            backEdges.add(new Edge(src, dst));
        else if (!hasParent.contains(dst)) {
            forest.add(new Edge(src, dst));
            hasParent.add(dst);
            search(dst, edgeMatrix, hasParent, ancestors, forest, backEdges);
        }
        ancestors.remove(src);
    });
}

```

Figure 3.4: Method `backEdges` takes an `edgeMatrix` and returns a set of back edges for some maximal spanning tree

is found through a modified depth-first search which records vertices that have been previously examined. See Figure 3.4 for a code snippet of the implementation. [AHU74] [AHU83]

In Figure 3.4, the line beginning `else if (!hasParent.contains(dst))` is of particular note. `hasParent`, a `Set` of `ERASStates` contains all previously visited states that have some parent state added to the in-construction forest. In the case where we a child state is not an ancestor (in the forest) of the currently examined state, nor does it have a parent, it must be the root of some *other* tree in the forest. When this else-if condition evaluates to true, the subsequent line adds an edge that conjoins two trees of the forest while maintaining that forest is, in fact, a valid forest of  $D_M$ .

To construct  $\tilde{B} = (\tilde{Q}_B, \Sigma_B, \tilde{\Delta}_B, \tilde{I}_B, R_M)$ , we combine  $B$  and  $M$  as follows:

- Let  $\tilde{Q}_B = \{v_B, v_M | v \in Q_B\}$
- Let  $\tilde{I}_B = \{v_B, v_M | v \in I_B\}$
- $\tilde{\Delta}_B$  is defined by:
  - $\tilde{\Delta}_B(v_M, u_M) = \Delta_M(v, u)$
  - If  $(v, u) \notin R_B$ , set  $\tilde{\Delta}_B(v_B, u_B) = \Delta_B(v, u)$ , else set  $\tilde{\Delta}_B(v_B, u_M) = \Delta_B(v, u)$

- To keep  $\tilde{B}$  lockup-free, add a sink state *Sink* for missing transitions, similar to that described in Subsection 3.2.1.

To give some intuition to this construction, observe Figure 3.5:  $\tilde{B}$  combines  $B$  and  $M$  into one graph, with the vertices of  $B$  and  $M$  distinct from each other. Recurring edges in  $B$  are no longer recurring, and, for each recurring edge in  $B$ , we add a corresponding edge from the “ $B$  section” of the automaton to the “ $M$  section” of the automaton. Any input word  $\mathbf{t}$  on a non-accepting chain in  $B$  must either:

- Never hit a recurring edge, in which case it must be on an acceptance chain in  $M$  and thus it is on an acceptance chain in  $\tilde{B}$ .
- Or, at some point, hit a recurring edge for a final time. Let  $\mathbf{t}'$  be the suffix of  $\mathbf{t}$  whose first symbol makes the final “recurring edge” transition in  $B$ . When running  $\mathbf{t}$  on  $\tilde{B}$ , a transition is made from the “ $B$  section” to the “ $M$  section” when the first symbol of  $\mathbf{t}'$  is read. The remainder of  $\mathbf{t}'$  must loop in the “ $M$  section” of the automaton, and all chains in  $M$  are an acceptance chain. Thus  $\mathbf{t}$  is accepted by  $\tilde{B}$ .

Thus we must have that  $\mathcal{L}(\bar{B}) \subseteq \mathcal{L}(\tilde{B})$ .

I implemented this computation in the static method `InclusionKurshan.kurshanComplement`.

### 3.4 Emptiness of the Intersection of Two Büchi Automata

The intersection construction of two Büchi automata is described in [Cho74]. This subsection will give some intuition to the construction, and describe my implementation (and adaptations for this specific use case) of the intersection and emptiness checking algorithms.

The intersection construction of two *finite automata* is familiar and straightforward. For two finite automata  $A_1 = (Q_1, \Sigma_1, \Delta_1, I_1, A_1)$  and  $A_2 = (Q_2, \Sigma_2, \Delta_2, I_2, A_2)$ , their intersection is  $A_\cap = (Q_\cap, \Sigma_\cap, \Delta_\cap, I_\cap, A_\cap)$  where:

- $Q_\cap = Q_1 \times Q_2$
- $\Sigma_\cap = \Sigma_1 \cap \Sigma_2$
- $\Delta_\cap((q_1, q_2), s) = (\Delta_1(q_1, s), \Delta_2(q_2, s))$  for all  $(q_1, q_2) \in Q_\cap$
- $I_\cap = I_1 \times I_2$
- $A_\cap = A_1 \times A_2$

It is easy to see that each state in the intersection is tracking which state we would be in for each of the original automata were we to run them simultaneously on the same input word.

One might naively construct the intersection of two Büchi automata in this way (taking the cross product of the recurring states in place of accepting states) such as in Figure 3.6c, but this is incorrect.

Instead, the intersection state set  $Q_\cap$  sees two states for each  $(q_1, q_2) \in Q_1 \times Q_2$ . Call these states  $(q_1, q_2)_1$  and  $(q_1, q_2)_2$ . The subscript of these states can be interpreted as an annotation

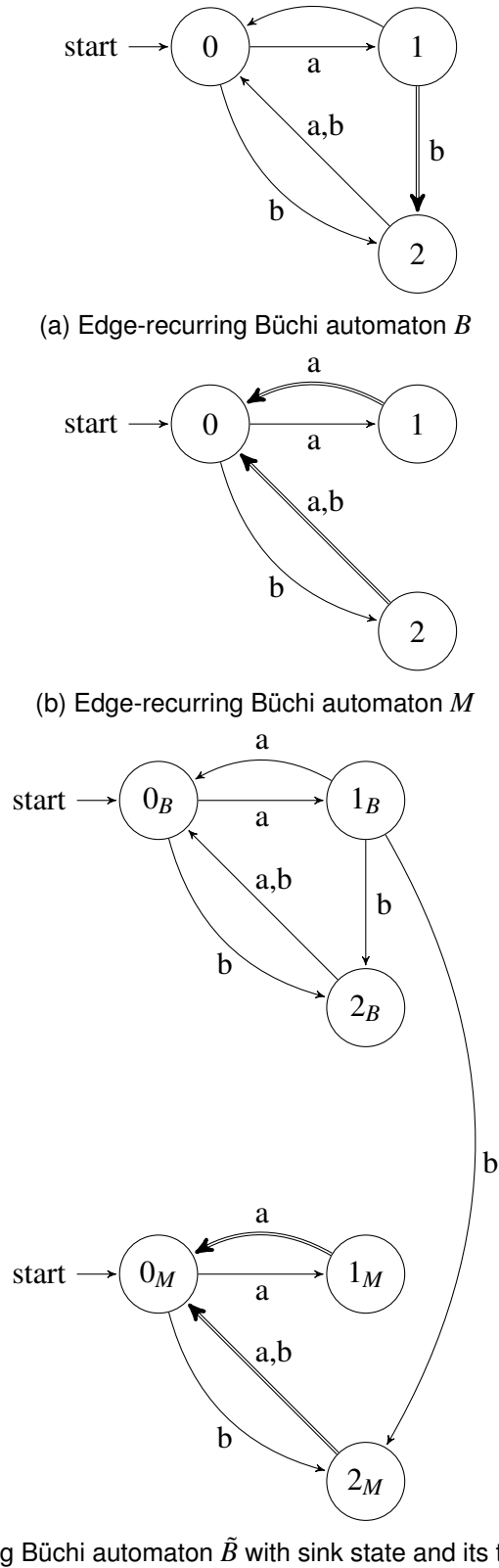
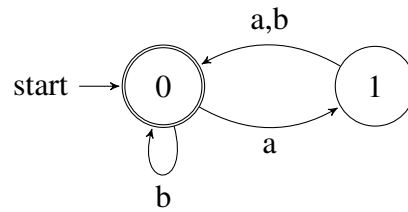
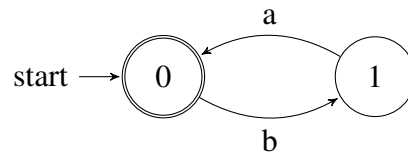


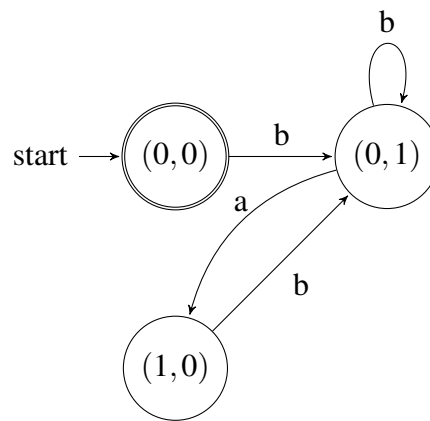
Figure 3.5: Quasi-complement of automaton. It is intuitive to see that any string accepted by  $\tilde{B}$  would not be accepted by  $B$ , as acceptance of a string by  $\tilde{B}$  suggests the execution path would eternally be in non-recurring states in  $B$ .



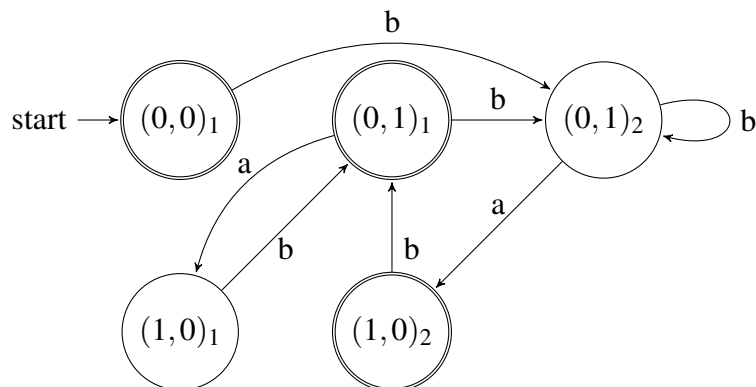
(a) Büchi automaton  $A$



(b) Büchi automaton  $B$



(c) Naive intersection of  $A$  and  $B$ .  $(1, 1)$  is a dead state so has been omitted. Note that  $(ba)^\omega$  is accepted by both  $A$  and  $B$ , but not by the intersection.



(d) Correct intersection of  $A$  and  $B$ . See that  $(ba)^\omega$  is now accepted.  $(0, 0)_2$  is a dead state so has been omitted.

Figure 3.6: Naive and correct intersection constructions

defining which of the original automata we would be “waiting on” to enter a recurring state, were we to run them on the same input word simultaneously.

For two Büchi automata  $A_1 = (Q_1, \Sigma_1, \Delta_1, I_1, A_1)$  and  $A_2 = (Q_2, \Sigma_2, \Delta_2, I_2, A_2)$ , we define their intersection  $A_\cap = (Q_\cap, \Sigma_\cap, \Delta_\cap, I_\cap, A_\cap)$  as follows:

- $Q_\cap = \{(q_1, q_2)_1, (q_1, q_2)_2 \mid (q_1, q_2) \in Q_1 \times Q_2\}$ , as described above
- $\Sigma_\cap = \Sigma_1 \cap \Sigma_2$ , as before
- $I_\cap = \{(q_1, q_2)_1 \mid (q_1, q_2) \in I_1 \times I_2\}$
- $R_\cap = \{(q_1, q_2)_2 \mid (q_1, q_2) \in I_1 \times I_2 \text{ and } q_2 \in R_2\}$
- $\Delta_\cap$  is a bit more complicated to define for Büchi automata. For edges out of non-recurring states, we define them as we would in the finite automata intersection for states with the same subscript i.e.  $\Delta_\cap((q_1, q_2)_i, s) = (\Delta_1(q_1, s), \Delta_2(q_2, s))_i$  if  $q_i \notin R_i$ , for  $i \in \{1, 2\}$ . However, for a state  $(q_1, q_2)_1$  with  $q_1 \in R_1$ , then  $\Delta_\cap((q_1, q_2)_1, s) = (\Delta_1(q_1, s), \Delta_2(q_2, s))_2$ , and symmetrically for  $(q_1, q_2)_2$  with  $q_2 \in R_2$ .

Observe that the subscript “swaps” when a transition is made out of a recurring state. Thus, after an automaton reading the input word leaves a recurring state  $(q_1, q_2)_2 \in R_\cap$  (and note that we must have  $q_2 \in R_2$ ) the subscript swaps to 1 and the automaton cannot enter another recurring state until the subscript swaps back to 2 i.e. until the automaton (enters and) leaves a state  $(q'_1, q'_2)_1$  with  $q_1 \in R_1$ . Thus  $R_\cap$  reading an input word  $\mathbf{t}$  visits states in  $R_\cap$  infinitely many times if and only if  $\mathbf{t}$ , given to  $A_1$  or  $A_2$ , enters states in  $R_1$  or  $R_2$ , respectively, infinitely many times.

See Figure 3.6d for an example of the correct Büchi automaton intersection construction.

The importance of the intersection construction to this implementation of Kurshan’s algorithm in the need check whether  $\mathcal{L}(A \cap \tilde{B}) = \emptyset$  holds for automata  $A$  and  $B$ . It seems remiss to construct the entirely new automata (with  $O(|Q_A| \cdot |Q_B|)$  states) when an acceptance chain could be found early in its construction. For this reason, I wrote the emptiness check to run while dynamically constructing the intersection. We can identify (and construct) the reachable states of any state of the intersection given the original transition functions of  $A$  and  $B$ . Emptiness is checked with a novel version of Tarjan’s algorithm [Tar72] that has been modified to take two automata  $A$  and  $\tilde{B}$  as input and dynamically construct the graph of their intersection,  $D_{A \cap \tilde{B}}$ , while checking for SCCs (Strongly Connected Components) that contain recurring states. This allows for early termination in the cases where emptiness does not hold i.e. the cases where inclusion does not hold between  $A$  and  $B$ . In cases where we want to minimise  $A$  and  $B$  prior to performing Kurshans algorithm, the time to construction the intersection is dominated by the minimisation time, so the early termination helps little in these cases. (We will see in Chapter 5 that we want to perform some minimisation in most cases if there is any amount of non-determinism.)

### 3.5 Bringing the Components Together

I wrote the primary interface to this implementation of Kurshan’s algorithm as a single method. It is short and straightforward to understand. See Figure 3.7.

```

public static Boolean includes(FiniteAutomaton a, FiniteAutomaton b, int
    lookahead) {
    a = lookahead < 1 ? min.removeDead(a) : min.Minimize_Buchi(a, lookahead);
    b = lookahead < 1 ? min.removeDead(b) : min.Minimize_Buchi(b, lookahead);
    Set<String> sharedAlphabet = new HashSet<>();
    sharedAlphabet.addAll(a.getAllTransitionSymbols());
    sharedAlphabet.addAll(b.getAllTransitionSymbols());
    FiniteAutomaton complementB = kurshanComplement(b, sharedAlphabet);
    if (Emptiness.isEmptyIntersection(a, complementB))
        return true;
    else if (isDeterministic(b))
        return false;
    else
        return null;
}

```

Figure 3.7: The final inclusion checking method

The parameters *a* and *b* are the system and specification automata respectively.

The parameter *lookahead* is used to determine the strength of minimisation performed on the input automata, prior to executing the steps of Kurshan’s algorithm discussed in the preceding section. Minimisation of Büchi automata is important to this project, as by reducing the number of states we also decrease the non-determinism of an automaton, so to speak, and sometimes even minimise a non-deterministic automaton to an equivalent deterministic one; this means that Kurshan’s algorithm of inclusion checking is more likely to return an answer if the specification automaton *B* is minimised prior to computing the quasi-complement. RABIT already contains minimisation methods. [CM19] These methods take an integer “lookahead” parameter. Increasing this parameter generally results in a smaller number of states in the minimised automaton, and in a less non-deterministic automaton, but increasing the lookahead also increases the execution time of the minimisation algorithm. The algorithm in [Kur87] makes no mention of minimisation, but in practice, if Kurshan’s algorithm is to be used on non-deterministic automata, it would be wise to minimise first in order to avoid `null` answers. See Chapter 5.



# Chapter 4

## Testing and Verification

### 4.1 Unit Testing

I generally wrote unit tests for each non-trivial method, before beginning work on a new method, to ensure correctness of the individual components of the implementation. Over 35 unit tests were written. Branch coverage of over 93% was achieved for the package `kurshan.algorithms`, where `kurshan`'s algorithm itself is actually implemented, while branch coverage of over 84% was achieved in the package `kurshan.datastructure`. This coverage could be improved from more extensive testing. However, many of the missed branches were in straightforward methods such as `compareTo` methods that were not used in the final implementation. These methods were left as I did not want to assume they would never be used in future.

### 4.2 Cross-checking With Pre-Existing RABIT Algorithms

As RABIT can already check inclusion in `inclusion_Buchi` (which combines several algorithms) I could cross-check Kurshan's algorithm with `inclusion_Buchi` to further ensure correctness. Automata pairs were generated randomly using the Tabakov-Vardi method (see








Element	Coverage	Covered Branches	Missed Branches
▶  kurshan.utils	0.0 %	0	16
▶  kurshan.algorithms	93.6 %	176	12
▶  kurshan.automata	63.3 %	19	11
▶  kurshan.datastructure	83.3 %	55	11
▶  kurshan.test	50.0 %	11	11
▶  kurshan.mainfiles	0.0 %	0	2
▶  kurshan		0	0

Figure 4.1: Branch coverage from the project's unit tests, as measured by Eclipse

subsection 2.3.1) with following parameters :

1. Size = 25
2. Alphabet Size = 2
3. Transition Density = 1.8
4. Acceptance Density = 0.5

I chose these parameters as they are similar to the parameters used in later data collection, and because inclusion holds for automata pairs generated with these parameters at a reasonable frequency. The size is 25 to allow for relatively fast computation.

I wrote a script to generate these automata and run both Kurshan's algorithm and `inclusion_Buchi` on each automata pair, and report any pairs for which the two methods are inconsistent in their answers i.e. if Kurshan's algorithm's answer is non-null, then `inclusion_Buchi` should return the same answer. If an inconsistency was found, the offending two automata would be saved to file and unit tests written to replicate the issue. I then found and fixed the bugs associated with these automaton pairs. I generated and saved in this fashion until all new unit tests passed and no new inconsistencies could be found in any reasonable amount of time

In the end, over 100,000 automata pairs were checked tested consistent answers. It is safe to say that, if any erroneous cases remain for these automaton parameters, they do not arise often. It might be prudent in future to cross check with automata generated from more varying parameters. Problematic edge-cases might be found sooner in this way. The results gathered in Chapter 5 still failed to flag up any further inconsistencies, strengthening my belief that the implementation is correct.

# Chapter 5

## Results and Analysis

This chapter presents results from various experiments to analyse the performance of Kurshan’s algorithm and compare it to RABIT’s pre-existing `inclusion_Buchi` method.

### 5.1 Initial Experiment

These experiments analyse the performance of Kurshan’s algorithm on non-deterministic automata generated by various parameters.

We wish to compare the performance of my implementation of Kurshan’s algorithm to the performance of RABIT’s pre-existing inclusion-checking method `inclusion_Buchi`, which makes use of several algorithms to compute an answer.

For brevity<sup>1</sup> in the rest of this chapter I will frequently refer to Kurshan’s algorithm as just “Kurshan” and `inclusion_Buchi` as “RABIT”.

#### 5.1.1 Setup

For each experiment in this section, a set of automaton pairs was generated using the Tabakov-Vardi method (see sub-section 2.3.1) with chosen parameters. The acceptance density used for all sets was 0.5 and alphabet size was 2. The other parameters differ between the sets and will be stated explicitly before each set of results. Kurshan’s algorithm and RABIT were run on each of these pairs, with their outputs and execution times recorded.

In some cases `inclusion_Buchi` can take an extremely long time, resorting to complete yet time-consuming inclusion checking algorithms. There are, in fact, still automaton pairs with fewer than just fifty states that remain to be solved by RABIT, due to stalling. See Section 5.3. For this reason, I set a timeout on RABIT running on each experiment. If the timeout is reached before RABIT can output an answer, we consider it to have failed to return an answer and we then write the output as null. The duration of these timeouts is stated explicitly prior to sets of results. Timeouts were chosen based on a time that was observed to be an upper limit on the

---

<sup>1</sup>and reduce the amount monospace font!

execution time of the non-stalling inputs, from when experiments were run without a timeout in previous attempts.

For each set of automaton pairs, I tried minimising the input automata  $A$  and  $B$  with various lookaheads before executing the rest of Kurshan’s algorithm. Lookaheads measured were 0, 1, 4, 8 and 12, but for the sake of brevity I will present mostly results for lookaheads 0, 4, 8 and (in one case) 12; these were the lookaheads with the best range of results. When I say “lookahead 0”, this is not technically correct, and I use it as shorthand for when no advanced minimisation is used - the only “minimisation” is the removal of dead states. See Section 3.5 for an explanation of minimisation and lookaheads in RABIT.

My earlier implementations of Kurshan’s algorithm for this project further minimised the quasi-complement  $\tilde{B}$  after computing it from a minimised  $B$ , but, as will be seen Subsection 5.2, the time to check for emptiness of the intersection of  $A$  and  $\tilde{B}$  is dominated by the time to minimise  $B$  in the first place, so any minimisation of  $\tilde{B}$  would likely *increase* execution time without even reducing the chances of a null answer being returned.

I partitioned the experimental results into categories based on answers returned by the algorithms and (if relevant) whether or not the automaton  $B$  was minimised to a deterministic automaton.

In addition to the execution times of each partition, we are also interested in the relative size of each partition: How often does Kurshan return an answer when inclusion holds? What about when inclusion doesn’t hold? Are there partitions in which RABIT times out but Kurshan returns an answer? I made the determinism partitions to find how often Kurshan’s algorithm returns an answer for non-deterministic  $B$ .

For each set of results, I will discuss general observations. Full result tables, for the keen reader, are given in Figures 5.1 and 5.3. For most results, median times are observed, rather than mean times, to avoid outliers asserting too much influence. I will make explicit which kind of “average” is presented.

### 5.1.2 Hardware

All experiments were run on the University of Edinburgh `student.compute` general purpose servers. [Edi] The `longjob` command to run the experiments was started with a nice [Ker10] value of 10, so as not to hog the server resources. It is inevitable that there is some noise in the results due to experiments run by other students simultaneously, and it is for this reason that such large sets of automaton pairs were used. With 500 pairs of automata, the noise will matter less and hopefully even out.

Specifications of the “`student.compute`” servers are as follows:

- CPU: 40 × Intel Xeon CPU E5-2690 v2 @ 3.00GHz
- RAM: 400GB

Although, as noted, only a small portion of these resources was used.

Ideally the experiments would have been run on a single dedicated machine rather than servers

accessible to all students, but that was not an option for this project due to how long the experiments took.

### 5.1.3 100 State Automata Results

For each of the three experiments in this subsection, a set of 500 pairs of automaton were generated, where each automaton in a pair has 100 states before minimisation. Each set was generated with a different transition density. (See 2.3.1 for an explanation of transition density and other parameters.)

RABIT was restricted by a timeout of 5 minutes for these experiments.

Full timing results are given in Figure 5.1.

RABIT was restricted by a timeout of five minutes for these experiments.

#### 5.1.3.1 Transition Density 1.6

A partition of particular interest is that where inclusion holds but  $B$  has *not* been minimised to a deterministic automaton by Kurshan. That partition in this results set sees Kurshan confirm inclusion in 32 or 49 cases with lookahead 8, and poorer results with lower lookaheads. In the partition where Kurshan returns true for these non-deterministic automata, RABIT is faster than Kurshan by a factor of over 5 for lookaheads greater than 4. It is useful to know that Kurshan even *can* return a result for these automata with any reasonable likelihood.

In the partitions where RABIT confirms inclusion and Kurshan returns the correct answer, Kurshan outspeeds RABIT only when it can  $B$  to a deterministic automaton - it's faster by a factor of around 8 for these cases. For the pairs where Kurshan cannot minimise  $B$  to an equivalent deterministic automaton, RABIT is extremely fast, relative to the rest of the results in Subfigure 5.3a.

RABIT is also slower where inclusion holds yet Kurshan returns null, though this is less useful as we don't actually get an answer out of Kurshan's algorithm. With lookahead 8, Kurshan is about 8 times faster than RABIT for this partition, and even faster for lower lookaheads. But a lower lookahead does of course see Kurshan return null more often.

This transition density for 100-state automata sees RABIT timeout at five minutes most frequently - with 38 timeouts - but Kurshan could not solve these timed-out automaton pairs either, even with lookahead 8. Kurshan does save time in the partitions where RABIT times out - by a large factor. This time reduction is not particularly useful given that Kurshan doesn't return an answer for these results, but it is perhaps a silver lining. Not *too* much time is wasted waiting for Kurshan, whereas RABIT could have taken far longer than five minutes had a timeout not been implemented. In Subsubsection 5.1.4.1, which analyses the same transition density on 1000 state automata, we will see that in cases where RABIT times out, it is probable that inclusion *does* hold.

When RABIT returns false, Kurshan is slower by at least a factor of 5 for non-zero lookaheads, regardless of answer returned by Kurshan. It is of course even slower for higher lookaheads. This trend will repeat in all sets of automata.

	L=0	L=4	L=8
R=true, det=true, K=true	2 / 0.0197 / 0.0889	8 / 1.71 / 0.2	35 / 3.95 / 0.49
R=true, det=false, K=true	31 / 0.0308 / 0.121	30 / 0.031 / 0.206	32 / 0.0319 / 0.371
R=true, det=false, K=null	51 / 4.42 / 0.129	46 / 4.52 / 0.298	17 / 4.86 / 0.633
R=false, K=false	20 / 0.041 / 0.0987	24 / 0.0424 / 0.223	48 / 0.0707 / 0.456
R=false, K=null	358 / 0.0762 / 0.131	354 / 0.0762 / 0.333	330 / 0.0762 / 0.608
RABIT=timeout, K=true	0 / NaN / NaN	0 / NaN / NaN	0 / NaN / NaN
RABIT=timeout, K=false	0 / NaN / NaN	0 / NaN / NaN	0 / NaN / NaN
RABIT=timeout, K=null	38 / 300 / 0.134	38 / 300 / 0.275	38 / 300 / 0.576

(a) 100 States, transition density 1.6

	L=0	L=4	L=8
R=true, det=true, K=true	0 / NaN / NaN	87 / 5.83 / 0.314	149 / 5.83 / 0.499
R=true, det=false, K=true	16 / 0.0321 / 0.122	20 / 0.0326 / 0.263	20 / 4.66 / 0.53
R=true, det=false, K=null	163 / 5.93 / 0.132	72 / 5.99 / 0.376	10 / 6.74 / 0.779
R=false, K=false	16 / 0.0406 / 0.1	91 / 0.0754 / 0.343	154 / 0.0755 / 0.505
R=false, K=null	303 / 0.0772 / 0.133	228 / 0.0777 / 0.416	165 / 0.0781 / 0.674
RABIT=timeout, K=true	0 / NaN / NaN	0 / NaN / NaN	0 / NaN / NaN
RABIT=timeout, K=false	0 / NaN / NaN	0 / NaN / NaN	0 / NaN / NaN
RABIT=timeout, K=null	2 / 300 / 0.14	2 / 300 / 0.352	2 / 300 / 0.775

(b) 100 States, transition density 1.8

	L=0	L=4	L=8
R=true, det=true, K=true	0 / NaN / NaN	222 / 7.35 / 0.328	235 / 7.33 / 0.547
R=true, det=false, K=true	11 / 0.0332 / 0.127	11 / 6.19 / 0.255	12 / 6.73 / 0.505
R=true, det=false, K=null	240 / 7.37 / 0.136	18 / 6.92 / 0.442	4 / 7.19 / 0.647
R=false, K=false	8 / 0.0422 / 0.1	179 / 0.0802 / 0.36	193 / 0.0796 / 0.498
R=false, K=null	241 / 0.0804 / 0.135	70 / 0.0802 / 0.398	56 / 0.0814 / 0.523
RABIT=timeout, K=true	0 / NaN / NaN	0 / NaN / NaN	0 / NaN / NaN
RABIT=timeout, K=false	0 / NaN / NaN	0 / NaN / NaN	0 / NaN / NaN
RABIT=timeout, K=null	0 / NaN / NaN	0 / NaN / NaN	0 / NaN / NaN

(c) 100 States, transition density 2.0

Figure 5.1: Timing results for kurshan's algorithm compared to RABIT's algorithm. "L" denotes the lookahead used by Kurshan, and the leftmost row describe the partitions of results. "R" refers to the outcome of RABIT's pre-existing algorithms - true, false, or timed out - while "det" refers to whether Kurshan minimised  $B$  to a deterministic automaton and "K" refers to the answer returned by Kurshan. Cell contents take the format "number of occurrences of partition / median RABIT time / median Kurshan time" where time is given in seconds. Results for larger automata in Figure 5.3.

By increasing lookahead, we can greatly decrease the number of null answers for inclusion-positive input pairs - from 51 null answers to 17 - but the improvements for the non-inclusion partitions are barely an improvement: from 358 null answers to 330.

For this first set of automata, comparing the algorithms as they are, there does not seem to be much reason to use Kurshan's algorithm over existing methods. In domains where inclusion mostly holds, perhaps Kurshan could see *some* applicability, but in domains that see non-inclusion hold any significant amount of time, Kurshan would be near-useless, as even with lookahead 8 we see Kurshan return null for more than 80% of inclusion-negative pairs. Even when it does return false, Kurshan is considerably slower than RABIT.

### 5.1.3.2 Transition Density 1.8

In the partitions where inclusion holds and  $B$  has not been made deterministic, this transition density sees an even lower proportion of null answers from Kurshan's algorithm. With lookahead 8 there are only 10 null answers of 179 inclusion-positive input pairs. For lookahead 8, Kurshan now outspeeds RABIT in all partitions where inclusion holds, regardless of whether  $B$  is minimised to deterministic.

Execution time in partitions where inclusion doesn't hold are similar to Subsection 5.1.3.1, though Kurshan is generally slightly faster than before. Thankfully, we also see a reduction in null answers for the non-inclusion partitions - the amount of null answers for transition density 1.6 was distressingly high. With lookahead 8, we see a non-null answer from Kurshan for about half of the automaton pairs for which we know inclusion to not hold.

RABIT also times out significantly less for this transition density. The timeout partitions are not large enough for us to draw many conclusions from the data regarding execution time. As before, Kurshan could not return an answer in experiments where RABIT timed out.

See Figure 5.2 for branching case diagrams that visualise how often each partition of results arises this set of automaton pairs for three different lookaheads.

As Kurshan (with lookahead 8) so rarely returns null on inclusion-positive pairs in this experiment, and outspeeds RABIT by a large margin on these automata, It *could* be beneficial to run Kurshan in practise on such automata. One might write a simple combined algorithm that runs Kurshan's algorithm on the input pair, then if a null answer is returned, the (now minimised) automata are passed to RABIT's `inclusion_Buchi` method. If inclusion needed to be checked between a large number of pairs, this combined algorithm should see time saved if the pairs tend towards inclusion holding, seeing as how Kurshan rarely returns null and executes faster than RABIT for inclusion-positive pairs. More nuanced and realistic integration of Kurshan into RABIT's `inclusion_Buchi` method will be discussed in Section 5.4.

### 5.1.3.3 Transition Density 2.0

This increase in transition density sees similar changes in proportion of null answers. Kurshan's algorithm now only returns null in a total of 60 of 500 experiments, with the vast majority of null answers being for inclusion-negative cases. RABIT never timed out on any automata generated by these parameters, and execution times are otherwise similar to those of subsubsection 5.1.3.2. Kurshan still tends to outspeed RABIT in the inclusion-positive cases, now

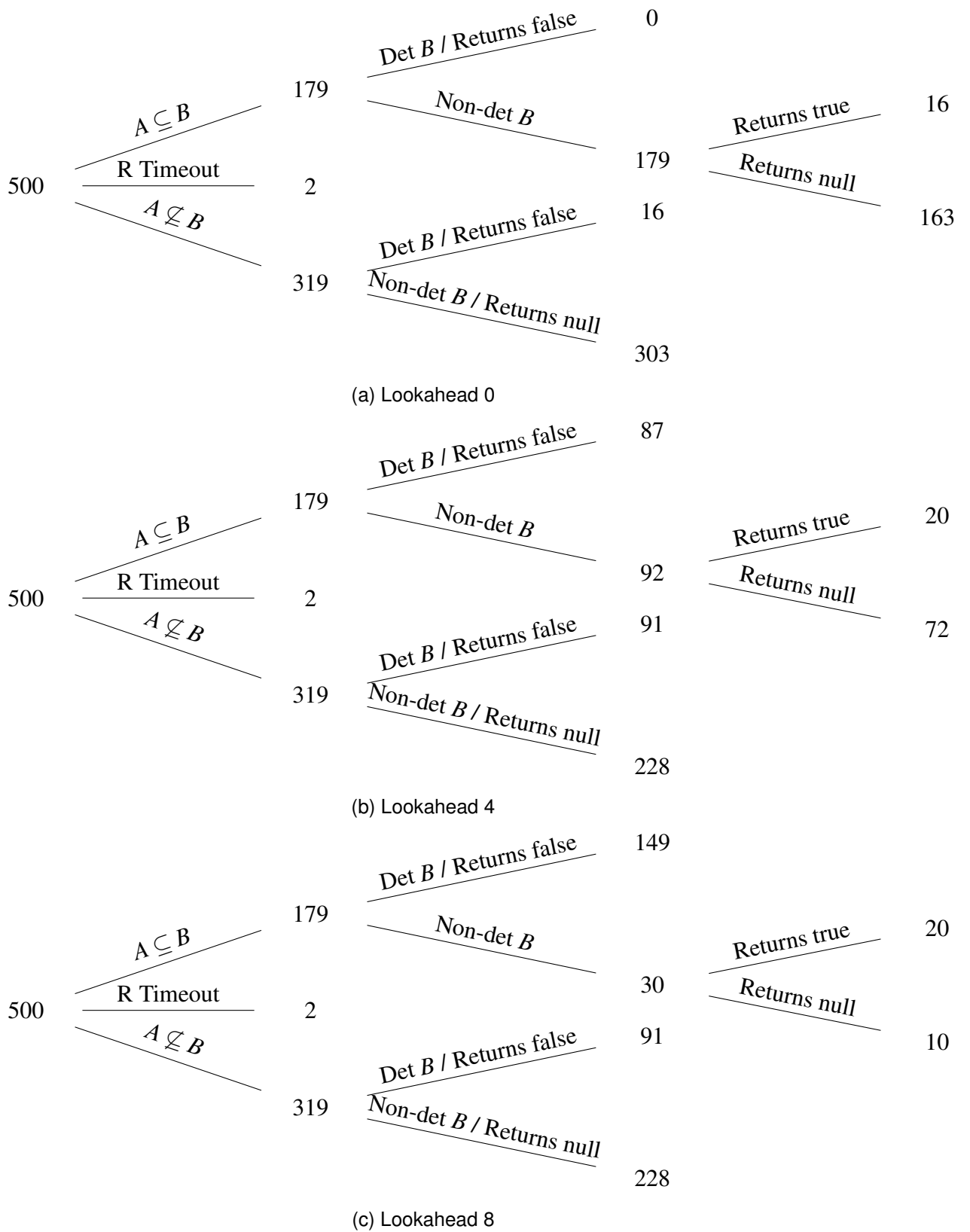


Figure 5.2: Case Diagram for 100 States / 1.8 Transition Density



with far fewer null answers. The combined algorithm proposed at the end of 5.1.3.2 would perform slightly better on this set of automata, as inclusion holds more often.

#### 5.1.3.4 General Conclusions on 100 State Automata

On lower transition densities, Kurshan probably returns null too often to be used in practise. The simple combined algorithm - proposed in Subsubsection 5.1.3.2 - could save some time on automata of transition densities 1.8 and 2.0, as:

- Kurshan generally outspeeds RABIT when inclusion holds
- These higher transition densities see fewer null answers
- Inclusion tends to hold more often for higher transition densities

. These results are not earth-shattering, but we now know that Kurshan's algorithm is not useless in practise, and perhaps could have some applicability if we delve deeper into other sets of automaton pairs. Larger automata, for instance...

### 5.1.4 1000 State Automata Results

For each of the following three experiments, automaton pairs were generated with the same parameters as in Subsection 5.1.3, but with the number of states increased to 1000.

A longer timeout of 10 minutes was used.

Full timing results are given in Figure 5.3.

#### 5.1.4.1 Transition Density 1.6

RABIT returns true less often for these larger automata. Regardless, we can observe that, in the cases where we know inclusion to hold, while increasing lookahead up to 8 does increase the time taken by Kurshan's algorithm, it doesn't change the partitions *whatsoever*. Due to this anomaly, I reran the experiments with an even higher lookahead of 12 as a sanity check, and we can see that the partition sizes *do* change slightly for this higher lookahead. An extra column with lookahead 12 as been added to the table in Subfigure 5.3a only. With this extra run of experiments, the overall difference in number of null results from Kurshan (looking only at the cases where RABIT does not timeout) is small. We go from a total of 363/406 null results across all partitions to 335/406.

Kurshan's algorithm witnesses inclusion in almost all 18 cases where RABIT does so - regardless of lookahead - for this set of automaton pairs, but RABIT returns its results so blindingly fast that Kurshan cannot compete. We can safely assume that these are near-trivial cases of inclusion checking. Two of them, in fact, were deterministic after just the removal of dead states. For lookahead 12, Kurshan returns true in all cases where RABIT does so.

In the partitions where inclusion doesn't hold, Kurshan's results are again disappointing for this transition density. For the lookaheads lower than 12, non-inclusion is only witnessed in 26/388 cases, and Kurshan is slower than RABIT by at least a factor of 15. The time difference between Kurshan with lookahead 12 and RABIT, in these partitions, is again stark.

	L=0	L=4	L=8	L=12
R=true, det=true, K=true	2 / 0.0713 / 0.111	2 / 0.0713 / 0.0187	2 / 0.0713 / 0.0176	3 / 0.0856 / 0.023
R=true, det=false, K=true	15 / 0.368 / 0.619	15 / 0.368 / 12.6	15 / 0.368 / 30.6	15 / 0.368 / 265
R=true, det=false, K=null	1 / 134 / 0.589	1 / 134 / 26.5	1 / 134 / 71.5	0 / NaN / NaN
R=false, K=false	26 / 0.505 / 0.149	26 / 0.505 / 12.9	26 / 0.505 / 29.9	53 / 1.14 / 262
R=false, K=null	362 / 1.31 / 0.626	362 / 1.31 / 24.9	362 / 1.31 / 60.5	335 / 1.31 / 431
RABIT=timeout, K=true	0 / NaN / NaN	0 / NaN / NaN	0 / NaN / NaN	22 / 600 / 315
RABIT=timeout, K=false	0 / NaN / NaN	0 / NaN / NaN	0 / NaN / NaN	0 / NaN / NaN
RABIT=timeout, K=null	94 / 600 / 0.625	94 / 600 / 27	94 / 600 / 67	72 / 600 / 450

(a) 1000 States, transition density 1.6

	L=0	L=4	L=8
R=true, det=true, K=true	0 / NaN / NaN	8 / 123 / 35.5	150 / 139 / 54.6
R=true, det=false, K=true	13 / 0.417 / 0.669	13 / 0.461 / 20.2	16 / 127 / 49.2
R=true, det=false, K=null	160 / 140 / 0.71	152 / 140 / 38.3	7 / 124 / 46.8
R=false, K=false	15 / 0.544 / 0.163	19 / 0.555 / 19	166 / 1.39 / 53.6
R=false, K=null	311 / 1.41 / 0.684	307 / 1.41 / 38	160 / 1.42 / 55.7
RABIT=timeout, K=true	0 / NaN / NaN	0 / NaN / NaN	1 / 600 / 194
RABIT=timeout, K=false	0 / NaN / NaN	0 / NaN / NaN	0 / NaN / NaN
RABIT=timeout, K=null	1 / 600 / 4.82	1 / 600 / 185	0 / NaN / NaN

(b) 1000 States, transition density 1.8

	L=0	L=4	L=8
R=true, det=true, K=true	0 / NaN / NaN	234 / 138 / 34.1	236 / 138 / 55.1
R=true, det=false, K=true	16 / 0.368 / 0.693	10 / 133 / 33	11 / 136 / 53.2
R=true, det=false, K=null	231 / 139 / 0.713	3 / 138 / 35.1	0 / NaN / NaN
R=false, K=false	7 / 0.557 / 0.151	198 / 1.42 / 34.6	206 / 1.43 / 55.2
R=false, K=null	246 / 1.43 / 0.715	55 / 1.44 / 34.7	47 / 1.44 / 55.5
RABIT=timeout, K=true	0 / NaN / NaN	0 / NaN / NaN	0 / NaN / NaN
RABIT=timeout, K=false	0 / NaN / NaN	0 / NaN / NaN	0 / NaN / NaN
RABIT=timeout, K=null	0 / NaN / NaN	0 / NaN / NaN	0 / NaN / NaN

(c) 1000 States, transition density 2.0

Figure 5.3: More timing results for larger automata

	Count	Minimum Time	Maximum Time	Median Time	Mean Time
K=true, trivial=false	8	246s	1628s	479s	649s
K=true, trivial=true	10	241s	689s	475s	487s
K=null	69	380s	3017s	629s	768s

Figure 5.4: test

Figure 5.5: Results for lookahead 12 on the 87 automata pairs on which RABIT timed out at 30 minutes. The value of “trivial” refers to whether  $B$  was minimised to a trivial automaton.

RABIT times out in 94 experiments here. With lookaheads of 8 and lower, Kurshan fails to solve all of these cases. Kurshan with lookahead 12, however, witnesses inclusion for 22 pairs! The median time taken by Kurshan in these experiments is just over 5 minutes - half of RABIT’s assigned timeout. This might provide insight into the ease of witnessing inclusion - by RABIT’s pre-existing methods - for automata generated by these parameters. It appeared at first that inclusion only held in a few cases, and in those cases it was trivial to solve. (See above.) Yet upon the introduction of lookahead 12 we can see that inclusion in fact holds for at least 22 of the timed out cases - and likely more. Proving inclusion seems to tend towards either exceedingly easy or exceedingly difficult.

It’s possible that inclusion holds for (almost) all of these timed out cases. As can be seen across all of the results tables, proving non-inclusion is typically far less computationally expensive than proving inclusion. To prove non-inclusion, an algorithm need only find an input word that is accepted by  $A$  yet not accepted by  $B$ . To prove inclusion, on the other hand, an algorithm must prove that *every single* input word accepted by  $A$  is also accepted by  $B$ . So it would make sense for the difficult cases to be mostly inclusion-negative. Of the cases where RABIT does prove non-inclusion, the longest execution time was 379 seconds; this is close enough to the timeout of 600 seconds that we *could* expect some of the timed out cases to be on pairs for which inclusion does not hold.

Concerned that these impressive results arose due to the RABIT timeout being set too low, I reran the experiments with a timeout of thirty minutes. This timeout was recommended by my supervisor, as anecdotal evidence suggests that if RABIT does not compute a result in 30 minutes for this size of automaton, then it is unlikely to ever compute a result in any reasonable time. These results are presented in Figure 5.5.

There are 7 fewer timeouts under the increased timeout, but still substantially many at 87. Kurshan’s algorithm with lookahead 12 solves 18 of these, with even the longest of these execution times not exceeding RABIT’s timeout of 30 minutes. (Kurshan’s algorithm was not run with a timeout.) 10 of these true answers followed  $B$  being minimised to a trivial automaton, i.e. an automaton that accepts either all words or no words, prior to calculation of  $\tilde{B}$ . Had RABIT “tried harder” to minimise  $B$  down to such a trivial automaton before attempting the complete-yet-expensive algorithms, it may not have stalled, so it is possible that many of the pairs for which RABIT stalled and Kurshan returned true may not quite have required Kurshan’s algorithm to avoid timeout. I shall analyse this properly in Section 5.4.

After not-so-impressive results from Kurshan on the 100 state automata, this set of 1000 state automata demonstrates that Kurshan certainly *can* outperform RABIT’s pre-existing methods

as they are. Kurshan is not effective at witnessing non-inclusion for these automata, and it does still struggle to witness inclusion - with many pairs remaining unsolved by either method - but for witnessing inclusion it ostensibly performs better than RABIT.

#### **5.1.4.2 Transition Density 1.8**

These results are less noteworthy than the preceding subsection's. We see similar trends similar to those observed in Subsubsection 5.1.3.2. It's worth noting, that, in contrast to the 100 state automata, we barely see a difference in the sizes of the partitions when increasing the lookahead from 0 to 4. For transition density 1.6 (above) we might say that some "threshold" was passed regarding the lookahead, somewhere between 8 and 12, at which point we get significant decrease in the number of null results. We can see something similar here, in that there is barely a difference between lookahead 0 and 4 and yet the improvement between 4 and 8 is markedly better. Increasing beyond this threshold barely sees a reduction in null results. This was not observed in the 100 state automata.

#### **5.1.4.3 Transition Density 2.0**

Again we can see a "threshold" passed, this time between lookaheads 0 and 4, at which point we see a drastic decrease in the number of null results. Increasing the lookahead beyond this threshold sees further improvement but returns are diminishing and, of course, the execution time is greater for higher lookaheads.

#### **5.1.4.4 General Conclusions on 1000 State Automata**

The most notable results from the experiments on these larger automata are those from the automata with transition density 1.6. Kurshan solving so many pairs on which RABIT timed out tells us that this algorithm could have a use case. On large automata with low transition densities, we still can't always expect an answer from Kurshan's algorithm - it is not complete - but we can expect it not to stall (unlike RABIT) and expect it to solve some inclusion-positive pairs on which RABIT would take an extremely long time.

A pattern on these larger automata that was not present in the 100 state automata is that, for different transition densities, there is some "threshold" that the lookahead must pass to reduce the frequency of null answers from Kurshan. After passing this threshold, any further increase in lookahead has diminishing returns.

Like the 100 state automata, Kurshan outperforms RABIT (in terms of execution time) on inclusion-positive pairs for transition densities 1.8 and 2.0, though by a smaller margin.

The combined algorithm proposed in Subsubsection 5.1.3.2 would generally not be useful on these larger automata, as the time saved in the inclusion-positive cases is far less drastic than for the 100 state automata. As we will see in the following section, the execution time of my implementation is dominated by the time to minimise the inputs using RABIT's pre-existing minimisation methods. Thus any issues with scaling up to larger automata cannot lie in my specific implementation of Kurshan's algorithm.

## 5.2 Execution Time of Each Stage of the Algorithm

For each of the experiments described in Subsections 5.1.3 and 5.1.4, while I presented the sum total execution time of all stages in Kurshan’s algorithm, the actual timing data of Kurshan’s algorithm was split into the following stages:

- Time to minimise  $B$
- Time to calculate the quasi-complement,  $\tilde{B}$
- Time to check if  $A \cap \tilde{B}$  is empty
- Time to check if  $B$ , after minimisation, was deterministic

All percentages given in this section refer to the *mean* percentage of time taken, so that I can refer to the proportion of time taken by one stage compared to another.

The recurring pattern across all experiments - regardless of automaton size, automaton transition density, and minimisation lookahead - is that the time to minimise the input automata dominates the execution time of Kurshan’s algorithm. Even with just lookahead 1, in all experimental sets of automata, minimisation took over 95% of the total execution time. With higher lookaheads, this percentage is even greater. Lookahead 12 sees over 99% of its execution time spent on minimisation.

It is for this reason that, for this implementation of Kurshan’s algorithm, only  $A$  and  $B$  are minimised, rather than additionally minimising  $\tilde{B}$  after computing it. This second stage of minimising would likely increase execution time. It would be worth investigating variations of the algorithm that don’t minimise  $A$  and minimise only  $B$  for the purpose of potentially making it deterministic and thus eliminating the possibility of receiving a null output. This could cut execution time, but it could not be decreased to more than half of the current time, as we would still be performing minimisation on the other input automaton  $B$  of the same size.

With lookahead 0 the construction of the quasi-complement  $\tilde{B}$  is generally the most expensive stage of the algorithm, taking over 85% of the computation time in all experiments. There are, however, too many null answers for this variant of the algorithm to be practical on input automata with the amount of non-determinism seen earlier.

## 5.3 Performance on Previously Known “Nasty” Automata

A set of 155 known “difficult” Büchi automaton pairs were provided by my supervisor. Of these, 111 pairs have been solved for inclusion by other methods. My implementation of Kurshan’s algorithm, with minimisation lookahead 12, solved 11 of the 111 previously-solved pairs. It solved none of the unsolved pairs. From this small experiment, it suggests that Kurshan’s algorithm, while fast in some cases, may not provide answers to currently unknown inclusion questions for non-deterministic automata. But as seen in Subsubsection 5.1.4.1, we have found randomly generated automaton pairs on which only Kurshan, so far, has returned an answer - but this could change were the timeout on `inclusion_Buchi` increased or removed, as `inclusion_Buchi` is complete.

## 5.4 Integration Into RABIT

### 5.4.1 Motivation

After the positive-looking results of Subsubsection 5.1.4.1, I was interested to see how well Kurshan's algorithm could perform when integrated into the `inclusion_Buchi` method already included in RABIT.

The primary method for checking inclusion in RABIT, `inclusionBuchi`, makes use of several techniques - trying one after another in increasing order of typical execution time, minimising the two automata along the way - until an answer is returned. See Figure 5.6. As we saw in Subsubsection 5.1.4.1, there are classes of automata for which Kurshan's algorithm may return a non-null answer long before RABIT would. Given that (for all experiments presented above) Kurshan's algorithm never stalled, it could be worth it for RABIT to run Kurshan's algorithm before calling the complete Ramsey procedure [Abd+11] for inclusion checking.<sup>2</sup> If RABIT ever reaches this stage in its checks for inclusion, the input automata have already been heavily minimised, and thus, as we know from Section 5.2, much of the heavy lifting required to employ Kurshan's algorithm effectively has already been done. The construction of quasi-complement  $\tilde{B}$  and checking the emptiness of the intersection is so computationally inexpensive compared to the complete Ramsey procedure and the minimisation that it would be remiss not to see if Kurshan's algorithm can give an answer. If it returns null, then the complete Ramsey procedure can be called as normal.

I thus decided to write a modified version of `inclusion_Buchi`, which attempts Kurshan's algorithm (without any pre-Kurshan minimisation, since `inclusion_Buchi` already minimises) before calling the complete complete Ramsey procedure.<sup>3</sup> See Subfigure 5.7a.

### 5.4.2 Results

The modified algorithm was run on the same set of automata from Subsubsection 5.1.4.1. Kurshan is called on 91 of the pairs by this modified `inclusion_Buchi`. Kurshan was called on all but one of the pairs which stalled at 30 minutes in Subsubsection 5.1.4.1, and was called only 5 times on pairs which did not stall in Subsubseciton 5.1.4.1. (Inclusion holds for none of these pairs, but Kurshan returned null for each of them in the modified `inclusion_Buchi`.) So inserting Kurshan's algorithm before the calling of the complete Ramsey procedure was a sensible choice.

Results for this first experiment are presented in Subfigure 5.8a. See that Kurshan only ever returned null! The time taken by Kurshan to return null is so small as to be insignificant, at least, but from this set of input automata the addition of Kurshan appears useless. I knew from Subsubsection 5.1.4.1 that Kurshan's algorithm *can* return true for some of these automata, after enough minimisation, so I modified `inclusion_Buchi` further (Subfigure 5.7b) to allow

---

<sup>2</sup>The complete Ramsey procedure can be seen as a last last ditch-effort - a final resort. It is a complete, but computationally expensive, method of checking for inclusion.

<sup>3</sup>The modified version actually runs Kurshan's algorithm *instead of* the Ramsey procedure. We don't really care what happens beyond this point. We just want to know how often Kurshan's algorithm is called and how often it can actually produce an answer. An actual implementation deployed to RABIT would call the complete Ramsey procedure after Kurshan if Kurshan could not produce an answer.

```

...
...
...
if (x.result) {
  if (Options.verbose)
    System.out.println("Included_(already_proven_during_preprocessing)");
  return true;
} else {
  Options.globalstop = false;
  FairsimThread fst = null;
  if (Options.jumping_fairsim) {
    fst = new FairsimThread(aut1, aut2);
    fst.start();
  }
  // — attempt Kurshan's algorithm here! —
  // Start Ramsey procedure with limit==0, which means no limit.
  InclusionOptBVLayered inclusion = new InclusionOptBVLayered(aut1, aut2,
    0);
  inclusion.run();
  if (Options.jumping_fairsim)
    fst.stop();
  if (Options.verbose)
    System.out.println("Metagraphs_added_to_the_next_set:" + inclusion.
      mggen);
  if (inclusion.isIncluded())
    return true;
  else {
    if (Options.verbose)
      System.out.println("Counterexample:" + inclusion.
        counterexample_prefix + "(" + inclusion.counterexample_loop + ")");
    return false;
  }
}
}
}

```

Figure 5.6: An excerpt of final lines of the `inclusion_Buchi` method, which takes as input two finite automata `aut1` and `aut2` and returns true if  $\mathcal{L}(\text{aut1}) \subseteq \mathcal{L}(\text{aut2})$ , and false otherwise. A comment has been added where Kurshan's algorithm could be inserted as a second-to-last attempt to prove inclusion or non-inclusion. The augmented `inclusion_Buchi` method would move on to the subsequent complete Ramsey procedure, which is where RABIT tends to stall, only if Kurshan's algorithm returned null. Otherwise it would return the answer returned by Kurshan.

```

execute unmodified pre-Ramsey procedure code as normal
if answer not found:
    call Kurshan's algorithm

```

(a) The first modification

```

execute unmodified pre-Ramsey procedure code as normal
if answer not found:
    minimise both automata with lookahead 12
    call Kurshan's algorithm

```

(b) The second modification

```

execute unmodified pre-Ramsey procedure code as normal
if answer not found:
    minimise both automata with lookahead 12
    call Kurshan's algorithm
    call complete Ramsey procedure

```

(c) The third modification

Figure 5.7: The three modifications to `inclusion_Buchi`

	K=true	K=null
# of Occurrences	0	91
Median pre-Kurshan time	NaN	335s
Median Kurshan time	NaN	0.696s

(a) No additional minimisation

	K=true	K=null
# of Occurrences	17	74
Median pre-Kurshan time	326s	325s
Median Kurshan time (including minimisation)	351s	531s

(b) Lookahead 12 minimisation before calling Kurshan's algorithm

Figure 5.8: Results from the modified version of `inclusion_Buchi` that attempts Kurshan's algorithm before calling the complete Ramsey procedure. Kurshan is called on 91 of the automaton pairs from Subsubsection 5.1.4.1.



for additional minimisation of  $A$  and  $B$  just before calling Kurshan’s algorithm. Lookahead 12 was used, as this was found to be the most effective lookahead in Subsubsection 5.1.4.1. A second experiment with this additional lookahead was run, and results are presented in 5.8b, where we can see that Kurshan’s algorithm returned true in 17 of its 91 calls.

From Figure 5.8 we can gather that (for this set of automata) there is little point in using Kurshan’s algorithm in this way if we do not perform further minimisation before calling it. The additional minimisation does take quite some time (generally about as much time as all preceding attempts to determine inclusion by `inclusion_Buchi`) but if Kurshan fails to return an answer then the proceeding complete Ramsey procedure call will at least be less likely to stall on the now further minimised automata.

### 5.4.3 Lookahead 12 Minimisation Before Attempting Complete Ramsey Procedure

As so many of the  $B$  automata for which Kurshan returned a true answer in Subsubsection 5.1.4.1 were minimised to a trivial automaton, I wondered whether Kurshan’s algorithm was actually making a difference to prevent stalling 5.1.4.1, or if the complete Ramsey procedure could just have easily been used in place of Kurshan and benefited from the additional lookahead 12 minimisation all the same.

I made a third modified version of `inclusion_Buchi` (Subfigure 5.7c) which, after performing the additional lookahead 12 minimisation, executes Kurshan’s algorithm *and* the complete Ramsey procedure on the now smaller automata, regardless of the answers returned by Kurshan, for the sake of comparing how effectively they each get results.

I ran this modified method on the 17 RABIT-stalling automaton pairs for which Kurshan returned true in Subsection 5.4.2 to see if I would have been as well calling the complete Ramsey procedure in place of Kurshan’s algorithm. Results are presented in Figure 5.9.

We can see that the complete Ramsey procedure did not stall on any of these input pairs when passed the minimised automata. It returned true on each of them, as did Kurshan’s algorithm. While Kurshan’s algorithm did execute faster than the complete Ramsey procedure in all cases, the execution times of both techniques so small compared to the time to minimise that they are negligible.

We can thus conclude that previous positive-looking results for Kurshan’s algorithm were really just due to the strong minimisation, and the pre-existing `inclusion_Buchi` method could just as well have been modified to “try harder” in minimising the automata before passing them to the complete Ramsey procedure. This more vigorous minimisation has been shown in this section to reduce the likelihood of RABIT stalling.

From these results it might be more accurate to say that, on the 1000 state / 1.6 transition density automata, Kurshan “keeps up with” RABIT, rather than “outperforms” RABIT, at least in terms of returning results.

	pre-Ramsey time	Ramsey time	Kurshan Time	$B$ size before extra min	$B$ size after
1	630s	1.68s	0.155s	887	1
2	559s	1.58s	0.0986s	895	1
3	691s	0.280s	0.0958s	909	1
4	640s	1.30s	0.0943s	894	1
5	1142s	5.63s	1.23s	902	454
6	731s	1.73s	0.130s	895	1
7	770s	1.68s	0.130s	901	1
8	667s	2.78s	0.231s	910	2
9	936s	1.88s	0.117s	892	2
10	1118s	0.148s	0.114s	914	1
11	911s	0.008s	0.117s	904	2
12	861s	2.54s	0.100s	912	1
13	667s	0.195s	0.127s	888	3
14	883s	3.15s	0.236s	898	369
15	626s	1.85s	0.179s	897	1
16	658s	0.351s	0.105s	909	2
17	1483s	0.296s	0.114s	905	7
Mean	822s	1.593s	0.198s	900	50

Figure 5.9: Results for the third modified version of `inclusion_Buchi`, seen in Subfigure 5.7c

## 5.5 Conclusions: Applicability of Kurshan’s Algorithm

We have seen that Kurshan’s algorithm *does* work in practise even when  $B$  is non-deterministic, though minimising  $B$  to be “more” deterministic (or fully deterministic) does reduce the amount of null answers at the cost of greater execution time.

On all randomly generated automata sets tested, there are cases in which Kurshan’s algorithm outspeeds RABIT’s `inclusion_Buchi` method, but the time saving is small enough, and the frequency of null answers high enough, that there would be little reason to use Kurshan’s algorithm *instead of* `inclusion_Buchi`.

For small automata (around 100 states) there is little reason to use Kurshan’s algorithm at all. It could perhaps be used to prove inclusion-positive pairs more quickly than `inclusion_Buchi`, but we would still need to call `inclusion_Buchi` in cases where Kurshan returns null. For proving non-inclusion, Kurshan has no conceivable advantage over RABIT.

For 1000 state automata, we see results similar to the 100 state automata when generated with higher transition densities, but it is in the large automata with low transition densities on which Kurshan seemed to perform best. This is a class of automata on which `inclusion_Buchi` frequently stalls, and it is these stalling inputs that Kurshan, with prior lookahead 12, has been shown to sometimes produce an answer for - although it still returns null more often than not on the stalling automata. When we replaced Kurshan’s algorithm with the complete Ramsey procedure, however, we found that it was in fact the high lookahead value that avoided stalling, rather than Kurshan itself. For this reason I do not believe Kurshan would be very useful if integrated into `inclusion_Buchi`, but it executes so quickly that it would do little *harm*

to call Kurshan before calling the complete Ramsey procedure, and we might save time in *some* cases. If working in domains with mostly deterministic automata, we might see Kurshan outperform RABIT. The execution time of Kurshan's algorithm itself - omitting the preparatory minimisation - is very small, and it is guaranteed to return an answer if input automaton  $B$  is deterministic.



# Chapter 6

## Future Work

### 6.1 Development

I'm overall very satisfied with my implementation of Kurshan's algorithm, but it is not without its weak points.

- I discussed in Subsection 3.2.1 that Kurshan's requirement in [Kur87] that Büchi automata be lockup-free - in order for his algorithm to be correct - was overlooked in my early implementation, and thus when using certain constructors for `EdgeRecurringAutomaton` it is currently possible to instantiate an edge-recurring automaton that does not meet the lockup-free requirement. Future versions of the implementation might solve this by having, for example, the "manual" constructor throw errors if given an `EdgeMatrix` instance that is not lock-up free, or ensuring the invariant in some other way.
- I would have liked to capture the "edge-recurring to state-recurring" transformation in a constructor for the pre-existing `FiniteAutomaton` class in RABIT, so as to be symmetrical with the `EdgeRecurringAutomaton` constructor, but modification of pre-existing RABIT code was off-limits for this project, for the sake of isolating my own work. If my implementation were integrated into RABIT (see Sections 5.4 and 6.4) such a constructor could be implemented and the `toStateRecurring` instance of method of `EdgeRecurringAutomaton` removed.
- Similarly to how the intersection automaton is constructed dynamically while checking for its emptiness, additional time could be saved by constructing the quasi-complement dynamically as the intersection is constructed dynamically. When no additional minimisation is performed, construction of the quasi-complement was found to be the most expensive step in Section 5.2.

### 6.2 Experiments

As I had little prior experience with gathering data on programs such as this, there were some false-start experiments run, and some unusable data that did not make it into this dissertation. I also found it difficult to glean conclusions from much of the data, both due to inexperience

with analysing such data and the fact that RABIT and Büchi automata in general were entirely new to me. We have seen some useful results, but there is room for more.

We have concluded that there do exist classes of automata, generated by the Tabakov-Vardi method, on which Kurshan’s algorithm keeps up with the multi-strategy inclusion-checking method `inclusion_Buchi` already in the RABIT framework. In particular: automata of 1000 states and transition density 1.6. Other such classes of automata might be found by varying other parameters, such as alphabet size and acceptance density.

It would be wise to run future experiments on a dedicated machine in future. Shared servers such as `student.compute` were not ideal for performance-measuring experiments, as the resources allocated to me were unreliable due to experiments run by other students. Additionally, I did not feel that I could use the computing resources to their full capacity, as this would be unfair on other students. Running experiments on a dedicated machine was not an option for this dissertation project, as some of the experiments I ran took over a week in total to complete.

## 6.3 Theory

Knowing that Kurshan performs strongly on certain automata is well and good, but we do not yet have an understanding of *why* it performs so strongly on certain automata, such as those with lower transition densities. More theoretical insight into this phenomenon would better enable us to use Kurshan’s method to its full potential.

We saw in Subsubsection 5.1.4.4 that for larger automata, there is a “threshold” in minimisation lookahead past which Kurshan is far more effective, with diminishing returns beyond this. It would be good to find some theoretical basis behind this, so that methods can be developed that determine the best lookahead to use on a given automaton pair, taking into account execution time and likelihood of returning a non-null answer.

Not all non-deterministic Büchi automata have a deterministic equivalent, but many do. There might be a “best effort” algorithm, to try to convert automata, that is more effective at producing equivalent deterministic automata than the minimisation used in this project.

## 6.4 Integration Into RABIT

We saw in Section 5.4.2 that, looking at the pairs on which Kurshan returns an answer after RABIT’s unmodified `inclusion_Buchi` method stalled, the results could just have well have been achieved by passing the heavily-minimised automata to the complete Ramsey procedure rather than to Kurshan. So there is not as much reason integrate Kurshan into `inclusion_Buchi` as it first appeared, at least in the way trialed in Section 5.4. Kurshan’s algorithm is so inexpensive that would do no *harm* to integrate it into `inclusion_Buchi`, at least, and perhaps in some niche applications it might save time.

# Bibliography

- [Abd+11] Parosh Aziz Abdulla et al. “Advanced Ramsey-Based Büchi Automata Inclusion Testing”. In: *CONCUR 2011 - Concurrency Theory - 22nd International Conference, CONCUR 2011, Aachen, Germany, September 6-9, 2011. Proceedings*. 2011, pp. 187–202. DOI: 10.1007/978-3-642-23217-6\_13. URL: [https://doi.org/10.1007/978-3-642-23217-6%5C\\_13](https://doi.org/10.1007/978-3-642-23217-6%5C_13).
- [AHU74] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974, pp. 187–189. ISBN: 0-201-00029-6.
- [AHU83] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983, pp. 215–218. ISBN: 0-201-00023-7.
- [Cho74] Yaacov Choueka. “Theories of Automata on omega-Tapes: A Simplified Approach”. In: *J. Comput. Syst. Sci.* 8.2 (1974), pp. 117–141. DOI: 10.1016/S0022-0000(74)80051-6. URL: [https://doi.org/10.1016/S0022-0000\(74\)80051-6](https://doi.org/10.1016/S0022-0000(74)80051-6).
- [CM19] Lorenzo Clemente and Richard Mayr. “Efficient reduction of nondeterministic automata with application to language inclusion testing”. In: *Logical Methods in Computer Science* 15.1 (2019). DOI: 10.23638/LMCS-15(1:12)2019. URL: [https://doi.org/10.23638/LMCS-15\(1:12\)2019](https://doi.org/10.23638/LMCS-15(1:12)2019).
- [Edi] University of Edinburgh School of Informatics. *General purpose servers*. [Online; accessed 4-February-2020]. URL: <http://computing.help.inf.ed.ac.uk/general-purpose-servers>.
- [GTW02] Erich Grädel, Wolfgang Thomas, and Thomas Wilke, eds. *Automata, Logics, and Infinite Games: A Guide to Current Research [outcome of a Dagstuhl seminar, February 2001]*. Vol. 2500. Lecture Notes in Computer Science. Springer, 2002. ISBN: 3-540-00388-6. DOI: 10.1007/3-540-36387-4. URL: <https://doi.org/10.1007/3-540-36387-4>.
- [HHK96] R. H. Hardin, Z. Har’El, and R. P. Kurshan. “COSPAR”. In: *Computer Aided Verification*. Ed. by Rajeev Alur and Thomas A. Henzinger. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 423–427. ISBN: 978-3-540-68599-9.
- [HR04] Michael Huth and Mark Dermot Ryan. Cambridge University Press, 2004, pp. 232–328.
- [Ker10] Michael Kerrisk. No Starch Press, 2010, pp. 733–737. ISBN: 978-1-59327-220-3.
- [Kur87] Robert P. Kurshan. “Complementing Deterministic Büchi Automata in Polynomial Time”. In: *J. Comput. Syst. Sci.* 35.1 (1987), pp. 59–71. DOI: 10.1016/0022-0000(87)90036-5. URL: [https://doi.org/10.1016/0022-0000\(87\)90036-5](https://doi.org/10.1016/0022-0000(87)90036-5).

- [Lis87] Barbara Liskov. “Keynote Address - Data Abstraction and Hierarchy”. In: *Addendum to the Proceedings on Object-oriented Programming Systems, Languages and Applications (Addendum)*. OOPSLA ’87. Orlando, Florida, USA: ACM, 1987, pp. 17–34. ISBN: 0-89791-266-7. DOI: 10.1145/62138.62141. URL: <http://doi.acm.org/10.1145/62138.62141>.
- [May] Richard Mayr. *Tools*. [Online; accessed 8-January-2020]. URL: <http://www.languageinclusion.org/doku.php?id=tools>.
- [Tar72] Robert Endre Tarjan. “Depth-First Search and Linear Graph Algorithms”. In: *SIAM J. Comput.* 1.2 (1972), pp. 146–160. DOI: 10.1137/0201010. URL: <https://doi.org/10.1137/0201010>.
- [TV05] Deian Tabakov and Moshe Y. Vardi. “Experimental Evaluation of Classical Automata Constructions”. In: *Logic for Programming, Artificial Intelligence, and Reasoning*. Ed. by Geoff Sutcliffe and Andrei Voronkov. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 396–411. ISBN: 978-3-540-31650-3.
- [Wika] Contributors to Wikipedia. *Büchi automaton* — *Wikipedia, The Free Encyclopedia*. [Online; accessed 4-November-2019]. URL: [https://en.wikipedia.org/wiki/B%C3%BCchi\\_automaton](https://en.wikipedia.org/wiki/B%C3%BCchi_automaton).
- [Wikb] Contributors to Wikipedia. *Omega automaton* — *Wikipedia, The Free Encyclopedia*. [Online; accessed 20-January-2020]. URL: <https://en.wikipedia.org/wiki/%5C%CE%5C%A9-automaton>.