CEGAR for Regular Inclusion

Martin Köhler

19. Oktober 2016

Masterarbeit

Technische Universität Kaiserslautern
University of Kaiserslautern

Fachbereich Informatik
Department of Computer Science

Erstprüfer	Prof. Dr. Roland Meyer
Zweitprüfer	M.Sc. Florian Furbach

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Zusammenfassung

Die Sprachen endlicher Automaten sind abgeschlossen unter Schnitt, Vereinigung und Komplement. Da diese Operationen effektiv berechenbar sind, ist es konzeptionell auch möglich, die Schnittmenge aus vielen regulären Sprachen zu bilden und zu prüfen, ob diese in einer weiteren regulären Sprache enthalten ist. Geprüft wird die Leerheit des Schnitts aus dem Komplement der weiteren regulären Sprache und den zu schneidenden Sprachen.

Es ist jedoch nicht praktikabel, große Instanzen solcher Probleme explizit zu berechnen. Das liegt daran, dass der Automat der Schnittmenge exponentiell mit der Anzahl der Automaten wächst. Zeit und Speicherbedarf machen die Berechnung auf physisch realisierbaren Computern unmöglich.

Diese Arbeit untersucht und entwickelt einen CEGAR-artigen Ansatz (counterexample-guided abstraction refinement) zur Lösung des Problems. Gestützt werden die theoretischen Resultate durch eine praktische Implementierung des gewonnenen Verfahrens. Die grundlegende Idee ist die Bestimmung einer Menge von potenziellen Gegenbeispielen. Diese ist eine Obermenge der echten Gegenbeispiele. Die potenziellen Gegenbeispiele werden dann auf Echtheit geprüft. Die geschieht in der Absicht, die Inklusion zu zeigen oder zu widerlegen. Die Abwesenheit von tatsächlichen Gegenbeispielen entspricht der gültigen Inklusion.

Abstract

The languages of finite automata are closed under intersection, union, and complement. Since these operations are effectively computable, it is conceptually possible to intersect many regular languages and check the inclusion of this intersection in another language. This is done by checking the intersection of the complement of the other language and the languages that are intersected for emptiness.

Practically, it is not feasible to process larger instances of such problems. The reason is the size of the intersection automaton that grows exponentially in the number of automata that are intersected. The time and space consumption render the computation on physically constructible computers impossible.

This work examines and develops a CEGAR-like approach (counterexample-guided abstraction refinement) to address these issues. The theoretical results are supported by a practical implementation. The basic idea is the construction of a set of potential counterexamples. This set over-approximates the set of actual counterexamples. The potential counterexamples are then checked for spuriousness. The goal is to prove existence or absence of actual counterexamples. The absence of actual counterexamples is equivalent to a proven inclusion.

Contents

1 Introduction

The class of languages that can be represented by finite automata is the class of regular languages. A useful property of this class is its closure under intersection, union, and complement. These operations are effectively computable. Conceptually, this allows to intersect many regular languages and check whether they are included in another regular language. We refer to this possible superset as the specification. This naming is justified by practical applications:

One can represent the control-flow (or its approximation) of a thread in concurrent systems as a finite automaton. We can use such automata to describe the interleaving behavior. The automata allow arbitrary sequences of letters of the other threads' automata at any time. The intersection of these automata's languages represents the interleaved system behavior (or approximates it). We want to know whether all possible behaviors satisfy a certain specification, which is given as a regular language. Formally, our goal is to check whether the intersection is included in the specification language.

However, the conventional approach has practical limitations. If we compute an automaton describing the intersection explicitly, the state space grows exponentially with the number of intersected automata. If we intersect eight automata with ten states each, we get a state space of 100 million states. Hence, the practically solvable instances of the problem are restricted to rather small input sizes.

The application is a fundamental problem of multi-threaded verification. Faster solutions for instances of this problem extent the capabilities of the said verification technique. Such solutions allow to practically verify larger programs. Hence, we contribute to verification by tackling the language-theoretic problem.

Another language-theoretic problem and its application in verification has been discussed in [LCMM12]. The authors tackle the undecidable emptiness problem for the intersection of context-free languages. To this end, they introduce a language-theoretic variant of counterexample-guided abstraction refinement (CEGAR). This verification technique was introduced in [CGJ$^+$00]. The basic idea of CEGAR is to find violations of the specification by searching for executions over an abstract domain. These executions are called counterexamples. Such executions may or may not be possible using the actual data domain of the program. The counterexamples are hence checked for spuriousness. If the discovered counterexamples are spurious, the model is refined in order not to show these spurious counterexamples again.

The language-theoretic CEGAR-approach for emptiness of context-free languages over-approximates one of the context-free languages by a regular language. The intersection between a context-free and a regular language is effectively computable. However, the resulting counterexample language might be spuriously non-empty. To this end, a spuriousness check is performed. It checks elementary bounded subsets of the counterexample language for non-spurious counterexamples. This is done by deciding whether they have a non-empty intersection with both context-free languages. If the Elementary Bounded Language was entirely spurious, it will be excluded from the regular over-approximation. Since the said problem is undecidable, the algorithm might refine infinitely often without ever reaching an empty set of counterexamples.

The approach for regular inclusion is quite similar: It over-approximates the regular intersection. The counterexample language consists of the words in the over-approximation that disobey the specification. Counterexamples are also generalized as Elementary Bounded Languages. They are then checked for words that are in all of the input languages. The counterexample language is refined in the negative case.

In contrast to the original work [LCMM12] on language-theoretic abstraction refinement, our approach considers a decidable problem. The original work aimed to solve the generally undecidable problem for certain instances. Our work's contribution is the acceleration of the solution for some instances of our problem. Conventional solutions for these instances consume more resources than available. Hence, our procedure increases the range of practically computable instances. Our solution does neither aim to be a decider nor a semi-decider for the general case: It might try to refine infinitely often without ruling out all spurious counterexamples and without finding an actual counterexample. More details on this property and variants of this procedure that are deciders for a certain subclass are given in Section 10. In that Section, we also suggest CEGAr (less refinement) and CEGaR (less abstraction). These variants of our procedure do not implement the classical CEGAR approach. They allow to semi-decide the problem since they can enumerate all possible languages of counterexamples. The restriction to be neither a decider nor a semi-decider was not a big issue for randomly generated test instances. Our implementation found exact solutions notably often: Only about 5% of the finished executions gave up without proving or disproving the claim (see also Section 9).

Our implementation can be used as a component of a decider: If our tool cannot find a solution, it can be ran again with a less coarse over-approximation. The counterexample might then be found in a repeated run of the automaton. In the worst case, the over-approximation is tightened until our procedure computes the exact intersection. Although this variant might fail due to practically infeasible instances, it is conceptually a decider.

Our work is structured as follows: In Section 2, we first give an overview on the fundamentals and the basic concepts that are used in this work. The theoretical part introduces the procedure to over-approximate the intersection. This is covered in Section 3. We then introduce the spuriousness check. Section 4 presents an overview how counterexamples are extracted and how their spuriousness is checked. Technical details how we generalize a single counterexample to a language of counterexamples are presented in Section 5. The actual counterexample check requires a preparation of the input automata. This is covered in Section 6. After this preparation, a Presburger Formula can be generated. This formula is satisfiable if and only if there is a non-spurious counterexample in the given Elementary Bounded Language of potential counterexamples. The procedure that extracts this formula is introduced in Section 7.

We support these theoretic results with an implementation of the procedure. Section 8 presents the modules of our implementation, the intended use, and details on part of the implementation. It also includes a tutorial on the usage and integration of our tool. Section 9 presents the results of the benchmarks we performed on our tool. We conclude the work and give an overview on possible variants of the procedure in Section 10.

2 Fundamentals

The chosen approach uses basic concepts of automata theory, formal languages, and logics. We now clarify our notation for finite automata and runs. Additional concepts will be introduced in Section 4.2 when we first use them in the context of the fast intersection check. These concepts are:

Elementary Bounded Languages Our procedure generalizes counterexamples as Elementary Bounded Languages. They satisfy a requirement that is needed for our fast intersection checks.

Parikh Images Our fast intersection check does not rely on the explicit computation of the intersection. We rather project the input languages to a domain where the (non-)emptiness intersection problem is in $\mathcal{NP}$. Parikh Images are one of the two projections we compose in order to project the languages to this domain.

Presburger Formulas Presburger Arithmetic is a certain fragment of first order predicate logic. We use Presburger Formulas to extract, represent, and intersect Parikh Images.

Notation of Finite Automata There are various notations on finite automata. In this work, we use this notation:

Definition 1 (Finite Automata) Let Q be a finite set of *states*, $q_0 \in Q$ the *initial state*, $\rightarrow \subseteq Q \times \Sigma \times Q$ a set of *transitions*, and $Q_F \subseteq Q$ be a set of *accepting states.* These components form the *automaton* $A = (Q, q_0, \rightarrow, Q_F)$ over the alphabet Σ. $\square$

Whenever we refer to automata, we usually mean non-deterministic automata. We use the same notation for deterministic finite automata. We just require that there is at most one outgoing transition for each combination of source state and letter, i.e. for all $q \in Q, a \in \Sigma : |\{(q, a, q') \in \rightarrow \mid q' \in Q\}| \leq 1$.

Formally, a transition is a three-tuple of $Q \times \Sigma \times Q$. We use the syntax $q \xrightarrow{a} q'$ to express: There is a transition $(q, a, q') \in \rightarrow$.

Definition 2 (Runs of Automata) Let $A = (Q, q_0, \rightarrow, Q_F)$ be an automaton. We call a sequence $q_0 \xrightarrow{a_0} q_1 \xrightarrow{a_2} \ldots \xrightarrow{a_{n-1}} q_n$ *a run of the automaton A (to q_n) (reading the word $a_0 \ldots a_{n-1}$)* if $q_0, \ldots, q_n \in Q$ and $(q_0, a_0, q_1), \ldots, (q_{n-1}, a_{n-1}, a_n) \in \rightarrow$. A run that ends with an accepting state $q_n \in Q_F$ is an *accepting run.* A run with a first state q_k that is different from the initial state is called *a run from q_k to q_n*. $\square$

If there is an accepting run reading a word w, the automaton is said to *accept* the word w. The set of accepted words form the *language* $\mathcal{L}(A)$ of the automaton A.

Regular languages are closed under union, intersection, and complement. If the languages are represented by automata, automata accepting the result of these operations can be obtained. The techniques include crossproduct automata (for intersection and deterministic union) and power set automata (for determinization and complement). When we say that we apply language-theoretic operations on automata, we mean that we apply the underlying techniques on the automata in order to obtain an automaton that accepts the resulting language of the operation.

3 Over-Approximating Regular Intersections

As already mentioned in the introduction, the exact computation of an intersection of many regular languages is practically infeasible. The reason is the size of state space of the cross-product automaton, which grows exponentially in the number of automata: Let $Reg_1, \ldots, Reg_k$ be finite automata with up to n states each. The cross product automaton accepts the intersection $\bigcap_{i=1}^{k} \mathcal{L}(Reg_i)$. States of the cross-product automaton are tuples of the form $(q_1, \ldots, q_k)$ where each component q_i is a state of Reg_i. Hence, it contains up to n^k states. The automaton accepts in precisely those states where each component is accepting.

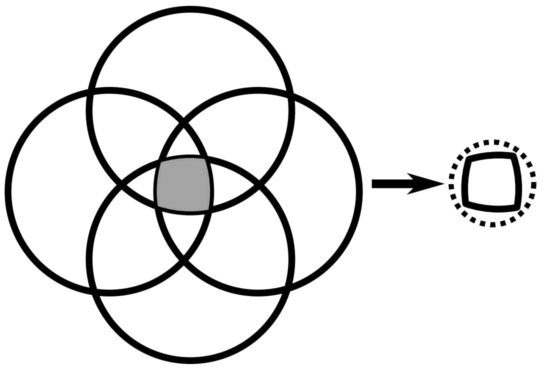

Figure 1: The goal of the over-approximation: Obtain a language (dotted circle) that contains the actual intersection (grey area).

We present a procedure that computes an over-approximation of the intersection: The resulting automaton has significantly less states than the cross-product automaton while accepting a superset $\mathcal{L}$ of the intersection $\bigcap_{i=1}^{k} \mathcal{L}(Reg_i)$. Figure 1 illustrates the approach. Furthermore, during the computation of the automaton, a complete exploration of the cross-product automaton is practically infeasible. Thus, our procedure over-approximates the cross-product automaton before its whole state space is explored. This is achieved in the following way: The procedure intersects some of the automata explicitly. The result is then over-approximated by an automaton with less states. The over-approximates automaton can then be intersected with input automata or intersections until all input languages will have been considered.

The advantage of this approach can be seen by comparing the result to an *on-the-fly* approach, which partially explores the state space of the whole cross-product automaton and over-approximates the partial automaton.

The difference between the two procedures is that the approximation on the fly certainly considers states that are found early – in graph theoretic terms: With a low distance from the initial state – more frequently than states that are explored later: Whenever the state space is reduced by over-approximating the explored states, the known part of the automaton is considered for the over-approximation. This part will still be a known part for later over-approximations. Thus it will be considered again while freshly explored parts of the automaton will be considered for the first time. Furthermore, states and transitions that are discovered at a later point in the on-the-fly approximation are already biased by the decisions that were made during earlier over-approximations. Our variant treats all states equally as the approximation takes place after all states of the partial intersection are known. Thus, the focus of this work is on the approximation after partial intersection.

In this section, we will first present the over-approximation of explicitly computed automata. We will explain the construction and properties of this procedure. After that, we will discuss how to combine partial intersections and over-approximations. We do so in order to obtain a procedure for the intersection of all automata. We will also discuss how knowledge about emptiness of intersections is preserved throughout the procedure and where we find bottlenecks.

3.1 Over-Approximating a Partial Intersection

This step is applied after an intersection was computed. The purpose is to prepare an automaton to be an input for the next intersection step. We are given an automaton $A = (Q, q_0, \rightarrow, Q_F)$ and want to construct an automaton $A' = (Q', q'_0, \rightarrow', Q'_F)$ with $|Q'| < |Q|$ states that accepts at least the original language $\mathcal{L}(A) \subseteq \mathcal{L}(A')$. We will later see how the whole procedure of stepwise over-approximating the intersection of the input automata keeps the properties of this *over-approximation*: Intersecting the smaller automaton A' with others will lead to smaller automata than doing the same with A.

Uniting States The core mechanism that we use in order to over-approximate a language unites two states to a single one that inherits the transitions and acceptance behavior of both original states.

Definition 3 (Uniting States) Let $A = (Q, q_0, \rightarrow, Q_F)$ be an automaton containing at least the states $q_1, q_2 \in Q$, $q_1 \neq q_2$. *Uniting the states* q_1 *and* q_2 *to* q' yields the automaton $A[(q_1, q_2); q'] := (\{q'\} \cup Q \setminus \{q_1, q_2\}, q'_0, \rightarrow', Q'_F)$ where

$$q'_0 := \begin{cases} q' & \text{if } q_0 = q_1 \text{ or } q_0 = q_2 \\ q_0 & \text{otherwise} \end{cases}$$

$$\rightarrow' := \left\{ (q'_i, a, q'_j) \text{ for each } q_i \xrightarrow{a} q_j \right\}$$

$$\text{with } q'_i = q' \text{ for } q_i \in \{q_1, q_2\} \text{ and } q'_i = q_i \text{ otherwise, } q'_j \text{ accordingly}$$

$$Q'_F := \begin{cases} \{q'\} \cup Q_F \setminus \{q_1, q_2\} & \text{if } \{q_1, q_2\} \cap Q_F \neq \emptyset \\ Q_F & \text{otherwise} \end{cases}$$

Example 1 (An Automaton for $\{ab\}$) Figure 2 shows the minimal automaton that accepts the language $\{ab\}$ and the various options of uniting states. Uniting the initial state q_0 and its direct neighbor state q_1 results in an automaton that accepts a language described by the regular expression a^*b. The reason is that the new state q_0, q_1 inherits the transition $q_0 \xrightarrow{a} q_1$ as a self-loop and $q_1 \xrightarrow{b} q_2$ as an outgoing transition to the accepting state q_2. The new state is now the initial state as q_0 was the initial state before it was united with q_1. Similarly, uniting q_1 with q_2 yields a new state that accepts as q_2 accepted before. The automaton is equivalent to the regular expression ab^*. The automaton that is constructed by uniting q_0 and q_2 accepts the language that is described

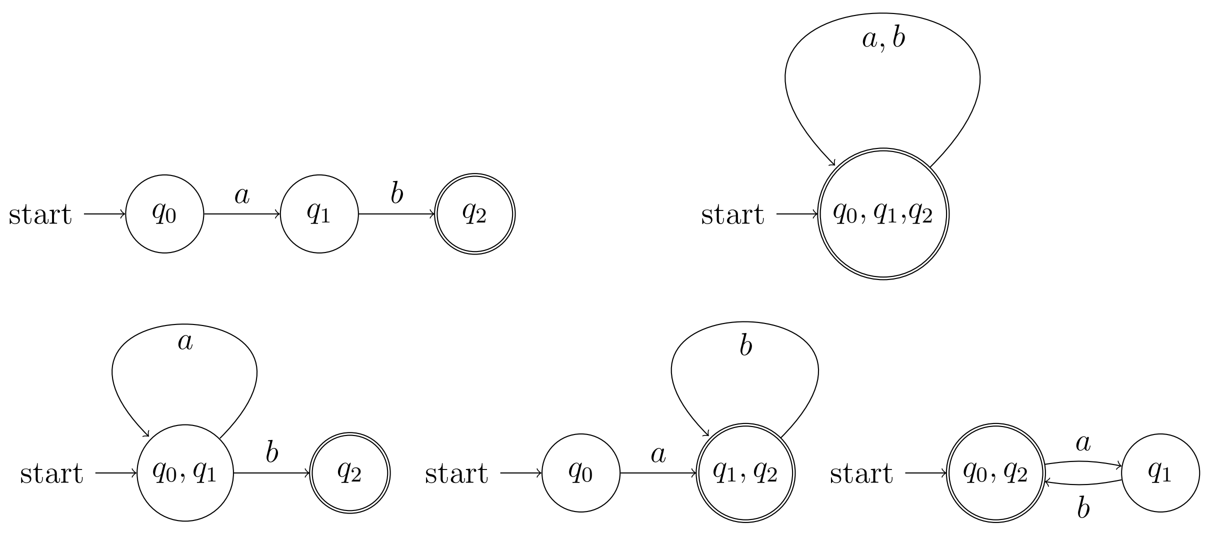

Figure 2: An automaton accepting $\{ab\}$ (upper left) and all possible results after uniting states. Lower row: The results after uniting q_0 with q_1, q_1 with q_2, and q_0 with q_2. Upper right: The result after uniting the remaining two states in any of the automata in the lower row.

by $(ab)^*$. Uniting the remaining two states of any of these three automata always creates the same single state automaton that accepts the language that is described by $\{a, b\}^*$.

In this case, uniting the first pair of states leads to a partial loss of information on the quantity of the characters: The first two automata only ensure that b (a respectively) occurs exactly once without a requirement on the number of occurrences of the other letter. The third automaton only ensures that a and b occur equally often without a requirement on the exact number. They also partially preserve information on the order of the characters: The first two automata in the second row ensure that a always occurs before b does. The third automaton in the second row ensures that each a is immediately followed by b. Interestingly, this is enough information about the structure of word to retrieve the original word ab by adding back exact information on the quantity: Restricting any of the three languages to words that contain a and b exactly once, leads back to the original language $\{ab\}$. Uniting another pair of states leads to a complete loss of information on the order.

Yet, the original word ab is still present in all four languages. The ability to completely restore the original language by adding information on the quantity can pay out throughout the procedure of over-approximating the whole intersection: While the intersection of some subset of the input automata loses information on the quantity after uniting states, the intersection of a different subset of the input automata might still carry this information. The lost information might be restored by intersecting these two intersections.

The number of states of an automaton decreases certainly by one when uniting two states in an automaton. In Example 1, all automata resulting from uniting states accepted a superset of the original language. In the following, we will show that uniting any pair of states in any automaton will generally lead to the acceptance of a superset of the original automaton's language.

Lemma 1 (Corresponding Runs After Uniting States) *Let $A = (Q, q_0, \rightarrow, Q_F)$ be an automaton, $q_1^u, q_2^u \in Q$ states of the automaton A, $q_0 \xrightarrow{a_0} \dots \xrightarrow{a_{n-1}} q_n$ a run of A and $A' = A\left[(q_1^u, q_2^u); q'\right] = (Q', q_0', \rightarrow', Q_F')$ the automaton after uniting the states q_1^u, q_2^u to $q' \in Q'$. Then $q_0' \overset{a_0}{\rightarrow} \cdots \overset{a_{n-1}}{\rightarrow} q_n'$ with*

$$q_i' = \begin{cases} q' & \text{if } q_i \in \{q_1^u, q_2^u\} \\ q_i & \text{otherwise} \end{cases} \quad \text{for all } 0 \leq i \leq n.$$

is a run of A'.

Proof We prove the claim by an induction on the length of the word that is read along the run. We need to show that the defined run is a valid run of A' and that each of its states q_i' is either the corresponding state q_i in the run of A or the substitute q' if the corresponding state was one of the two states that were united.

Base Case: Let q_0 be a run of A reading the word of length 0. Then, the corresponding run of A' needs to consist of only q_0', which is already defined to be the initial state of the automaton A'. The initial state of A' is either the former initial state q_0 of A or the new state q' if q_1 or q_2 is the initial state of A. Hence, the new initial state q_0' itself is a run of A' reading the empty word.

Induction Hypothesis: For a fixed length k, for every run $q_0 \xrightarrow{a_0} \dots \xrightarrow{a_{k-1}} q_k$ that exists in A, there exists a run $q_0' \xrightarrow{a_0} \dots \xrightarrow{a_{k-1}} q_k'$ in A' where $q_0', \dots, q_k'$ are chosen according to the definition above.

Induction Step: A run $q_0 \xrightarrow{a_0} \dots \xrightarrow{a_{k-1}} q_k \xrightarrow{a_k} q_{k+1}$ of A reading a word of length $k+1$ implies a run $q_0 \xrightarrow{a_0} \dots \xrightarrow{a_{k-1}} q_k$ of A reading a prefix of length k. As a result of the induction hypothesis, there is also a run $q_0' \xrightarrow{a_0} \dots \xrightarrow{a_{k-1}} q_k'$ in the automaton A'. As the run of A ends in $q_k \xrightarrow{a_k} q_{k+1}$, the automaton A contains a transition $(q_k, a_k, q_{k+1}) \in \rightarrow$. According to the definition of the transition relation after uniting two states, there exists a corresponding transition in $A\left[(q_1^u, q_2^u); q'\right]$ where q_k (q_{k+1}) is supplemented by q' if and only if q_k (q_{k+1}) is q_1^u or q_2^u. This means that the new transition leads from the end of the run of A' according to the induction hypothesis to the state q_{k+1}', which corresponds to q_{k+1} in A. Thus, the existing transition $(q_k', a_k, q_{k+1}') \in \rightarrow'$ of A' extends the run of A' to $q_0' \xrightarrow{a_0} \dots \xrightarrow{a_{k-1}} q_k' \dots \xrightarrow{a_k} q_{k+1}'$. ■

When two states are united, the resulting state inherits the acceptance behavior of the original states: It is a final state if at least one of the two original states accepted it. Thus, the previous lemma implies that a run remains accepting after uniting two states if it corresponds to an accepting run in the original automaton.

Corollary 1 *Let A be an automaton and A' be the automaton after uniting two of the states of A. If there is an accepting run of A reading a word w, then there is also an accepting run in the automaton A' reading the word w.* $\square$

Therefore, we know that uniting two states is a method to always obtain an over-approximation of the previous language:

Theorem 1 *Let A be an automaton that accepts $\mathcal{L}(A)$ and A' be the automaton after uniting two of the states of A. The automaton A' accepts at least the language $\mathcal{L}(A) \subseteq \mathcal{L}(A')$ of the original automaton.* $\square$

Selecting a Pair of States to Unite As shown before, we can unite an arbitrary pair of states, and the result is certainly an over-approximation of the input. We are left with the choice of the pairs of states to unite first. The choice decides how much and which kind of information about the language we lose. We now present a heuristic to select promising pairs of states.

The basic idea is to consider all pairs of states, rate the effect of uniting this pair and select the pair rated best. The central criterion for this rating is the similarity as we assume that uniting similar states does not add many new words. After uniting a pair of states, ratings involving the adjacent states need to be updated in order to choose the next pair. We now examine which words are added after uniting a pair of states. This will motivate the concrete measure for the effect of this operation.

Let $A = (Q, q_0, \rightarrow, Q_F)$ be an automaton and $q_1, q_2 \in Q$ be two states of A. Assume $\mathcal{L}^q(A) \subset \mathcal{L}(A)$ are the words that can be read along an accepting run containing the state q, $\mathcal{L}^q_{pre}(A)$ are the words that can be read by a run from the initial state to q, and $\mathcal{L}^q_{post}(A)$ are the words that can by read by an accepting run from q.

Uniting the states q_1 and q_2 adds new accepting runs in this way: The runs have a prefix up to the new state that was also a prefix of an accepting run through q_1 or q_2. They also have a suffix from the new state that was also a suffix from the other state. Thus, the new runs' words are $\bigcup_{(i,j)\in\{(1,2),(2,1)\}} \mathcal{L}^{q_i}_{pre}(A) \cdot \mathcal{L}^{q_j}_{post}(A)$. However, these words do not necessarily extend the language of the automaton: The words with a prefix $w \in \mathcal{L}^{q_1}_{pre}(A) \cap \mathcal{L}^{q_2}_{pre}(A)$ that can lead a run both to q_1 and q_2 had been accepted by the automaton before the states were united. The situation for postfixes is similar. This reduces the newly accepted words to

$$(i,j) \in \{(1,2),(2,1)\} \left(\mathcal{L}^{q_i}_{pre}(A) \setminus \mathcal{L}^{q_j}_{pre}(A)\right) \cdot \left(\mathcal{L}^{q_j}_{post}(A) \setminus \mathcal{L}^{q_i}_{post}(A)\right).$$

However, these words are only added if they had not been accepted by a run through states other than q_1 and q_2. Furthermore, this set of newly added words cannot directly be used as a measure for the effect of uniting a pair of states: The sets are usually infinite as there are in general infinitely many runs of an automaton. Tracing all possible pre- and suffixes takes too long to be done for each rating of a state pair.

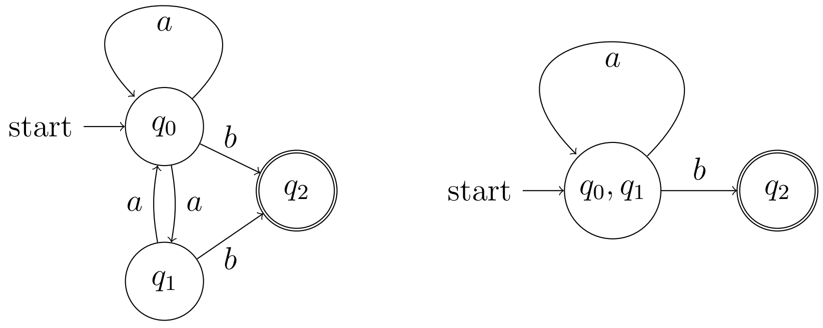

Figure 3: An example automaton. The similarity rating for q_0, q_1 is $(1, 1)$. The automaton on the right is the result after uniting the two states.

The phenomenon that uniting states adds words that are built crosswise from a prefix to one state and a suffix from the other state is the key motivation of our measure. Additionally, we assume that states with more adjacent states occur in more runs than states with less adjacent states. This assumption is not true in general but it is a simple heuristic as it only involves comparing the states' direct neighbors.

Hence, we define the similarity rating of a pair of states. It counts the number of incoming and outgoing transitions that are exclusive to one of the states:

Definition 4 (Similarity Rating) Let $A = (Q, q_0, \rightarrow, Q_F)$ be an automaton containing at least the states $q_1, q_2 \in Q$. The *similarity rating of* q_1 *and* q_2 is:

$$
\begin{aligned}
sim(q_1, q_2) &:= (sim_{in}(q_1, q_2), sim_{out}(q_1, q_2)) \\
&\quad\text{with} \\
sim_{in}(q_1, q_2) &:= \left| \bigcup_{i \in \{1,2\}} \Big(\{(q, a) \mid q \xrightarrow{a} q_i\} \Big) \setminus \Big(\{(q, a) \mid q \xrightarrow{a} q_1\} \cap \{(q, a) \mid q \xrightarrow{a} q_2\} \Big) \right|, \\
sim_{out}(q_1, q_2) &:= \left| \bigcup_{i \in \{1,2\}} \Big(\{(a, q) \mid q_i \xrightarrow{a} q\} \Big) \setminus \Big(\{(a, q) \mid q_1 \xrightarrow{a} q\} \cap \{(a, q) \mid q_2 \xrightarrow{a} q\} \Big) \right|.
\end{aligned}
$$

The rating assigns two numbers to each pair of states. An example and some properties of this measure will motivate why it is beneficial to count incoming and outgoing transitions separately and why exclusive edges are counted as defined above.

Example 2 (State Similarity Rating) Figure 3 shows an example automaton with two similar states: The initial state q_0 has an incoming transition from q_1 with letter a and a self-loop also with the letter a, that counts as an incoming transition from q_0. The only incoming transition of q_1 is the one from q_0 with the letter a. Thus, the self-loop at q_0 and the transition from q_0 to q_1 are counted as a common incoming transition from

q_0 with the letter a. The only exclusive incoming transition is the one from q_1 to q_0 with the letter a as q_1 has no incoming transition from q_1 and therefore none from this state reading the letter a. Hence, the incoming similarity rating is $sim_{in}(q_0, q_1) = 1$.

Similarly, the self-loop at q_0 and the transition from q_1 to q_0 count as a common outgoing transition to q_0 reading a. Also the transitions from q_0 and q_1 to q_2 with the letter b count as a common outgoing transition. The only exclusive transition is the one from q_0 to q_1 reading b as q_1 has no outgoing transition to itself. Hence, the outgoing similarity rating is $sim_{in}(q_0, q_1) = 1$. In total, there is only one exclusive outgoing and one exclusive incoming transition. Thus, the similarity rating is $sim(q_0, q_1) = (1, 1)$.

The second automaton is the result after uniting those two states. Interestingly, the new automaton also accepts exactly the language b^*a. Since the original language is not only included in the resulting language but even exactly the same language, the new automaton is not only an over-approximation of the old one but even an exact minimization. □

Although the previous example is indeed an exact minimization, this does not hold for all pairs of states with the similarity rating $(1, 1)$. A simple counterexample is a pair of a state with a unique incoming and a state with a unique outgoing transition. If those unique transitions read other letters than all other transitions, then uniting the pair of state adds a new word that contains both characters sequentially. For example, the states q_0 and q_2 of the initial automaton in Figure 2 have this property. Their similarity rating is $sim(q_0, q_2) = (1, 1)$ as the only transition of q_0 is outgoing and the only transition of q_2 is incoming. Yet, uniting these two states yields an automaton accepting $\{a, b\}^*$.

However, there are similarity ratings that imply that uniting the considered pair of states does not add new words to the language of the automaton: Similarity ratings with a component being 0 imply that the language will not change when uniting two states under certain conditions. These conditions ensure that there are no edge cases where uniting the two states adds accepting states to the end or initial states to the beginning of runs.

Lemma 2 (Similarity Ratings with 0 as a Component) *Let $A = (Q, q_0, \rightarrow, Q_F)$ be an automaton. Let $q_1, q_2 \in Q$ be two of the states of A. A rating with $sim_{in} = 0$ or $sim_{out} = 0$ implies $\mathcal{L}\left(A\left[(q_1, q_2); q'\right]\right) = \mathcal{L}\left(A\right)$ if none of the following holds:*

- *The incoming rating is $sim_{in}(q_1, q_2) \neq 0$ and $|\{q_1, q_2\} \cap Q_F| = 1$.*
- *The outgoing rating is $sim_{out}(q_1, q_2) \neq 0$ and $q_0 = q_1$ or $q_0 = q_2$.*

Proof We prove the statement by considering runs of accepting words in the new automaton after uniting the states and argue how we can be sure to find an accepting run for the same word in the original automaton. Since we are assuming that one of the components of the similarity rating is zero, we split our argument in two cases depending on the component. If both components are zero, both arguments will lead to an accepting run in the original automaton:

$sim_{in}(q_1, q_2) = 0$ The two states agree on the incoming transitions. This means that any non-empty word that leads from any state q to q_1 (q_2) can also lead to q_2 (q_1) by changing the choice in the last transition. If a word in the automaton after uniting the states is accepted by a run that includes the new state elsewhere than in the first position then it was already accepted by the old automaton: From the new state, the word is accepted directly, after taking a transition that belonged to q_1 or after a transition that belonged to q_2.
The original automaton shows the same behavior in all three cases: If the new state accepts the word, then q_1 and q_2 accepted the word. As they are reachable reading the same prefixes, the old automaton could choose a run going to the accepting state.
If the run continues after the new state by taking a transition that is exclusive to q_1 (q_2), the old automaton can choose to take a run to this state and continue the run from the exclusive transition.

There is a special case where q_1 (q_2) is the initial state of the original automaton A: Uniting q_1 and q_2 could possibly add new words that are accepted by a run that starts with a transition that is exclusive to q_2 (q_1). The original automaton required runs to have a prefix leading to q_2 (q_1). However, exclusive outgoing transitions to the non-initial state are not considered here as they fall under the second of the excluded cases.

$sim_{out}(q_1, q_2) = 0$ The two states agree on the outgoing transitions. Thus, every run through the new state can be reconstructed in the old automaton as a run to one of the two states as it can always be continued in the same way since the outgoing transitions exist for both states q_1 and q_2. If the new state is initial, then it was possibly not reached by a run to this state but the run started there. Since the new state is only initial if one of the two states q_1 or q_2 was initial, the old automaton can reconstruct this run by starting in the initial state.

There is a special case where q_1 (q_2) is accepting and the other one is not: If the new state is reached by a transition that was exclusive to the non-accepting state q_2 (q_1), then the new automaton can accept the word although the old automaton required a non-empty suffix to reach an accepting state. This can only happen if the two states q_1 and q_2 disagree both on incoming transitions and on the acceptance behavior. The first of the excluded cases rules out this situation.

If the accepting run contains the new state at more than one position, we can prove the statement by applying the argument in an inductive way: We always follow the run in the automaton until the next state is the new state. If the states agree on the incoming states, we have to choose q_1 or q_2 so that there is an outgoing transition to the successor of the new state in the run in the new automaton. If the states agree on the outgoing state, we just follow the available transition to q_1 or q_2. We repeat this until we reach the end of the run. ■

If we have more information on the state pair than just the similarity rating, we can strengthen the previous claim about cases when a similarity rating with a component being zero implies that uniting states is a minimization rather than an actual over-approximation:

$sim(q_1, q_2) = (0, 0)$ Always.

$sim_{out}(q_1, q_2) = 0$

1. The states agree on the acceptance behavior, i.e $|Q_F \cap \{q_1, q_2\}| \neq 1$
2. The non-accepting state does not have exclusive incoming edges, i.e.

$$(\{q_1, q_2\} \setminus Q_F) \cap \left\{ q \in \{q_1, q_2\} \mid \exists_{q' \in Q} \forall_{q'' \in \{q_1, q_2\} \setminus q} q' \xrightarrow{a} q \wedge q' \not\xrightarrow{a} q'' \right\} = \emptyset$$

$sim_{in}(q_1, q_2) = 0$

1. None of the two states is the initial states, i.e. $q_1 \neq q_0 \neq q_2$
2. Only the initial state has exclusive outgoing edges, i.e. $\{(a, q) \mid q_1 \xrightarrow{a} q\} \cup \{(a, q) \mid q_2 \xrightarrow{a} q\} \subseteq \{(a, q) \mid q_0 \xrightarrow{a} q\}$

Applying the Measure Previously, we have motivated the chosen measure and proven that it can even help finding state pairs that can be united without changing the language. Although this is not the main purpose of the measure, it still gives an impression that low numbers in one of the components indicate good candidates when uniting states. We will now explain how to select a state pair after obtaining the similarity measure for each possible pair of states.

The intuition of the measure is to rate the amount of new prefixes and suffixes. When uniting two states, we might add new words that start with a prefix leading to one of the two states and end with a suffix leading from the other state to an accepting state. The first component of the measure can be seen as a judgement for the number of new prefixes, the other one as judgement for the number of suffixes.

We decided to minimize by the product of two components as it reflects the way how prefixes that are exclusive to one of the states and suffixes that are exclusive to the other state form new words.

The overall procedure for the over-approximation of a partial intersection is hence as follows: Given a subset of possibly over-approximated automata, compute the actual intersection. Determine the current number of reachable states and the desired number of states. If the determined number of states exceeds the desired number of states by k, reduce the number of states by uniting k pairs of states: Determine the similarity rating for every pair of states, select a pair of states by the minimal product of the two components of the similarity rating, unite the pair of states, update the similarity ratings involving states that are adjacent to the new state, repeat.

Why does the selection minimize by the product of the two components? Other possibilities to select a pair include considering a fixed component, the minimal component of each tuple, and the sum. Just selecting the pair of states with the minimal rating with respect to one of the components is not always a good solution: If we unite two states that promise to add only a few new prefixes, this still might have a big impact if there are many suffixes added for each of those prefixes. It might be better to allow a little more prefixes if each of them only adds a moderate amount of suffixes. Similarly, minimizing by the sum of the components has pitfalls as well. If we have moderately many prefixes and suffixes with an apparently low sum, this might already have a big impact since each of the prefixes combined with each of the suffixes can form a new word.

An example we have seen before are similarity ratings with a component being 0. After forming the sum, we cannot tell whether a component was 0 before. If we consider the product, it will be 0 if at least one of the components was 0.

3.2 Chaining Intersections and Over-Approximations

The previously explained procedure does create over-approximations. Its input is still an exact intersection. The goal is to create an over-approximation of the intersection (as illustrated in Figure 1) without having to explore the whole state space of the cross-product automaton. In order to keep the input for the over-approximation step small, we want to create small cross-product automata that describe the intersection of only two automata. After applying the over-approximation procedure for this exact intersection, we intersect the over-approximation with another language. This chaining of exact intersections and over-approximations maintains the actual intersection: If a word is contained in all input languages, then it remains in any over-approximation (see Corollary 1) and in any intersection.

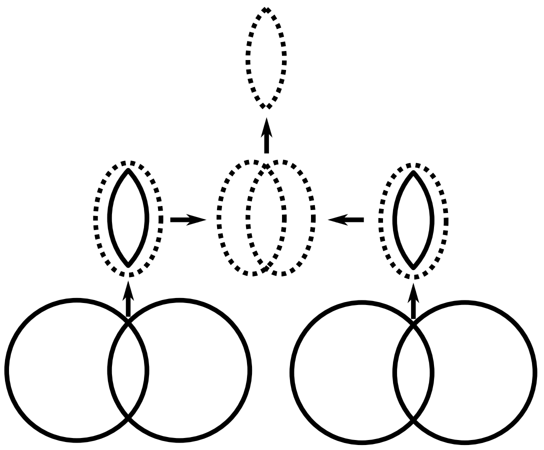
Figure 4: The tree-like chaining of exact intersection and over-approximations. The four circles represent the input languages. The dotted circles represent the over-approximation of an intersection.

This allows us to combine intersection and over-approximation steps in any order. We chose a tree-like procedure as shown in Figure 4: We compute exact intersections for pairs of automata and over-approximate the result. We start with pairs of input automata and repeat the same process for pairs of resulting languages. After logarithmically many layers of intersections and one intersection less than input languages, we are left with a single over-approximation of the intersection of all input languages.

Over-Approximating Empty Intersections Over-approximating languages might lead to an entire loss of information of the language. We already saw a worst-case scenario in Example 1 were the resulting automaton accepted all words over the alphabet $\{a, b\}$. Emptiness of intersections is a useful result: An empty intersection between a language and the complement of another language is equivalent to the inclusion of the first language in the latter. Thus, we want emptiness to propagate during the procedure: If an intersection is empty, the over-approximation shall stay empty so that all following intersections will become empty as well.

We first show a result on pairs of states that are both reachable through runs starting from the initial state. Uniting such pairs of states in automata that accept no words yields automata that still do not accept any word.

Lemma 3 (Uniting Reachable States Preserves Emptiness) *Let there be an automaton $A = (Q, q_0, \rightarrow, Q_F)$ and be $q_1, q_2 \in Q$ be states that are reachable from q_0. If $\mathcal{L}(A) = \emptyset$ holds, then $\mathcal{L}(A[(q_1, q_2); q']) = \emptyset$.*

Proof None of the states q_1 and q_2 accepts nor starts a run to an accepting state. Otherwise, the language of A could not be empty. Thus, the automaton A' after uniting q_1 and q_2 still does not have an accepting run from the initial state. ∎

If one of the two states is not reachable from the initial state, it can still be an acceptable state or start a run that leads to an accepting state although the language is empty. Uniting such a state with a state that is reachable from the initial state will add a word to the language of the automaton. The similarity rating alone does not prevent this:

Let q_{reach} be a state that is reachable from the initial state and $q_{unreach}$ be a state that is not reachable from the initial state. The similarity rating of these states is at least $sim(q_{reach}, q_{unreach}) = (0, 0)$ if both states have no transitions at all, q_{reach} is the initial state and $q_{unreach}$ the accepting state. Even if the states are neither initial nor accepting, uniting them can still add a word to the language when the similarity rating is $sim(q_{reach}, q_{unreach}) = (1, 1)$: The state q_{reach} has only one incoming transition. It is reachable from the initial state via this transition. The state $q_{unreach}$ has only one outgoing transition. An accepting state is reachable from $q_{unreach}$ via this transition. Once we unite q_{reach} and $q_{unreach}$, the accepting state becomes reachable from the initial state, and the language of the automaton is no longer empty.

The only way to keep a language empty when over-approximating it is to only unite reachable states. Recall that the languages to over-approximate are the results of intersections. In actual implementations, the cross-product automaton is usually generated by exploring both automata simultaneously and adding transitions when they are seen in both automata. The result is a cross-product automaton that only contains reachable states. Our over-approximation will propagate the emptiness of the languages of such automata.

Determining the Number of State Uniting Steps We have discussed the tree-like chaining of intersections and over-approximations. In the following, we present how to determine the number of state uniting steps. Since each step reduces the number of states by one, the number of state uniting steps defines the size of the automata we intersect. The problem that we address is the exponential size of the state space. Thus, we need to reduce the number of states in the automaton depending on the size of the intersected automata.

If the automata that are intersected have $|Q_1|$ and $|Q_2|$ states, then the resulting automaton will be over-approximated so that the number of states in the over-approximated automaton does not exceed $2 \cdot \max(|Q_1|, |Q_2|)$. The final automaton that over-approximates the whole intersection has up to $2^s \cdot |Q|$ states where $|Q|$ is the size of the biggest automaton that is initially intersected and s is the number of sequential intersections. Due to our tree-like construction, there are $s = \log_2(k)$ sequential intersections and $|Q| = \max\{|Q_i| \mid Q_i \text{ state set of an input automaton }\}$. Thus, the resulting automaton has up to $2^{\log_2(k)} \cdot |Q| = k \cdot |Q|$ states. Before the last over-approximation, the last cross-product automaton consists of up to $\left(2^{\log_2(k)-1} \cdot |Q|\right)^2 = \frac{k^2}{4} \cdot |Q|^2$ states. Unlike the exact computation of the intersection, our over-approximating procedure does not have to deal with an exponential but only with a polynomial state space.

Keeping Down the Number of Candidates We have ensured that the size of the intersection automata is always bounded by a polynomial in the size and number of input automata. This requires us to reduce the number of states from $\frac{k^2}{4} \cdot |Q|^2$ to $k \cdot |Q|$.

Since we consider all pairs of states as possible candidates for a state union, we need to compute the similarity rating for any possible state pair. If there are n states, there are $\frac{n\cdot(n-1)}{2}$ unique pairs of distinct states. If we want to reduce n to m states, we need to unite $n - m$ pairs of states. Hence, we need to consider up to

$$\begin{aligned}
&\sum_{i=1}^{n-m} \frac{(n-i+1)\cdot(n-i)}{2} = \sum_{i=1}^{n-m} \frac{n^2 - 2\cdot i\cdot n + n - i^2 - i}{2}\\
=&(n-m)\cdot\frac{n^2}{2} + n\cdot\frac{(n-m+1)\cdot(n-m)}{2} + (n-m)\cdot\frac{n}{2}\\
&+\frac{(n-m)\cdot(n-m+1)\cdot(2n-2m+1)}{12} + \frac{(n-m)\cdot(n-m+1)}{4}\\
=&-\frac{m}{3} + \frac{m^2}{2} - \frac{m^3}{6} + \frac{n}{3} - 2\cdot m\cdot n + m^2\cdot n + \frac{3\cdot n^2}{2} - 2\cdot m\cdot n^2 + \frac{7\cdot n^3}{6}
\end{aligned}$$

similarity ratings of state pairs. This means that the over-approximation of the last intersection step requires up to

$$\frac{7}{384}\cdot k^6\cdot|Q|^6 - \frac{1}{8}\cdot k^5\cdot|Q|^5 + \frac{11}{32}\cdot k^4\cdot|Q|^4 - \frac{2}{3}k^3\cdot|Q|^3 + \frac{7}{12}k^2\cdot|Q|^2 - \frac{1}{3}\cdot k\cdot|Q|$$

state similarity ratings. We see that the dependency between the number of state similarity ratings and the input size is polynomial with a degree of 6. In order to speed up the procedure and to make it feasible for bigger input automata, we need to reduce the number of state pairs that are being rated. We use two approaches to reduce the number of comparisons. The first one is a side-effect of a heuristic with the main purpose to improve the accuracy of the over-approximation.

In Lemma 2, we showed that zero as a component of the similarity rating implies that uniting the rated states does not alter the language under some circumstances. Those circumstances included that the two states may not disagree in the acceptance behavior if they have exclusive incoming transitions. The problem is that uniting an accepting and a non-accepting state has to result in a state that accepts in order to keep words in the language that were accepted by this accepting state. However, this also adds words that only lead the run to a non-accepting state. Another problem is that uniting states spreads the acceptance behavior over the automaton: Whenever we unite two states that disagree in the acceptance behavior, we lose a non-accepting state. In a worst case scenario, all states in the automaton could become accepting states.

Thus, we chose the heuristic not to unite accepting with non-accepting states. This also allows us to compute the state similarity ratings of fewer state pairs: If we compared n states, we needed to compute $\frac{n\cdot(n-1)}{2}$ state similarity ratings. If $\alpha \cdot n$ out of n states accept, we can reduce the number of considered state pairs to $\frac{(1-\alpha)\cdot n\cdot((1-\alpha)\cdot n-1)}{2} + \frac{n\cdot(n-1)}{2}$. The highest impact can be observed when half of the states are accepting. In this case, we save more than a half of the state similarity rating and we are left with $\frac{1}{2} \cdot \left(1 - \frac{1}{n-1}\right)$ of the similarity ratings that the whole set of state pairs required.

On Automata Minimization The presented heuristic has the side-effect of reducing the number of considered state pairs. Automata minimization is another method that increases the number feasible practical examples. The automaton in Figure 3 gives an example how uniting two states does not always add additional words to the language of the automaton. Lemma 2 named cases when the state similarity rating will pick state pairs with this property. This means that many of the uniting steps are in fact minimization steps as they do not actually over-approximate the language. Each of those steps still requires us to update the state similarity ratings, which can mean up to quadratically many entries.

Thus, a minimization of the automaton prior to the over-approximation saves effort. Automata minimization algorithms can reduce the number of states without changing the language. This shifts minimization effort from the over-approximation algorithm to an actual minimization algorithm.

The problem is the worst-case complexity: Minimizing automata in order to obtain non-deterministic finite automata is known to be PSPACE-complete as shown in [JR93]. The Hopcroft minimization algorithm is sub-quadratic but it requires the input automaton to be deterministic. The deterministic equivalent of a non-deterministic finite automaton can be exponentially bigger as shown in [Moo71].

An example is the language of words that have a specific letter at the n-th position from the end. The non-deterministic automaton of this language requires n states: When the special letter is seen, a non-deterministic transition allows to leave the initial state to a chain of states that accepts an n letter long suffix. Since the deterministic automaton cannot guess which occurrence of the special letter is the n-th position from the end, it needs to keep track of all occurrences in the last n positions. As there are 2^n different possibilities, there also need to be that many states.

The determinization of worst-cases like the mentioned one takes exponentially many steps and the result cannot be used as even the minimized version is bigger than the original non-deterministic automaton. In this case, the similarity ratings have to be computed on the original automaton and the minimization is in vain. In experiments, a determinization and an application of Hopcroft did pay out: Adding the minimization step extended the possible input size of random examples while still being able to compute previously feasible examples. The over-approximation itself was still required in the experiments: A tree-like chaining of intersections steps with applications of the Hopcroft minimization but without an over-approximation step lead to automata that were too big for eight gigabytes of RAM.

In practical applications, it might be a critical question whether a minimization procedure should be applied or not. The decision depends on the actual use-case scenario. If the involved classes of languages tend to show the described worst-case behavior, attempts to minimize these automata should be avoided. Another possibility is an opportunistic approach: At the beginning of each minimization attempt, a maximal duration or determinization and minimization is determined. Once this duration is exceeded, the procedure directly switches to over-approximating the not minimized automaton.

Advantages of the Tree-Like Approach
We illustrated the tree-like approach of chaining actual intersection steps with over-approximation steps in Figure 4. We explained several effects like the preservation of emptiness by the over-approximation procedure and the remaining details how the steps of the over-approximation procedure are combined. Not all of those properties and details did directly depend on the tree-like approach. Yet, the choice of this concrete chaining pattern does have implications on the procedure and its result. In the following, we will motivate this choice by comparing a different pattern.

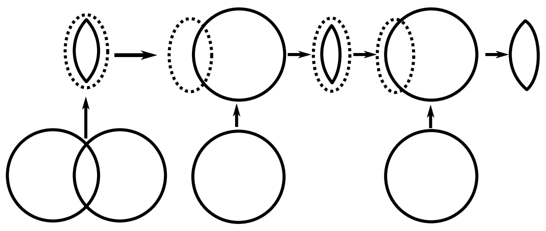
Figure 5: The sequential chaining of exact intersection and over-approximations. The circles represent the input languages. The lower row represents the input. The upper row shows the procedure with its intermediate and final results.

The sequential approach as illustrated in Figure 5 is a different possible pattern. Initially, two of the input languages are intersected. All other input automata are sequentially intersected with a single intermediate language. A technical difference is the change of the sizes of the involved sets. If the input automata are all of the same size, then the tree-like approach always intersects automata of the same size as well. The sequential approach intersects the potentially big intermediate result with comparably small input automata.

Thus, the desired automata size for the over-approximation steps needs to be adjusted. The tree-like approach doubles the size once per layer. In the sequential approach, it needs to grow in every step as the intermediate intersection represents an increasing number of automata. Therefore, the desired size after the i-th intersection is $|Q| + i \cdot |Q|$ where $|Q|$ is the maximal size of state sets of the input automata. If there are k input automata, the resulting size is $k \cdot |Q|$, which is the same size as in the tree-like approach.

An interesting difference between the approaches is the point in time when all input automata were considered at least once. In the tree-like approach, this happens after $\frac{k}{2}$ of $k - 1$ intersection steps, assuming that the intersection steps are done layer-per-layer. The sequential approach considers only $\frac{k}{2} + 1$ automata within that many intersection steps. If the last two automata have an empty intersection, the tree-like approach will be aware of that after $\frac{k}{2}$ of the intersection steps while the sequential approach will take all $k - 1$ intersection steps. Conversely, if the first half of the input automata is the smallest subset of the input automata with an empty intersection, the sequential approach determines that after $\frac{k}{2} - 1$ intersection steps. The layer-wise tree-like approach takes $k - 3$ intersection steps as it needs to compute a child node of the root in the tree of intermediate intersections.

These examples show that the advantage of one of the two approaches in determining emptiness depends on the concrete input. Even the exact order of the input automata matters: If both the first and the last automaton need to be considered in order to conclude emptiness, both approaches need to do all intersection steps. If the second half of automata is the one with the empty intersection, then the sequential approach needs to do all intersections. Although there might be classes of inputs where one approach can conclude emptiness earlier, the advantages disappear for unfortunate input orders.

A difference when it comes to practical implementation is the ability to execute the steps in parallel. The idea for the tree-like approach is the following: If we want to execute the procedure in 2^i threads, we let each thread compute a node that is i layers below the root node. This is done by following the node to its leaves. The thread runs the tree-like approach just on the input automata that correspond to the leaves. Whenever two threads finish, we combine their results by letting a thread intersect and over-approximate the result. In this way, we make use of the fact that there is not a single intermediate result in the tree-like approach on which everything is dependent.

The sequential approach depends on the single intermediate result. Thus, it can not be split to several threads. The solution is a slight variant of the approach: The set of input automata is split in parts of equal size and each thread executes the sequential approach on its part of the input set. The results can then be grouped to be processed by several threads again. This procedure is repeated until a single result is left.

This parallelizeable variant of the sequential approach is tree-like as well: Whenever a set of automata is processed by the sequential approach, the resulting intermediate result is the parent node of the automata in the set. Thus, the variant can be seen as non-binary tree-like.

The final decision for the tree-like approach was made due to a property of the result itself: An intermediate result carries information about all automata whose intersection it over-approximates. If the number of automata k is a power of 2, the information about each input automaton will be processed equally often, namely $\log_2(k)$ times, which is once per each layer of the tree.

In a strictly sequential approach, this is not the case: If the input is a list of automata $A_1, \dots, A_k$, the automaton A_1 will be considered in each of the $k-1$ intermediate results and thus in $k-1$ intersection and over-approximation steps. Any other automaton A_i with $1 < i \leq k$ will be considered $k-i+1$ times. This means, that information concerning the first automaton will be subject to the over-approximation $k-1$ times as often as information concerning the last automaton. Thus, we choose the tree-like approach in the hope that information on the automaton that occur earlier in the list is fairly considered.

4 On Counterexamples

Section 3 showed how an over-approximation of the intersection of the input automata is obtained. The over-approximation intersection can now be checked against the specification. If $\mathcal{L}$ is the over-approximation of the intersection of the input automata $\mathcal{L}(Reg_1) \cap \cdots \cap \mathcal{L}(Reg_k)$, and $\mathcal{L}(Reg)$ is the specification, then a word w that is in the over-approximation $w \in \mathcal{L}$ is a *counterexample.*

A counterexample in the over-approximated intersection $\mathcal{L}$ exists if and only if the over-approximation $\mathcal{L}$ is not included in the specification $\mathcal{L}(Reg)$. Since the over-approximation $\mathcal{L}$ contains the intersection $\mathcal{L}(Reg_1) \cap \cdots \cap \mathcal{L}(Reg_k) \subseteq \mathcal{L}$, the absence of a counterexample implies that the intersection is included in the specification $\mathcal{L}(Reg)$. However, the existence of a counterexample in the over-approximation $\mathcal{L}$ does not imply that the intersection is not included in the specification. A counterexample $w \in \mathcal{L}, w \notin \mathcal{L}(Reg)$ is *spurious* if it only exists in the over-approximation but not in the actual intersection $w \notin \mathcal{L}(Reg_1) \cap \cdots \cap \mathcal{L}(Reg_k)$. Hence, the absence of non-spurious counter-examples in over-approximation $\mathcal{L}$ is equivalent to the inclusion of the intersection in the specification $\mathcal{L}(Reg_1) \cap \cdots \cap \mathcal{L}(Reg_k) \subseteq \mathcal{L}(Reg)$.

The inclusion of the intersection in the specification can hence be determined by testing for the existence of a non-spurious counterexample. It is not feasible to compute the exact set of all non-spurious counterexamples because this requires the exact intersection of the input automata.

Thus, the test for non-spurious counterexamples is done in this way: If there is a counterexample w, check whether w is spurious. If it is not, then the inclusion is disproven. If it is a spurious counterexample, proceed with the next one as long as there are counterexamples available. If no counterexample is non-spurious, the inclusion is proven.

Unlike determining the whole set of non-spurious counterexamples, the spuriousness check for a single counterexample w is feasible: A counterexample w is non-spurious if and only if it is included in all of the intersected languages $\mathcal{L}(Reg_1), \ldots, \mathcal{L}(Reg_k)$. This check can be done without obtaining the actual intersection if we just solve the word problem for w and each of the languages.

Performing the spuriousness check for every single counterexample is only a decider for finite sets of counterexamples. Thus, we will present a way to check certain infinite subsets of the set of counterexamples at once. It does not only extend the class of languages of counterexamples that can be decided. It also speeds up the search for non-spurious counterexamples by ruling out infinitely many spurious counterexamples at once.

In this section, we will discuss how to extend a single counterexample to an infinite language of counterexamples so that the existence of non-spurious counterexamples can be checked. We will give an overview on this procedure, name difficulties, and explain how to address them. Section 5 will cover technical and theoretical details on obtaining those languages. Sections 6 and 7 will explain the technique behind the said spuriousness check.

4.1 Generalizing Counterexamples

We are given a language $\mathcal{L}$ that over-approximates an intersection $\mathcal{L}(Reg_1) \cap \cdots \cap \mathcal{L}(Reg_k) \subseteq \mathcal{L}$ and a specification $\mathcal{L}(Reg)$. In the following, we will discuss how to use a counterexample $w \in \mathcal{L}$, $w \notin \mathcal{L}(Reg)$ to obtain a generalized counterexample $\mathcal{L}^w, w \in \mathcal{L}^w$. Section 5 presents technical details on this generalization. Subsection 4.2 explains a quick procedure to check whether there is a non-spurious counterexample in the generalization $\mathcal{L}^w$. Technical details on the necessary preparations can be found in Section 6. Technical details on the inclusion check itself will be given in Section 7.

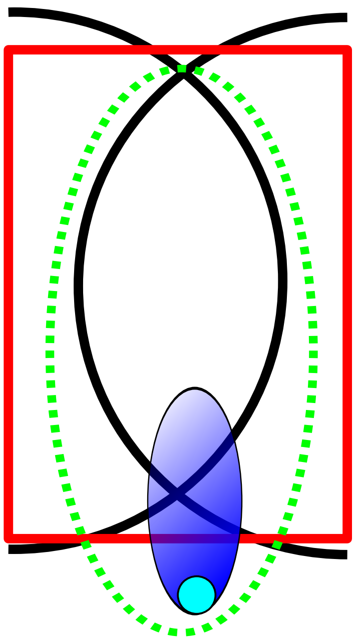

Figure 6: The generalization $\mathcal{L}^w$ (small ellipse) of the counterexample w (circle) is *spuriously non-spurious.*: The part that touches the intersection does not violate the specification $\mathcal{L}(Reg)$ (rectangular). Only the over-approximation $\mathcal{L}$ (dotted ellipse) violates it.

Generalizing counterexamples can lead to *spuriously non-spurious counterexamples.* Figure 6 illustrates when they occur: There is a spurious counterexample $w \in \mathcal{L}$, $w \notin \mathcal{L}(Reg)$, $w \notin \mathcal{L}(Reg_1) \cap \cdots \cap \mathcal{L}(Reg_k)$. The generalization $\mathcal{L}^w$ of the counterexample does contain words that are in the intersection. However, those words do not violate the specification. The problem is that such generalizations can lead to falsely concluding that there are non-spurious counterexamples.

Avoiding this wrong conclusion is hence a requirement of our procedure. We chose not to enforce this requirement when checking the generalization of the counterexample for non-spurious counterexamples. This could be done by using the intersection not only to ensure a counterexample to be non-spurious, which means that it is in the intersection $\mathcal{L}(Reg_1) \cap \cdots \cap \mathcal{L}(Reg_k)$, but also to check whether it actually disobeys the specification, which means that also $\overline{\mathcal{L}(Reg)}$ is considered in the intersection. Our procedure generalizes the counterexamples in a way so that each word in the generalization disobeys the specification. This means that we construct a generalization $\mathcal{L}^w$ of the counterexample w that is included in the language of all possible counterexamples $\overline{\mathcal{L}(Reg)} \cap \mathcal{L}$.

This does not only supersede the check of the generalization against the complemented specification $\overline{\mathcal{L}(Reg)}$ but it can also speed up the elimination of non-spurious counterexamples: If all words in the generalization $\mathcal{L}^w$ are spurious, they will be excluded from the over-approximation by intersecting the over-approximation with the complement of the generalization. Since the generalization consists only of words that are in the over-approximation and not in the specification, each word in the generalization is a potential counterexample that we will not need to check in a later step of the loop. This avoids cases where the counterexample w is the only representative its generalization $\mathcal{L}^w$ that actually violates the specifications. Such cases are expensive as they require the whole intersection check and the refinement without significantly reducing the set of possible counterexamples that remain to be checked.

Thus, the procedure works as follows:

1. Compute the automaton describing all possible counterexamples $\mathcal{L}_C := \mathcal{L} \cap \overline{\mathcal{L}(Reg)}$.
2. Find a counterexample $w \in \mathcal{L}_C$.
3. Generalize the counterexample w along the automaton describing the set of possible counterexamples $\mathcal{L}_C$ to $\mathcal{L}^w$ (see Section 5).
4. Use the intersection check (see Subsection 4.2) to test for counterexamples that are also in $\mathcal{L}(Reg_1) \cap \dots \cap \mathcal{L}(Reg_k)$.
    - If there is any, the inclusion does not hold.
    - If there is none, repeat but exclude the entirely spurious generalized counterexample. The new language of possible counterexamples is thus $\mathcal{L}'_C := \mathcal{L}_C \cap \overline{\mathcal{L}^w}$.

4.2 Fast Intersection Checks

Once we have a generalization $\mathcal{L}^w \subseteq \overline{\mathcal{L}(Reg)} \cap \mathcal{L}$ of a counterexample w, we need to check whether it contains any non-spurious counterexample. This means that we want to decide $\mathcal{L}^w \cap \mathcal{L}(Reg_1) \cap \dots \cap \mathcal{L}(Reg_k) \overset{?}{=} \emptyset$.

To this end, we define the Fast Intersection Check as follows. The underlying techniques Elementary Bounded Languages, Parikh Images, and Presburger Formulas will be introduced throughout this section.

1. A generalization of a counterexample w is given as a regular Elementary Bounded Language $\mathcal{L}^w = \{w_1^{x_1} \dots w_l^{x_l} \mid (x_1, \dots, x_l) \in L\}$ where $L \subseteq \mathbb{N}^l$ is a semi-linear set.
2. Introduce fresh letters $a_1, \dots, a_l$ for the infixes $w_1, \dots, w_l$ and obtain another Elementary Bounded Language $\mathcal{L}'^w = \{a_1^{x_1} \dots a_l^{x_l} \mid (x_1, \dots, x_l) \in L\}$ using the same semi-linear set L.
3. Given k regular languages $\mathcal{L}(Reg_1), \dots, \mathcal{L}(Reg_k)$, try to express its words as sequences of the letters $a_1, \dots, a_l$, representing the infixes $w_1, \dots, w_l$. Skip words that cannot be expressed using theses infixes.
   $\mathcal{L}(Reg'_i) = \{a_{i_1} \dots a_{i_m} \in \{a_1, \dots, a_l\}^* \mid 1 \leq i_m \leq l, m \in \mathbb{N}, w_{i_1} \dots w_{i_m} \in \mathcal{L}(Reg_i)\}$ for $1 \leq i \leq k$.
4. Intersect the resulting regular languages with the modified Elementary Bounded Language: $\mathcal{L}(Reg'_i) \cap \mathcal{L}'^w$ for $1 \leq i \leq k$.

5. Describe the Parikh Images of $\mathcal{L}\left(Reg'_1\right) \cap \mathcal{L}'^w, \ldots, \mathcal{L}\left(Reg'_k\right) \cap \mathcal{L}'^w$ as Presburger Formulas. The Parikh Images are of the form $\{(x_1, \ldots, x_l) \in L \mid w_1^{x_1} \ldots w_l^{x_l} \in \mathcal{L}\left(Reg_i\right)\}$.
6. Intersect the Parikh Images by forming the conjunct of the Presburger Formulas and deciding satisfiability.
   - The conjunct is satisfiable. Then, there is a satisfying assignment of the variables $x_1, \ldots, x_l$. Thus, the word $w_1^{x_1} \ldots w_l^{x_l}$ is in $\mathcal{L}\left(Reg'_1\right), \ldots, \mathcal{L}\left(Reg'_n\right)$ and $\mathcal{L}'^w$. *The intersection is non-empty.*
   - The conjunct is not satisfiable. *The intersection is empty.*

The basic idea is that we project the words to a domain where we can test more easily whether the given languages have an empty intersection. In general, this does find all non-empty intersections but not all empty intersection: If the intersection in the original domain is non-empty, then there is at least one element in it, whose projection is also in the projection of all of the intersected sets and thus in the intersection of the projection. If the intersection in the original domain is empty, then there still might be different elements in each of the intersected sets that have the same projection. In this case, the projection has a non-empty intersection although the intersection in the original domain is empty.

We can still use this approach to actually decide emptiness for intersections with the following property: Non-empty intersections in the projection need to contain at least one unique element. This element in the projected domain is unique in the sense that there is only one element in the intersection in the original domain that is projected it.

We strengthen the requirement and get a property that implies the previous one. Instead of requiring the existence of an element with a certain property, we extend the property to all elements in the intersection: We are interested in instances of the intersection problem whose intersection in the new domain is guaranteed to only consist of elements with a representative in the original domain that is in each of the intersected sets.

We produce such instances by preventing the generalization of the counterexample from having several elements that are projected to a single element. Before projecting the input languages, we intersect each of them with this generalization of the counterexample. The intersection of these languages is still the same. Yet, the intersection of the projected languages is different: Intersecting with the generalization of the counterexample rules out elements that are projected to the same element as others in the set. Thus, the instance does have the desired property.

Existential Presburger Formulas In order to solve the said emptiness problem in the projected domain, we use formulas in first order predicate logic. Each formula represents a set of the elements in the projected domain that are satisfying assignments of the formula. We solve the emptiness of the intersection of the sets by checking satisfiability of the conjunct of formulas. Satisfiability of first order predicate logic is in general not decidable. However, satisfiability of first order formulas is decidable for certain fixed theories as shown in [Sca84].

We use *Existential Presburger Arithmetic*. This is a fragment of first-order logic formulas whose only predicate is the equality of sums.

Definition 5 (Existential Presburger Arithmetic) The signature of a *formula in Presburger Arithmetic* is:

- Domain: $\mathcal{D} := \mathbb{N}$
- Functions: $0_{/0}, 1_{/0}, +_{/2}$
- Predicate: $=_{/2}$

The functions and the predicate are interpreted as expected in the natural numbers.

A *formula in Existential Presburger Arithmetic* is equivalent to a Presburger Formula of the form $\exists x_1 \dots \exists x_k B$ for some $k \in \mathbb{N}$ and a quantifier-free Presburger Formula B. $\square$

There are interesting properties of our syntax of Presburger Arithmetic:

- Domain: The natural numbers are the domain. However, an integer variable can be represented by a natural number variable for the positive component and one for the negative component.
- Functions: The only non-constant function is the addition. Subtraction can be represented as an addition on the other side of an equation or inequality. Multiplication between a constant c and a variable can be expressed as a sum of c occurrences of the variable. Constants in the natural numbers are represented as sums of 1. The constant 0 is redundant: As the neutral element of addition, it can be left out of sums. If the constant 0 appears as a side of an equation or inequality, it can be turned to a sum by adding 1 to both sides of the equation or inequality.
- Predicates: Equality is the only predicate. Less-than-or-equal can be expressed using an additional variable that is existentially quantified: For example, the formula $\exists z : x + z = y$ is equivalent to $x \leq y$.

We will make use of formulas in Existential Presburger Arithmetic as their satisfiability problem is in $\mathcal{NP}$ as shown in [Sca84]. Thus, the sets need to have a representation as an Existential Presburger Formula. Each Presburger Formula can be transformed into an Existential Presburger Formula as shown in [Coo72]. However, the size of the formula grows exponentially when eliminating the universal quantifiers as discussed in [LOW15]. We will present a procedure that purely relies on sets that can be defined by Existential Presburger Formulas. We will present a procedure in Section 7 to obtain these formulas and show that their size is still polynomial although they do not contain universal quantifiers.

The *Parkih Image* of a language is a set of vectors of natural numbers that represent words by carrying the frequency of each letter of the alphabet in a component of the vector. Parikh Images of context-free (and hence regular languages as well) are semi-linear.

This is the main statement of Parikh's Theorem as republished in [Par66]. According to [GS66], semi-linear sets can be defined by Presburger Formulas.

We can check the satisfiability of the conjunct of the Existential Presburger Formulas describing the Parikh Image of regular languages. This means that we know whether there is an assignment that satisfies all of the formulas. This assignment corresponds to a vector in the Parikh images. The existence of such a vector in all of the sets proves that there are some permutations of a word so that each language contains at least one of the permutations.

Elementary Bounded Languages It does not suffice to only check the existence of permutations of a word. The idea is to shape the generalization of the counterexample $\mathcal{L}^w$ so that each word can be represented by vectors of natural numbers so that no pair of words shares the same vector. We then present a way to obtain these vectors as a Parikh Image of a projection of the languages. To this end, we use so called *Elementary Bounded Languages*. All words in a regular language are composed out of the same infixes in the same, fixed order. Only the number of repetitions differs. Hence, each word in the Elementary Bounded Language corresponds to a vector.

Definition 6 (Elementary Bounded Language (EBL)) Let $w_1, \dots, w_l$ be words and the vector set $L \subseteq \mathbb{N}^l$ be definable by Elementary Functions. A language $\mathcal{L}$ of the form

$$\mathcal{L} = \{w_1^{x_1} \dots w_l^{x_l} \mid (x_1, \dots, x_l) \in L\},$$

is an *Elementary Bounded Language* over the *infixes* $w_1, \dots, w_l$. □

A vector $(x_1, \dots, x_l) \in L$ denotes the numbers of repetitions of the infixes in a word $w_1^{x_1} \dots w_l^{x_l} \in \mathcal{L}$.

The class of Elementary Bounded Languages is incomparable to the classes of regular, context-free and context-sensitive languages. For example the regular language $\{a, b\}^*$ is not an Elementary Bounded Language. The language $\{(ab)^{x_1}(ba)^{x_2} \mid x_1 = x_2 \in \mathbb{N}\}$ is an Elementary Bounded Language, a context-free language and not a regular language. The language $\{(ab)^{x_1}(ba)^{x_2} \mid x_1, x_2 \in \mathbb{N}\}$ is also an Elementary Bounded Language but it is regular. The regular language $\{a, b\}$ is not an Elementary Bounded Language. The Elementary Bounded Language $\{a^{x_1}b^{x_2}c^{x_3} \mid x_1 = x_2 = x_3 \in \mathbb{N}\}$ is not context-free. There are subclasses of the class of Elementary Bounded Languages that consist only of regular languages:

Lemma 4 *An Elementary Bounded Language $\mathcal{L} = \{w_1^{x_1} \dots w_l^{x_l} \mid (x_1, \dots, x_l) \in L\}$ is regular, if the components of L do not depend on each other, i.e. $L = L_1 \times \dots \times L_l$ and $L_1, \dots, L_l \subseteq \mathbb{N}$, and each component is semi-linear, i.e. $L_1, \dots, L_l$ are semi-linear sets.*

Proof The proof is constructive. Since the sets $L_1, \dots, L_l$ are independent, the language $\mathcal{L}$ is a concatenation of languages of the form $\{w_i^{x_i} \mid x_i \in L_i\}$. Since regular languages are closed under concatenation, it remains to be shown that each of these languages is regular. Since each set L_i is semi-linear, there is a regular language using a single letter so that the word lengths of this regular language correspond to the set L_i. If we replace any edge in this automaton by a sequence of edges reading w_i, we get an automaton for the language $\{w_i^{x_i} \mid x_i \in L_i\}$. ■

The procedure in Section 5 generalizes the counterexample to an Elementary Bounded Language. There, the sets $L_1, \dots, L_l$ are either $\mathbb{N}$ or $\{1\}$. This means that the language is described by a regular expression that consists of the infixes $w_1, \dots, w_l$ and Kleene stars for some of the infixes.

The useful property of Elementary Bounded Languages is the fixed order of the suffixes: A vector $(x_1, \dots, x_l) \in \mathbb{N}^k$ denoting the number of repetitions of the suffixes describes a single word $w_1^{x_1} \dots w_l^{x_l}$. This is not the case for vectors in Parikh Images: For example, the Parikh Image of $(ab)^*(ba)^*$ is the set of all vectors in $\mathbb{N}^2$ with equal first and second component. The words *abab* and *baba* are both represented by the vector $(2, 2)$ in the Parikh Image. The vectors describing the number of repetitions in the Elementary Bounded Language are $(2, 0)$ and $(0, 2)$.

The set of vectors denoting the frequency of infixes in the Elementary Bounded Language can still be expressed as a Parikh image: If we represent each of the infixes $w_1, \dots, w_l$ as a fresh letter $a_1, \dots, a_l$ and determine the Parikh Image of the modified word, then the Parikh Image vector coincides with the corresponding vector denoting the frequency of each infix in the original word. For example, if we represent the infix *ab* as *c* and *ba* as *d*, the language $(ab)^*(ba)^*$ will be represented as c^*d^*. The words *abab* and *baba* will be represented as *cc* and *dd*. The Parikh Images of the modified words are $(2, 0)$ and $(2, 0)$, which coincides with the exponents in the representation $(ab)^2(ba)^0$ and $(ab)^0(ba)^2$.

If there are words in an intersection between an Elementary Bounded Language and other languages, then those words can certainly be expressed by the vector denoting the frequency of infixes. In order to intersect languages with the Elementary Bounded Language using Presburger Formulas describing sets of such vectors, we need to represent the languages as sets of those vectors. This means that all words of the language need to be split in infixes of the Elementary Bounded Language and represented as a sequence of the letters that represent the corresponding infix. If we want to intersect a language $\mathcal{L}_{in}$ with an Elementary Bounded Language $\mathcal{L} = \{w_1^{x_1} \dots w_l^{x_l} \mid (x_1, \dots, x_l) \in L\}$ over the infixes $w_1, \dots, w_l$ that are represented by the letters $a_1, \dots, a_l$, we first prepare the language

$$\mathcal{L}'_{in} = \{a_{i_1} \dots a_{i_l} \in \{a_1, \dots, a_k\}^* \mid \text{ex. } k \in \mathbb{N}, w_{i_1} \dots w_{i_k} \in \mathcal{L}_{in}, 1 \leq i_j \leq l \text{ f.a. } 1 \leq j \leq k\}\,.$$

A constructive approach to obtain $\mathcal{L}'_{in}$ from an automaton for $\mathcal{L}_{in}$ will be presented later in Section 6. However, the resulting language $\mathcal{L}'_{in}$ might still contain words with the same Parikh Image as different words in other languages or the Elementary Bounded Language. The problem is that we do not enforce an order of the infixes. We solve this by intersecting the resulting language $\mathcal{L}'_{in}$ with the language $\mathcal{L}' = \{a_1^{x_1} \dots a_l^{x_l} \mid (x_1, \dots, x_l) \in L\}$ that represents the Elementary Bounded Language $\mathcal{L}$.

The intersection is of the form

$$\mathcal{L}'_{in} \cap \mathcal{L}' = \{a_1^{x_1} \dots a_l^{x_l} \in \mathcal{L}' \mid w_1^{x_1} \dots w_l^{x_l} \in \mathcal{L}_{in}, (x_1, \dots, x_l) \in L\} .$$

The Parkih Image of $\mathcal{L}'_{in} \cap \mathcal{L}'$ consists of vectors that represent a single word since the Elementary Bounded Language enforces the order or the infixes. If we prepare all languages that we want to intersect in the same way, it suffices to intersect their Parikh Images. Thus, we came up with the procedure we presented at the beginning of this section.

Note on Non-Regular Elementary Bounded Languages Note that this procedure does not necessarily require the generalization $\mathcal{L}^w$ to be regular. Regularity was used in step 4: The intersection of two regular languages is still regular. Thus, we are sure that the result is still a finite automaton from which we can then extract the Presburger Formula describing the Parikh Image. Our way to obtain the Presburger Formula describing the Parikh Image requires finite automata. However, even if the Elementary Bounded Language is non-regular, a regular intersection suffices at that point:

If the generalization $\mathcal{L}^w = \{w_1^{x_1} \dots w_l^{x_l} \mid (x_1, \dots, x_l) \in L\}$ is non-regular but L is still semi-linear, we can intersect the input languages with a regular hull $w_1^* \dots w_l^* \supseteq \mathcal{L}^w$ of $\mathcal{L}^w$. This is already enforces the order of the infixes. Yet, we still need to consider the actual Elementary Bounded Language in the intersection of the Parikh Images. This can be done by obtaining a Presburger Formula describing L since L is semi-linear. We then add the Presburger Formula to the conjunct.

5 Generalizing Counter-Examples as Elementary Bounded Languages

The over-approximation of the intersection may contain words that violate the specification. Such counterexamples might be actual counterexamples if they disprove the inclusion or spurious counterexamples. We motivated in Section 4.1 that we want to check the spuriousness of infinitely many counterexamples at once and want to ignore this whole set if it is entirely spurious. Thus, we will present a procedure to find such generalized counterexamples.

The generalized counterexamples are Elementary Boundeded Languages since the procedure we presented in Section 4.2 requires such languages. As mentioned in Section 4.1, entirely spurious counterexamples will be ignored by intersecting the language of potential counterexamples with the the complement of the set of spurious counterexamples. Hence, the languages of generalized counterexamples will be regular.

If $\mathcal{L}(Reg)$ is the specification and $\mathcal{L}$ is the over-approximation of the intersection, then $\mathcal{L}_C := \mathcal{L} \cap \overline{\mathcal{L}(Reg)}$ is the set of exactly those potential counterexamples that might be in the intersection and violate the specification. After excluding a set of spurious counterexamples, the language $\mathcal{L}_C$ will be updated. We assume that the language of possible counterexamples is given as an automaton A_C

We present an approach whose input is this automaton A_C: The given automaton A_C contains potential counterexamples that we have not examined before. A path from the initial state to an accepting state of A_C admits a counterexample w. A language $\{w\}$ containing a single word w is already a finite Elementary Bounded Language as it can be expressed as a sequence of infixes with an exponent that is fixed to 1. The idea is to search for loops along the path to the accepting state. Such loops are then added as infixes with a Kleene star.

This is done in a procedure like this:

1. Given is the automaton A_C describing the non-empty language of potential counterexamples $\mathcal{L}_C$.

2. Perform a depth first search to find a run to an accepting state. Extract the counterexample w that is read by this run.

3. Move along the run to the accepting state. For each state, run a depth first search to find a run to the state itself. If the run to the state itself reads w_{loop}, insert w_{loop}^* at the corresponding place of the word that is read along the run to the accepting state.

4. The result is an Elementary Bounded Language of the form

$$\mathcal{L}^w = \left\{ w_1^{x_1} \dots w_l^{x_l} \mid (x_1, \dots, x_l) \in L \subseteq \mathbb{N}^l \right\} \text{ with } L = L_1 \times \cdots \times L_l, L_i \in \{\{1\}, \mathbb{N}\}$$
$$\text{for all } 1 \le i \le l \text{ and } w = w_1^{y_1} \dots w_l^{y_l} \text{ with } y_i = 1 \text{ if } L_i = \{1\}$$
$$\text{and } y_i = 0 \text{ if } L_i = \mathbb{N} \text{ for all } 1 \le i \le l.$$

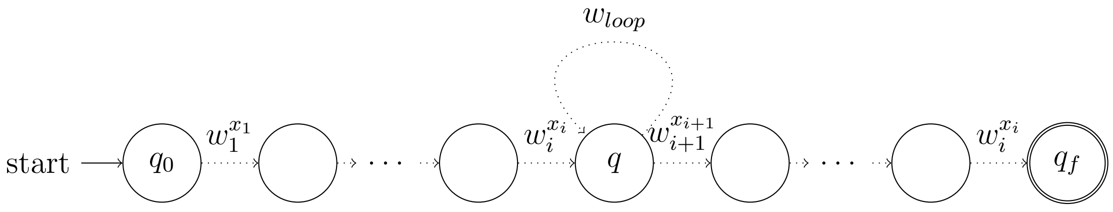

Figure 7: The run of A reading $w_1^{x_1} \dots w_i^{x_i} w_{i+1}^{x_{x+1}} \dots w_l^{x_l}$ leads through a state q.

We will first formalize the said basic principle of this approach and show that it works as intended. After that, we will explain how to find loops and how to add them to the Elementary Bounded Language.

Lemma 5 (Adding Loops to Elementary Bounded Languages) .
Let $A = (Q, q_0, \rightarrow, Q_F)$ be an automaton whose language contains an Elementary Bounded Language $\mathcal{L} = \{w_1^{x_1}, \dots, w_l^{x_l} \mid (x_1, \dots, x_l) \in L\} \subseteq \mathcal{L}(A)$. Let w_i and w_{i+1} be infixes and $q \in Q$ a state of A so that for every word $w = w_1^{x_1} \dots w_i^{x_i} w_{i+1}^{x_{i+1}} \dots w_l^{x_l} \in \mathcal{L}$ there is a run from the initial state q_0 to an accepting state $q_f \in Q_F$ that passes state q after reading $w_1^{x_1} \dots w_i^{x_i}$.

If there is a run from state q to state q reading the infix w_{loop}, then the Elementary Bounded Language $\mathcal{L}' = \left\{w_1^{x_1}, \dots, w_i^{x_i} w_{loop}^{x_{loop}} w_{i+1}^{x_{i+1}}, \dots, w_l^{x_l} \mid (x_1, \dots, x_l) \in L, x_{loop} \in \mathbb{N}\right\}$ is contained in the language $\mathcal{L}(A) \supseteq \mathcal{L}'$ of the automaton and contains the original Elementary Bounded Language $\mathcal{L} \subseteq \mathcal{L}'$.

Proof Since the exponent x_{loop} of the new infix w_{loop} is independent from the others exponents $x_1, \dots, x_l$, restricting x_{loop} to 0 yields precisely the original language $\mathcal{L}$.

In order to show the inclusion in the language of the automaton $\mathcal{L}(A)$, we consider all words $w \in \mathcal{L}'$. These words are always of the form $w_1^{x_1} \dots w_i^{x_i} w_{loop}^{x_{loop}} w_{i+1}^{x_{i+1}} \dots w_l^{x_l}$. The word $w_1^{x_1}, \dots, w_i^{x_i} w_{i+1}^{x_{i+1}} \in \mathcal{L}'$ is also contained in $\mathcal{L}$. Thus, there is a run from the initial state q_0 to the state q reading $w_1^{x_1}, \dots, w_i^{x_i}$ and a run from the state q to an accepting state $q_f \in Q$. Since there is a run from q to q reading w_{loop}, we can construct a run that starts in q_0, reaches q for $x_{loop} + 1$ times, ends in the final state q_f and reads the word $w_1^{x_1} \dots w_i^{x_i} w_{loop}^{x_{loop}} w_{i+1}^{x_{i+1}} \dots w_l^{x_l}$. ■

It is an interesting property of this principle that the resulting Elementary Bounded Language might be in a different class than the original language: The Elementary Bounded Language $(aa)^* = \{a^{x_1} a^{x_2} \mid x_1 = x_2 \in \mathbb{N}\}$ is regular. If we add b^* between the existing infixes, we get the context-free language $\{a^{x_1} b^{x_{loop}} a^{x_2} \mid x_1 = x_2, x_{loop} \in \mathbb{N}\}$, which is not regular. Conversely, there is the context-free but not regular Elementary Bounded Language $\{a^{x_1} b^{x_2} \mid x_1 > x_2 \in \mathbb{N}\}$. However, adding b^* between the two infixes yields the regular language $\{a^{x_1} b^{x_{loop}} b^{x_2} \mid x_1 > x_2, x_{loop} \in \mathbb{N}\} = a^* b^*$.

A way to enforce regularity of the resulting Elementary Bounded Language lies in the choice of the place where to insert the loop: If a regular Elementary Bounded Language $\mathcal{L}$ can be split into two regular Elementary Bounded Languages $\mathcal{L}_1 \mathcal{L}_2 = \mathcal{L}$, then the concatenation $\mathcal{L}_1 w_{loop}^{x_{loop}} \mathcal{L}_2$ of the two languages and a regular language $w_{loop}^{x_{loop}}$ is still regular.

Another property worth noting is the preservation of the prefixes and suffixes of the runs: Lemma 5 requires all words to have a path in the automaton that passes state q after reading the corresponding part of the word. If these runs also admit a state q' with the same property, then this state q' remains a candidate for a second application of the lemma: Let q, q' be two states so that each word of the original Elementary Bounded Language admitted a run going through q and q' at a corresponding place in the words. Let the Elementary Bounded Language be modified by adding a loop reading w_{loop} at q as an infix. Then the new words are of the form $w' w_{loop}^{x_{loop}} w''$ and $w'w''$ is a word of the original Elementary Bounded Language. A new word can still be accepted by a run with the same prefix reading w' and suffix reading w''. Such a new run also passes q' when reading w' or w''.

We make use of those two properties in our approach: We start with a single run to an accepting state. The induced language is a single word and thus both regular and a Elementary Bounded Language. Then, we traverse this single run and try to find loops along the states of this run. While traversing the states along the single run, the Elementary Bounded Language can be split into a prefix and a suffix language. The prefix language is already build up of infixes that are not repeated and such that are repeated arbitrarily often. The suffix language contains a single word and is thus regular. Hence, regularity is preserved after adding the infix with the Kleene-star. Furthermore, the suffix of the initial run from the currently inspected state to the accepting state is still a suffix of at least one run for each word in the modified Elementary Bounded Language. Hence, Lemma 5 remains applicable while traversing the initial run and extending the Elementary Bounded Language.

Thus, the extraction of Elementary Bounded Languages works conceptionally as shown at the beginning of this section.

6 On Symbolic Language Representations

In the previous sections, we motivated the use and construction of Elementary Bounded Languages for fast intersection checks based on Parikh Images and Presburger Formulas. The idea was to uniquely identify a word by a vector denoting the repetitions of the infixes in the Elementary Bounded Language. As presented in Sections 4.2 and 4.1, we want to use Parikh Images in order to project the languages that we want to intersect to sets of such vectors.

Vectors in the Parikh Image denote the frequency of each letter. Thus, the Parikh Image is equivalent to the vector denoting the repetitions of the infixes if we represent each infix by a fresh letter: If the Elementary Bounded Language is of the form $\{w_1^{x_1} \dots w_l^{x_l} \mid (x_1, \dots, x_l) \in L\}$, then we represent it as $\left\{a_1^{x_1} \dots a_l^{x_l} \mid (x_1, \dots, x_l) \in L \subseteq \mathbb{N}^l\right\}$ where $a_1, \dots, a_l$ are different letters. If the Elementary Bounded Language is represented as a list of infixes and some representation of the set L, the transformation is quite easily done by replacing the list of words by a list of letters and by keeping the set L.

In this section, we will explain how to transform any regular language to this form.

We assume the language to be given as an automaton $A = (Q, q_0, \rightarrow, Q_F)$. The infixes $w_1, \dots, w_l$ of the Elementary Bounded Language shall be represented as the letters $a_1, \dots, a_l$. We construct a new automaton $A' = (Q, q_0, \rightarrow', Q_F)$ over the alphabet $\{a_1, \dots, a_l\}$. The new automaton copies all states and their acceptance behavior but it replaces the transitions in the following way:

$$\rightarrow' = \left\{ q' \xrightarrow{a_i} q'' \;\middle|\; q' \in Q, q'' \in Q, w_i \text{ s.t. there is a run } q' \xrightarrow{w_{i,1}} \dots \xrightarrow{w_{i,|w_i|}} q'' \right\}$$
$$\text{where } w_{i,j} \text{ is the } j\text{-th letter of } w_i$$

Hence, the new transitions are shortcuts for a sequence of transitions that is induced by the corresponding infix of the Elementary Bounded Language. Yet, many of the transitions and states are not necessary if they are not reached at the end of an infix of Elementary Bounded Language. Thus, the transitions are updated while traversing the automaton. This is done by the following steps:

1. Initialize a worklist with the initial state q_0 of the automaton A.
2. Inspect each state in the worklist until it is empty.
   a) Ignore states that have been inspected before.
   b) If the currently inspected state is q_{curr}, add

$$\left\{ q_{curr} \xrightarrow{a_i} q'' \middle| \text{ex. } q'', w_i \text{ with } q_{curr} \xrightarrow{w_{i,1}} \dots \xrightarrow{w_{i,|w_i|}} q'' \right\} \text{ as new transitions,}$$
$$\left\{ q'' \middle| \text{ex. } w_i \text{ with } q_{curr} \xrightarrow{w_{i,1}} \dots \xrightarrow{w_{i,|w_i|}} q'' \right\} \text{ to the worklist.}$$

In general, the modified automaton does not accept all words that are accepted by the original automaton. We will now characterize the language of the automaton after performing the presented modification that replaces sequences of transitions reading the infixes $w_1, \dots, w_l$ by transitions reading single letters $a_1, \dots, a_l$.

Lemma 6 (Language of Word-Transition Automaton) *Let $A = (Q, q_0, \rightarrow, Q_F)$ and $A' = (Q, q_0, \rightarrow', Q_F)$ with $\rightarrow'$ as defined earlier for $w_1, \dots w_l$ and $a_1, \dots a_l$.*

$$\mathcal{L}(A') = \left\{ w = a_{i_1} \dots a_{i_{|w|}} \mid w_{i_1} \dots w_{i_{|w|}} \in \mathcal{L}(A) \cap \{w_1, \dots, w_l\}^* \right\}.$$

Proof We prove the equality of the languages by showing the inclusion in both directions:

$\subseteq$: *Claim:* Any word $w \in \mathcal{L}(A')$ that is accepted by the automaton A' is of the form $w = a_{i_1} \dots a_{i_{|w|}}$ so that $w_{i_1} \dots w_{i_{|w|}}$ is a word of $\mathcal{L}(A)$.

We use an inductive argument: The initial state for both A and A' is q_0. If reading a_i in A' leads from q' to q'', then it does so for w_i in A as well because a transition $q' \overset{a_i}{\rightarrow'} q''$ was introduced if reading w_i leads from q' to q'' in A. Thus, if a word $a_{i_1} \dots a_{i_{|w|}}$ leads to an accepting state in A', the corresponding word $w_{i_1} \dots w_{i_{|w|}}$ leads to the same state in A and is accepted as well.

$\supseteq$: *Claim:* If any word $w \in \mathcal{L}(A)$ is accepted by the automaton A and of the form $w = w_{i_1} \dots w_{i_m}$, then the automaton A' accepts the word $a_{i_1} \dots a_{i_m} \in \mathcal{L}(A')$.

We apply a similar inductive argument that constructs runs to the same accepting state in both automata: The initial state for both A and A' is q_0. For any infix $w_i, 1 \leq i \leq l$ is read from the initial state q_0 in A, a transition with the same destination state is introduced in A'. If an infix w_i leads from q' to q'' in A, then a transition that reads a_i and leads from q' to q'' was inserted in A' if q' was ever on the worklist. Remember that the approach always adds states to the worklist when a transition with that state as the destination was added to the automaton. Hence, while constructing the run and reaching state q', it is known that this state is reachable from the initial state by a sequence of infixes. Therefore, it is also known that all possible transitions starting there have been added to A'. ■

Thus, the procedure constructs an automaton that only accepts a representation of $\mathcal{L}(A) \cap \{w_1, \dots, w_l\}^*$ rather than $\mathcal{L}(A)$ itself. This does suffice: The language will be intersected with the Elementary Bounded Language

$$\{w_1^{x_1} \dots w_l^{x_l} \mid (x_1, \dots, x_l) \in L\} \subseteq \{w_1, \dots, w_l\}^*.$$

Thus, the actual intersection certainly lies in $\{w_1, \dots, w_l\}^*$. Restricting an input language $\mathcal{L}(A)$ to this language does hence not alter the intersection.

In fact, the restriction is rather coarse. For the Elementary Bounded Language $d(abc)^*$, the procedure would still try to add a transition starting at the initial state that represents the infix *abc*. If the initial state is never visited again, then this transition is of no use as the intersection with the representation of $d(abc)^*$ will only keep transitions representing d at the initial state. Thus, the procedure can be adapted so that it does not only do the substitution of edges but also the intersection with the representation of the Elementary Bounded Language.

6.1 A Combined Procedure for Word-Transitions and Intersection with Elementary Bounded Languages

When an automaton reads words from an Elementary Bounded Language, there might be infixes that are not read from all states. Adding edges is rather costly because every edge requires to track all states to which reading an infix leads. We obtain this set of states by starting at the state q, reading the prefix letter-by-letter, and keeping track of the set of states where the current prefix of the infix can lead to. Once the whole prefix is read, we know all states that can be reached from q by reading this infix. In the worst case, each currently inspected state carries transitions to all states for the current inspected character. Therefore, we need to touch up to $(|w| - 1) \cdot |Q|^2 + |Q|$ transitions for each of the $|Q|$ states when adding transitions for an infix with $|w|$ characters.

Thus, we want to reduce the number of transitions we add: We only consider those transitions that can actually be taken when reading a word from the elementary bounded language. Conceptionally, we already compute an automaton for the intersection of the Elementary Bounded Language and the input automaton. We only compute those transitions that are needed for the intersection automaton.

The approach is a variant of the cross-product automaton: Each state of the resulting automaton represents a state of both automata. Here, we do not explicitly compute the crossproduct of two existing automata. We compute the needed parts of the automata on the fly as we are constructing the crossproduct automaton. The states and transitions for the Elementary Bounded Language are hard-coded in the procedure. We rely on a fixed subclass of Elementary Bounded Languages. The shortcut transitions of the input automaton are computed whenever needed and stored for multiple uses in the cross-product automaton.

The intuition is that we create multiple layers of the original automaton. Each layer denotes how many of the infixes of the Elementary Bounded Language have been read. Depending on that, different infixes are represented by the transitions.

We fix the subclass of Elementary Bounded Languages to those languages whose infixes occur either exactly once or arbitrarily often. Thus, the Elementary Bounded Language is assumed to be given as

$$\{w_1^{x_1} \dots w_l^{x_l} \mid (x_1, \dots, x_l) \in L\} \quad \text{with} \quad L = L_1 \times \dots \times L_l \quad \text{and} \quad L_i \in \{\mathbb{N}, \{1\}\}, 1 \leq i \leq l.$$

There are several observations about reading words of languages of this subclass of the Elementary Bounded Languages:

- Once an infix w_i was recognized, there may be no infixes w_j with lower indices $j < i$ because the Elementary Bounded Language enforces an order on the infixes.

- An infix w_i may only be immediately followed by infixes w_j if there is no infix $w_{j'}$ with a lower index $i < j' < j$ that has to occur, i.e. $L_{j'} = \{1\}$.
- A word may end right after an infix w_i if all other infixes $w_j, j > i$ are optional, i.e. $L_j = \mathbb{N}$.

These three observations are the basis for the conditions of the transitions in the automaton describing the intersection between the said Elementary Bounded Language and a finite automaton $A = (Q, q_0, \rightarrow, Q_F)$. In the intersection automaton, a state $q_{i,h}$ represents state q_i of the automaton A right after reading the representation of infix w_h. Intuitively, the resulting automaton can be seen as a multi-layered automaton where each layer contains a copy of the automaton. Whenever a different infix is read, the run switches to a higher layer. The height of a layer denotes the last infix that was actually read. The following automaton accepts a representation of the intersection of $\mathcal{L}(A)$ and an Elementary Bounded Language of the form as previously described. The infixes $w_1, \ldots, w_l$ are represented as the letters $a_1, \ldots, a_l$:

$$\begin{aligned}
A' &= (Q', q_{0,0}, \rightarrow', Q'_F) \quad \text{where} \\
Q' &= \{q_{i,h} \mid 1 \le h \le l, q_i \in Q, \quad \text{ex. a run } \; q_0 \rightarrow \cdots \rightarrow q_i \text{ of } A, \quad \text{reading } \; w_1^{x_1} \ldots w_h^{x_h} \\
&\qquad\qquad \text{with } \; x_1 \in L_1, \ldots, x_{h-1} \in L_{h-1}, x_h \in L_h \setminus \{0\}\} \\
&\quad \cup \{q_{0,0}\} \\
\rightarrow' &= \left\{ q_{j,h} \overset{a_{h'}}{\rightarrow} q_{j',h'} \mid q_{j',h'}, q_{j,h} \in Q', f(h) \le h' \le g(h), \; \text{ex. } q_j \rightarrow \cdots \rightarrow q_{j'} \text{ reading } w_{h'} \right\} \\
&\qquad f(h) = \begin{cases} h & \text{if } \; L_h = \mathbb{N} \\ h+1 & \text{if } \; L_h = \{1\} \end{cases}, \; g(h) = \min\left(\{h < h'' \le l \mid L_{h''} = \{1\}\} \cup \{l\}\right) \\
Q'_F &= \{q_{i,h} \in Q' \mid q_i \in Q_F, \quad \text{if f.a. } \; j \; \text{ with } \; h < j \le l : L_j = \mathbb{N}\}
\end{aligned}$$

The definition of Q' enforces that we only introduce states that are actually reachable. The the choice of the destinations of the transitions $\rightarrow'$ ensures that the transitions obey the first two of the three observations. These two definitions ensure that we do not insert a transition that is never taken while reading a word in the Elementary Bounded Language. This does not rule out all unnecessary transitions: We might still insert transitions that never lead to an accepting state as we cannot tell in advance where they lead to.

While the definition of the transitions is rather straight forward and mainly depends on comparisons of indices of the states, the definition of state set itself uses an argument on reachability. A procedure to obtain the whole automaton A' will be given now. It illustrates how an automaton according to the said definition can be obtained algorithmically.

1. The state set is initialized with the new initial state $Q' = \{q_{0,0}\}$. If the initial state $q_0 \in Q$ of A is an accepting state $q_0 \in Q_F$ and there is no required infix $w_i, L_i = \{1\}$, add $q_{0,0}$ as an accepting state.

2. We iterate through the infixes $w_1, \dots, w_l$.
3. The current infix w_i can be read after reading w_j where $j < i$ is the biggest index with $L_j = \{1\}$. If there is no such index, w_i can also be read initially at the beginning of the word, and j is set to 0. Thus, we initialize a worklist with all already known states $W = \{q_{s,j'} \mid j \leq j' < i, q_{s,j'} \in Q'\}$. Check the set L_i:

   $L_i = \{1\}$: Search for runs of A reading w_i that start with states q_s if there is a state $q_{s,j'} \in W$ in the worklist. Store those runs and add the transitions

   $$\left\{ q_{s,j'} \xrightarrow{a_i}{}' q_{s',i} \mid q_{s,j'} \in W, q_s, q_{s'} \text{ are conneceted by a run reading } w_i \right\}.$$

   Add all new transition goals states to Q'. If a state $q_{s'}$ is an accepting state $q_{s'} \in Q_F$ of A and there is no later necessary infix $w_{i'}, i' > i, L_{i'} = \{1\}$, mark it also as an accepting state $q_{s,i} \in Q'_F$.

   $L_i = \mathbb{N}$: Add transitions in the same way as for the other case but update the worklist by adding all states to it that are freshly introduced to Q'.
4. The resulting automaton is $A' = (Q', q_{0,0}, \rightarrow', Q'_F)$.

7 Parikh Images of Regular Languages as Presburger Formulas

In Section 4.2, we explain how to decide emptiness of intersections between Regular Languages and Elementary Bounded Languages using a decider for emptiness of intersections of semi-linear sets. The emptiness of such intersections can be decided by describing those semi-linear sets by Presburger Formulas, forming their conjunct and deciding its satisfiability. In Section 6, we constructed automata so that their Parikh Images are exactly the said semi-linear sets. This section explains how to obtain a Presburger Formula describing a Parikh Image of a given automaton.

The Verma-Seidl-Schmentick Construction [VSS05, Section 3] presents a procedure to obtain Parikh Images of Context-Free Grammars. The key concept of this procedure is counting the number of applications of each production rule: The procedure introduces a counting variable for each production rule. A constraint ensures that each non-terminal symbol is consumed by production rules as often as it is initially present or produced by production rules. The actual Parikh variables of the letters just sum up how often the applied production rules produce a certain non-terminal symbol.

An example: Let there be two rules $S \rightarrow R, S \rightarrow a$, which consume the start symbol S and either produce the terminal symbol a or the non-terminal symbol R. Let there be two more rules $R \rightarrow Rb, R \rightarrow b$, which consume the non-terminal symbol R and produce either the terminal symbol b, either with or without the non-terminal symbol R. The language is $a + b\,(b^*)$ and the correct Parikh Image is $\{(1,0)\} \cup \{(0,n) \mid n \in \mathbb{N} \setminus \{0\}\}$. If just the rule $S \rightarrow a$ is applied exactly once, then the constraint requiring the production and deletion of a non-terminal symbol to happen equally often is satisfied. This adds the Parikh vector $(1,0)$

A problem is the rule $R \rightarrow Rb$: This rule itself closes a cycle of rule applications. Whenever the rule is applied, it consumes and produces the non-terminal R equally often. This means that any number of applications of this rule does not violate the constraint as long as neither $S \rightarrow R$ nor $R \rightarrow r$ are applied. However, this should be impossible. Yet, it allows to count applications of $R \rightarrow Rb$ even if $S \rightarrow a$ instead of $S \rightarrow R$ is counted. This leads to the wrong Parikh Image $\{(1,n),(0,1+n) \mid n \in \mathbb{N}\}$.

An application of a rule should only be counted if its non-terminal symbol on the left side is produced as a consequence of a sequence of rule applications starting with the start symbol. As a way to do that, [VSS05] proposed an additional counting variable: The variable is similar to a distance measure of terminal and non-terminal symbols: Exactly the symbols that are produced by consuming the start symbol are allowed to be marked with 1. All other symbols are either marked 0 or with a non-zero value under specific conditions: There needs to be an applied rule with a non-zero marking for the consumed non-terminal symbol. The distance marking of the consumed symbol needs to be exactly 1 lower than the marking of the produced symbol.

Trying to Enforce The Distance Marking The distance marking is enforced by requiring all produced terminal symbols to be marked with a non-zero distance marking.

This does rule out the previous counterexample: If the resulting Parikh vector counts the terminal symbol b at least once, this symbol has to have a non-zero distance marking. In the previous example, both the terminal symbol a and the terminal symbol b are supposed to have a non-zero distance marking. The idea is that this marking implies a sequence of rule applications from the start symbol to all rules producing those terminal symbols. However, it is possible to consume and produce non-terminal symbols even if their distance marking is 0.

The problem is that the constraint was not strict enough: It only required the occurring terminal symbols to have a non-zero count without requirements on the rules that produced these symbols. There just needs to be one correct sequence of applied rules from the start symbol to a terminal symbol. Terminal symbols could still be produced as a consequence of a cycle of rule applications: If a cycle of non-terminal symbols only produces terminal symbols that are also produced as a consequence of rule applications from the start symbol, there is no constraint requiring the non-terminal symbols in the cycle to have a distance marking. Thus, the rule applications along the cycle are still counted although the involved non-terminal symbols are never produced from the start symbol.

Bypassing the Distance Marking A variant of the previous example illustrates the flaw: Let there still be the rule $S \to R$ that converts the start symbol to the non-terminal symbol R. We modify the other rule consuming the start-symbol: The new rule $S \to ab$ now also produces the terminal symbol b. The rules $R \to Rb, R \to b$ remain as before.

We count the rule $S \to ab$ once and are thus allowed to set the distance marking of the terminal symbols a and b to 1. All non-terminal symbols are hence marked and can be used. We mark the non-terminal symbol R with 0 and just apply the rule $R \to Rb$ n times. The zero marking is allowed as b is already marked with 1. Thus, the formula allows the wrong Parikh Image $\{(0,n),(1,n) \mid n \in \mathbb{N} \setminus \{0\}\}$. The correct Parikh Image should be $\{(1,1)\} \cup \{(0,n) \mid n \in \mathbb{N} \setminus \{0\}\}$.

Customizing the Verma-Seidl-Schwentick Method Technically, it is easily possible to convert any deterministic or non-deterministic finite automaton to a context-free grammar:

Each state in the finite automaton is represented by a non-terminal symbol. The initial state is the start symbol. Transitions of the form $q' \xrightarrow{a} q''$ are modelled as grammar rules: The rule $R_{q'} \to aR_{q''}$ consumes the non-terminal symbol $R_{q'}$ representing the state q' and produces the accompanying letter a followed by the non-terminal symbol $R_{q''}$ representing the next state q''. Accepting states also get a rule that just eliminates the non-terminal symbol. In this construction, applying rules always yields a word followed by the representation of a state that can be reached when reading the word. Conversely, every run induces a sequence of rule application that produces the read word and the last state of the run. If and only if the run is accepting, the non-terminal symbol can be deleted and the word is in the language of the grammar.

However, the said class of Presburger Formulas does not work as intended. As we mentioned earlier, the problem about loops of rule applications introduces additional satisfying variable assignments: The formulas accept more vectors than the actual Parikh Image. Thus, we devise a customized method to obtain a Presburger Formula for the Parikh Image. It differs from the Verma-Seidl-Schwentick solutions in two ways: The formula is generated directly from the finite automaton without using a Context-Free Grammar. Furthermore, this method also fixes the said issue about unrelated loops.

7.1 Verma-Seidl-Schwentick-like Formulas for Finite Automata

We construct a Presburger Formula for the Parikh Image of finite automata using a similar approach as the one shown in [VSS05]: We introduce variables to count how often transitions are taken and add constraints to ensure that we only count combinations that are induced by an actual run in the automaton. A possible Parikh vector can be tested by comparing each of its components to the number of times transitions reading the corresponding letter are taken.

A transition variable $x_{q',a,q''}$ counts how often a transition $q' \xrightarrow{a} q''$ is taken. Actual runs require that each state is entered as often as it is left. Thus, we add a constraint for each state that the sums of the transition variables for the incoming transitions and the sums for the outgoing transitions are equal.

Initial and accepting states are special cases: Outgoing transitions of the initial state are taken once more than its incoming transitions as each run starts there. We take this into account by increasing the sum of all incoming transition variables of the initial sate by 1. Similarly, one of the accepting states is entered once more than left. We introduce the variable f_q for all accepting states $q \in Q_F$. We add it to the sum of all outgoing transition variables of the corresponding accepting state. An additional constraint ensures that precisely one of these variables is 1 and all others are 0.

We also need to ensure that passing a transition is only counted if a sequence of traversed transitions leads to the corresponding state. To this end, we apply a similar technique as explained for context-free grammars:

We use state distance variables z_q for each state $q \in Q$. The initial state is always marked with 1. All other states may be marked with 0. A non-zero marking needs to be 1 higher than the marking of a predecessor. Here, a state q' is the predecessor of a state q'' if they are connected by a transition $q' \to q''$ that is counted as taken.

This state distance rating of each state hence expresses the length of a run from the initial state to that state so that each transition on the run is counted as taken. We use this rating to avoid the problem of unrelated cycles of transitions that are counted as taken: A constraint ensures that transitions may not be counted as taken if they concern a state with a zero distance rating.

Combining the gathered insights, we formalize the constraints to a Presburger Formula:

Definition 7 (Parikh Image Presburger Formula) Let $A = (Q, q_0, \rightarrow, Q_F)$ be a finite automaton over the alphabet Σ. The Parikh Image Presburger Formula of A is:

$$\mathcal{P} \equiv \sum_{q \in Q_F} f_q = 1 \wedge z_{q_0} = 1 \bigwedge_{(q,a,q') \in \rightarrow} 0 \leq x_{q,a,q'} \bigwedge_{a \in \Sigma} p_a = \sum_{q \xrightarrow{a} q'} x_{q,a,q'} \tag{1}$$

$$\bigwedge_{q \in Q_F} 0 \leq z_q \wedge 0 \leq f_q \leq 1 \wedge \sum_{q \xrightarrow{a} q'} x_{q,a,q'} + f_q = \sum_{q' \xrightarrow{a} q} x_{q',a,q} + \begin{cases} 0 & q \neq q_0 \\ 1 & q = q_0 \end{cases} \tag{2}$$

$$\bigwedge_{q \in Q \setminus Q_F} 0 \leq z_q \wedge \sum_{q \xrightarrow{a} q'} x_{q,a,q'} = \sum_{q' \xrightarrow{a} q} x_{q',a,q} + \begin{cases} 0 & q \neq q_0 \\ 1 & q = q_0 \end{cases} \tag{3}$$

$$\bigwedge_{q \in Q \setminus \{q_0\}} \begin{pmatrix} \bigvee_{q' \xrightarrow{a} q} (z_{q'} + 1 = z_q \wedge z_{q'} > 0 \wedge x_{q',a,q} > 0) \\ \vee \left(z_q = 0 \bigwedge_{q' \xrightarrow{a} q} x_{q',a,q} = 0 \bigwedge_{q \xrightarrow{a} q'} x_{q,a,q'} = 0 \right) \end{pmatrix} \tag{4}$$

Usually, we specify assignments of the formula by just assigning values to the variables $p_a, a \in \Sigma$. All other variables are assumed to be bound by existential quantifiers. □

Line (1) of this formula ensures basic properties of the variables: Exactly one of the variables for the accepting states is 1. The distance marking of the initial state is 1. All transition variables are non-negative. The Parikh variables p_a, f.a. $a \in \Sigma$, count the frequency of letters by summing up how often corresponding transitions are taken.

The flow constraints are enforced by line (2) and line (3): Each state is entered as often as it is left. As explained earlier, this constraint is not satisfied for the last state of an accepting run and the initial state. Thus, the flow equations for an initial or accepting states have slightly modified sums: Line (2) adds the accepting state variable to the sum of outgoing transitions of the accepting states. As ensured by line (1), exactly one of those variables is 1 and compensates that the final state is entered once more than left. Similarly, both lines add 1 to the sum of incoming transitions of the initial state.

Line (4) sets up the distance marking and enforces that transitions are only counted if all transitions on a run from the initial state to these transitions are counted as well: We have two options to assign a distance marking to a state: Either, we assign 0 to it and do not count any incoming or outgoing transition as taken. Or, we pick a state with a non-zero marking, follow a transition that is counted as taken and assign a distance marking to the next state that is increased by 1.

We will first show that the formula has a satisfying assignment for each vector in the Parikh image:

Lemma 7 (Parikh Vector to Assignment) *Let $A = (Q, q_0, \rightarrow, Q_F)$ be a finite automaton over the alphabet $\Sigma = \left\{a_1, \ldots, a_{|\Sigma|}\right\}$ and $\mathcal{P}$ its corresponding Parikh Image Presburger Formula.*
If there is a word $w \in \mathcal{L}(A)$ and a vector $\vec{p} = \left(|w|_{a_1}, \ldots, |w|_{a_{|\Sigma|}}\right) \in \mathcal{P}(\mathcal{L}(A))$, then there is a satisfying assignment with $p_{a_1} = |w|_{a_1}, \ldots, p_{a_{|\Sigma|}} = |w|_{a_{|\Sigma|}}$.

Proof If there is a vector $\vec{p}$ in the Parikh image, then there is also a corresponding accepting run of A that reads the word w. We construct an assignment for the variables of $\mathcal{P}$:

- We assign the values of the Parikh vector $\vec{p}$ to assignment of the Parikh variables of $\mathcal{P}$: $p_{a_1} = |w|_{a_1}, \ldots, p_{a_{|\Sigma|}} = |w|_{a_{|\Sigma|}}$
- The variable of the final state f_q is set to 1 for the last final state q in the run. All other variables of final states $f_{q'}, q' \in Q_F \setminus \{q\}$ are set to 0.
- The transition variables $x_{q,a,q'}, q \xrightarrow{a} q'$ are set to the number of occurrences of the corresponding transition in corresponding run.
- The state distance variables $z_q, q \in Q$ are set along the run: The initial state has the distance marking z_{q_0}. We follow the transitions of the run. Whenever a state has not been marked with a distance yet, we assign the previously seen or assigned distance marking and increment it by 1. All other states are marked with 0.

This is clearly an assignment with $p_{a_1} = |w|_{a_1}, \ldots, p_{a_{|\Sigma|}} = |w|_{a_{|\Sigma|}}$. It also satisfies all constraints of the formula $\mathcal{P}$:

Line (1): The basic constraints are satisfied:

- The sum of the variables of the accepting state is 1 since exactly one of them, namely the variable of the last state of the accepting run, was set to 1 while all others are 0.
- The distance marking of the initial state is set to 1.
- Transition variables are set to the number of occurrences in the run and are hence non-negative.
- The Parikh variables are set according to the frequencies of the letters in the word that is read by the accepting run. Thus, each Parikh variable is equal to the sum of all occurrences of transitions reading the corresponding letter in the accepting run.

Line (2): States in the run, other than the first and the last one,are surrounded by an incoming and an outgoing transition. Thus, the states have equally many incoming and outgoing transitions if we do not count the first and the last transition of the run. The additional outgoing transition of the initial state and the incoming transition of the last state in the accepting run are compensated by adding 1 to the other side of the equation. Thus, those flow equations are satisfied.

Line (3): The only difference between flow equations for accepting and non-accepting states is the final state variable. Since non-accepting states cannot be the last states of a run, we can safely leave those variables out and the constraint remains satisfied.

Line (4): The constraint for the distance marking addresses states except for the initial state. If a state is not in the accepting run, it is marked with 0 and the the last part of the disjunct is satisfied. Otherwise, the state q was marked while traversing the accepting run and the state inherits the successor of the marking of a state q'. The state q' occurs in the run right before the first occurrence of q. Thus, there is a transition from q' to q that is taken, q' has a non-zero distance marking and the marking of q is the direct successor in the natural numbers. The constraint is hence satisfied. ■

We will now show that each satisfying assignment of the formula corresponds to a vector in the Parikh image:

Lemma 8 (Assignment to Parikh Vector) *Let $A = (Q, q_0, \rightarrow, Q_F)$ be an automaton over $\Sigma = \left\{a_1, \ldots, a_{|\Sigma|}\right\}$ with the Parikh Image Presburger Formula $\mathcal{P}$.*
For each satisfying assignment of the formula, there is a word $w \in \mathcal{L}(A)$ and a vector $\vec{p} = \left(|w|_{a_1}, \ldots, |w|_{a_{|\Sigma|}}\right) \in \mathcal{P}(\mathcal{L}(A))$ so that the variable assignment corresponds to the vector $\vec{p}$, i.e. $p_{a_1} = |w|_{a_1}, \ldots, p_{a_{|\Sigma|}} = |w|_{a_{|\Sigma|}}$.

Proof Each satisfying assignments specifies how often each transition is taken. It also specifies how often each character is read. The constraint in line 1 of the formula ensures that those values are not contradicting. It remains to show that there is a run of the automaton that takes each transition as often as specified.

The correctness of the original formula for context-free grammars was shown using a result on communication-free Petri nets as shown in [Esp97, Theorem 3.1]: Let there be a specification how often each transition in a Petri net shall be applied. There is a sequence of applications of Petri net transitions if the specification satisfies two requirements: There is no place where the specified applications of transitions consume more tokens than produced or given by the initial marking. Every applied transition needs to be reachable along a sequence of applied transitions from a place that was initially marked.

A finite automaton is a special case of a communication-free Petri net: Each state corresponds to a place in the Petri net. The only existing token marks the current state of a run. The initial marking sets the token to the place corresponding to the initial state. Petri net transitions move a token from a state to another state if there is a transition in the automaton in the same direction.

A sequence of applications of Petri transitions can directly be translated to a run in the automaton. Thus, the said result allows to conclude the existence of a run from a certain specification. In order to apply this result, we need to show that its two requirements hold for the Petri net that is implied by the satisfying assignment:

- In sum, transitions may not consume more tokens from a place than they and the initial marking place there. This is guaranteed since we ensured that each state is entered as often as it is left.

- There has to be a path of applied transitions that transports a token from an initially marked place to each place where transitions shall be applied. In this special case, only the place corresponding to the initial state is marked. Thus, we will show that all satisfying assignments only count transitions as used if they are at the end of a whole path of used transitions starting from the initial state. ■

We consider each transition $q \xrightarrow{a} q'$. If the variable assignment sets the transitions count $x_{q,a,q'} \neq 0$, then the state distance markings $z_q, z_{q'}$ may not be 0 since the last part of line (4) is the only case that allows a zero distance rating. This part of the formula cannot be satisfied if an adjacent transition is used. The other parts of this line ensure that the marking is either 1 for the initial state or inherited from a predecessor state along an edge that is counted as used. When tracing back the distance marking, it always needs to decrease by 1 for each transition until the initial state is reached. Thus, the required paths of used transitions from the initial state to each used transition do exist.

Hence, the said result on communication free Petri nets applies and a sequence of applications of Petri net transitions with the said frequencies of each transition does exist. Thus, there is a run in the automaton reading a word with the Parikh image that is specified by the satisfying assignment of the formula. The run is accepting as line (2) ensures that precisely the accepting state q with $f_q = 1$ is entered once more than left. This means that this state is the last one of the run.

Remark: The said Petri net construction from the original proof for context-free grammars is not necessary for finite automata. Unlike context-free grammars, the regular case can also be modelled graph-theoretically:

Each state is modelled as a vertex of the (multi-)graph, each application of a transition is modelled as a directed edge between the vertices that correspond to the involved states. We search for an Euler-Walk from the initial state to the final state q with $f_q = 1$.

Euler-Walks use every edge in the graph precisely once. The result is a sequence of edges where every edge is used exactly once and the goal vertex of each edge is the start vertex of the next one. This directly corresponds to an accepting run of the automaton.

A characterization of graphs containing Euler-walks and Euler-cycles was first shown in [HW73]. The conditions are graph theoretical reachability and the equality of the number of incoming and outgoing edges. The rest of this variant of the proof is quite similar to the variant using communication-free Petri nets.

We conclude the correctness of the formula:

Corollary 2 *The Parikh Image Presburger Formula $\mathcal{P}$ characterizes the Parikh Image of the corresponding finite automaton.* □

An easy example illustrates the usage of this formula:

Example 3 Let $A = (Q, q_0, \rightarrow, Q_F)$ be a finite automaton with two states $Q = \{q_0, q_1\}$ where q_0 is the initial and the only accepting state $Q_F = \{q_0\}$. Reading a from the initial state q_0 leads to q_1, reading b from there leads back to q_1, i.e. $\rightarrow = \{(q_0, a, q_1), (q_1, b, q_0)\}$. Figure 8 represents this automaton graphically.

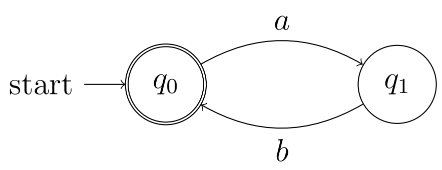

Figure 8: An automaton accepting the language $(ab)^*$.

The Parikh Image is $\mathcal{P}\left(\mathcal{L}\left(A\right)\right) = \{(n,n) \mid n \in \mathbb{N}\}$. We apply the Parikh Image Presburger Formula for this automaton and get:

$$\mathcal{P} \equiv f_{q_0} = 1 \wedge z_{q_0} = 1 \wedge 0 \leq x_{q_0,a,q_1} \wedge 0 \leq x_{q_1,b,q_0} \wedge p_a = x_{q_0,a,q_1} \wedge p_b = x_{q_1,b,q_0} \tag{5}$$
$$\wedge\, 0 \leq z_{q_0} \wedge 0 \leq f_{q_0} \leq 1 \wedge x_{q_0,a,q_1} + f_q = x_{q_1,b,q_0} + 1 \wedge 0 \leq z_{q_1} \wedge x_{q_1,b,q_0} = x_{q_0,a,q_1} \tag{6}$$
$$\wedge\, ((z_{q_0} + 1 = z_{q_1} \wedge z_{q_0} > 0 \wedge x_{q_0,a,q_1} > 0) \vee (z_{q_1} = 0 \wedge x_{q_0,a,q_1} = 0 \wedge x_{q_1,b,q_0} = 0)) \tag{7}$$

Line (6) contains the flow constraints of Line (2) and (3) of the general formula. This line mainly enforces the equality $x_{q_1,b,q_0} = x_{q_0,a,q_1}$. Since line (5) of the example formula sets the Parikh variables p_a and p_b as synonyms of the corresponding transitions, this implies the equality $p_a = p_b$.

The state distance variable z_{q_0} of the initial state is fixed to 1. The state distance variable of the other state is either

$z_{q_1} = 0$: In this case, line 7 enforces $x_{q_0,a,q_1} = x_{q_1,b,q_0} = 0$. This yields a satisfying assignment corresponding to the Parikh vector $(0,0)$.

or

$z_{q_1} = 1$: Line 7 implies that the predicate $z_{q_0} + 1 = z_{q_1}$ needs to be satisfied. Since Line 5 already set $z_{q_0} = 1$, a satisfying assignment needs to set $z_{q_1} = 2$. Furthermore, the transition $q_0 \xrightarrow{a} q_1$ needs to be taken, i.e. $x_{q_0,a,q_1} > 0$. Due to the already mentioned requirement $x_{q_1,b,q_0} = x_{q_0,a,q_1}$, this yields Parikh vectors of the form $(n,n), n \in \mathbb{N} \setminus \{0\}$.

Thus, the formula accepts precisely those assignments corresponding to the Parikh vectors in $\mathcal{P}\left(\mathcal{L}\left(A\right)\right) = \{(n,n) \mid n \in \mathbb{N}\}$. $\square$

8 Notes on the Implementation

This section presents our implementation of the procedure we presented in the previous sections. The implementation provides a library for the several algorithms that were introduced. It also offers a command line interface to run the CEGAR loops on randomly generated instances.

The source code is written in F#. Hence, it relies on the Mono/.net framework. This open-source programming language was initially introduced by Microsoft in 2005. It is currently maintained by the F# Software Foundation. Software using the Mono/.net framework is platform-independent: It runs on multiple processor architectures and operating systems. It can use any libraries for the Mono/.net platform independent of their programming language.

The main programming paradigm of F# is functional programming. In 2013, many software developers in Microsofts *Most Valuable Professionals (MVP)* program switched to functional programming and hence to F#, according to [Fan14, p. xv]. For this work, this programming language was not only chosen for being an increasingly popular programming language but also for interoperability with other projects:

Existing verification projects like the framework Averest and its programming language Quartz are programmed using F#. This allows to easily integrate this work in Averest. Furthermore, some existing frameworks for Mono/.net and F# were used in this work:

- *Fare (Finite Automata and Regular Expressions).* A Mono/.net library offering basic data structures and algorithms on finite automata. It was released by the Greek computer programmer Nikos Baxevanis who is also known for the F#-based automated testing tool “LightCheck”. Fare is heavily inspired by the Java-Library `dk.brics.automaton` by Anders Møller (Aarhus University, Denmark).

  Our library uses the data structures of Fare as the input format for automata. Fare is also used for standard operations on non-deterministic finite automata, such as intersection, union and complement. The ability to convert a regular expression to an automaton is used when creating finite automata from Elementary Bounded Languages.

- *Microsoft Research Z3.* A solver for satisfiability modulo theories written in C++. In our implementation, it is used to check the satisfiability of the Presburger Formula for the the fast inclusion check.

- *z3fs.* An F#-interface to Z3 by Anh-Dung Phan. It allows writing formulas for Z3 in native F# syntax. In our implementation, it is used when generating the Parikh Image Presburger Formula for a given automaton.

Note that the binaries of Z3, as it is written in C++, are not platform-independent. This means that our implementation always needs to be shipped with the `.so`-file of Z3 for the current architecture and operating system.

We now present a short overview on a selection of details of the implementation. More details on the API can be found in the full API specification. It can be generated from the source code since the functions are documented using the built-in XML Documentation feature of F#. We start with a short introduction to the data structures. Especially the data structure for automata as provided by Fare is important since it is the main input format for our library. After introducing the data structure, we will present the modules of the program and their role in the presented algorithm.

8.1 Data Structures for Finite Automata

Many of the introduced algorithms work on finite automata. Thus, data structures for finite automata are frequently used throughout almost all parts of the implementation. All used and implemented data structures are strongly tied to the ones offered by Fare. Although Fare uses whole Unicode as the alphabet of the automata, the input languages need to use an alphabet that is a subset of a `a-z`. The allowed range of Unicode characters is defined in the file `AutomataGenerator.fs` and can be altered provided that all characters in that range are legal letters in regular expressions.

The data structures in Fare are state-centric. This means that data is stored per state and bound to the object representing the state. Therefore, the view on automata is quite similar to the one that is implicitly used when reading a word: We move from state to state. The only needed information is which states are the next ones along transitions reading the current letter. Hence, the most important data structures in Fare are `Fare.State`, `Fare.Transition`, and `Fare.Automaton`.

`Fare.Automaton` Fare stores an absolute minimum of information with each object of the type `Fare.Automaton`: It stores a flag for the determinism as well as the initial state. All other information is stored together with the states themselves.

An application for this kind of representation is concatenation: When concatenating two automata, it suffices to copy all states from the first automaton and add transitions to the initial state of the second automaton. The second automaton will neither be altered nor copied at all. The automaton accepting the concatenation of both languages reuses exactly the same data for the states of the second automaton.

`Fare.Transition` Transitions are stored carrying the target state and a range of letters: An object of the type `Fare.Transition` represents a whole set of transitions with the same start and goal state. If there are transitions $q \xrightarrow{a_1} q', \ldots, q \xrightarrow{a_n} q'$ where $a_1, \ldots, a_n$ is a sequence of consecutive Unicode-characters, then these transitions are represented by a single object of `Fare.Transition`: The attribute `Fare.Transition.min` is set to a_1 and `Fare.Transition.max` to a_n. The transition can be taken in a run if the current Unicode character is not less than a_1 and not greater than a_n in the order of Unicode characters. We implemented functions that simplify transitions by uniting overlapping or adjacent intervals.

`Fare.State` The state itself has an attribute denoting whether the state is an accepting state. The advantage is that this property can be checked in constant time. The disadvantage is that a variant of an automaton with a different acceptance behavior needs to be stored as a complete copy as the acceptance behavior cannot be separated from the states in this model.

Each state is annotated by a constant ID. The ID is unique in the program since there is a global integer variable for the next state ID. Each construction of a `Fare.State` object copies that ID as the current ID and increases the global value. We heavily rely on this property in our implementation: We use the ID of a state as a primitive key for data structures. Other than composed values like strings or recursive types, primitive types allow comparison in constant time, which is needed when accessing values by a key.

A state may also be annotated by an attribute called `Fare.State.Number`. This mutable integer value can be assigned and changed. The class `Fare.Automaton` provides a method `Fare.Automaton.SetStateNumbers` that assigns the numbers $0, \ldots, n-1$ uniquely to each state in a set of n states. Fare uses this internally to store states in arrays: The numbers are the keys in the array. This allows to only allocate an array of exactly the size of the set while being able to access the elements in constant time. It is also possible to use the IDs as the keys of an array. This, however, requires to allocate an array in the size of at least the interval from the lowest to the highest ID in the set.

The state also carries a `Fare.Transition` list that denotes all transitions starting from this state. Checking for a specific goal state or a specific Unicode character always requires searching in the whole list.

8.1.1 Custom Data Structures

As already mentioned, there are cases when the given data structures do not allow the fastest possible way of obtaining or manipulating certain data. The biggest issues for our implementation were:

Access Specific States If an object of the type `Fare.Automaton` is given, then the only way to get access to a specific object of the type `Fare.State` is traversing the whole automaton. In the worst case, we need to look at all states and their transitions each time we want to access a specific state.

Accessing specific states is especially crucial in our algorithm when we over-approximate an automaton: We need to compare each pair of states. Furthermore, whenever we unite two states, we need to access those states in order remove them and transfer their properties to the new state.

Incoming Transitions If we want to get transitions of a state, we need to get at least the corresponding object of the type `Fare.State`. This, however, only works for outgoing transitions. If we want to know all incoming edges of a specific state, we need to traverse the whole automaton and check for every single edge whether it leads to the current state.

Incoming transitions are for example needed to be known when computing the similarity rating for a pair of states. Another example is the Parikh Image Presburger Formula that relies on knowledge about incoming edges of each state.

Filtering Transitions All objects of the type `Fare.Transition` starting from a state are organized in a single list. This means that we need

> to search the whole list for transitions leading to a specific state or reading a certain letter.
>
> While the implemented algorithms do not only need to filter transitions by their letters, they do need to group letters by the states: For example, we compare two states by the number of mismatching transitions. To this end, we need to count for each successor state how many character are read uniquely in transitions from one of the start states. Another application of finding transitions with the same goal is the simplification of intervals: We unite two objects of the type `Fare.Transition` to a single one if the letter intervals are adjacent or overlapping.

We designed data structures that organize states and transitions in a different way. We build a redundant data structure once for the automaton and keep it updated when manipulating the automaton. There are two main applications for our custom data structure: Comparing and reducing states as described in section 3 and extracting the Parikh Image Presburger Formula as described in section 7.

Most collections in our data structure are maps that use the ID of a state as the key. As already mentioned, the mutable attribute `Number` can be used to assign unique, consecutive numbers to a state that serve as the key in an array. However, a problem arises when some methods in `Fare.Automaton` are called: They also use the numbers in order to produce arrays. This might override values that are required by our data structure. For example, if we follow a transition, read the attribute `Number` of the goal state, and try to access the corresponding state in the array, we might get a wrong object because a method like `Fare.Automaton.GetSortedTransitions` may have called `Fare.Automaton.SetStateNumbers` to reassign numbers.

Besides storing the states itself, there are other uses for data structures that use the state ID to organize data in a map: When rating the similarity of a pair of states, we organize those ratings in a map. The keys are tuples of the two involved state IDs. In order to have unique keys, the first ID has to be the lower one.

One of our custom types is `AnnotatedState`. It has a property `State` that carries the original object of the type `Fare.State`. The attributes `Intrans` and `Outtrans` are both maps from `int` to lists of tuples of two Unicode characters.

The intended semantics are: The integer value is the `Fare.State.Id` of the neighbor state. If the value is a key in `AnnotatedState.Intrans`, then there are incoming transitions from the state with that ID. Similarly, `AnnotatedState.Outtrans` groups the lists of outgoing transitions by the ID of neighbor states. The value of those maps is an interval list: The first character in the tuple denotes the beginning, the second one the end of and interval of Unicode character for which there is a transition connecting the current state and the state corresponding to the key in the map.

A map in F# is essentially an AVL tree. AVL trees are self-balancing trees that were introduced by [AVL62]. Typical operations like the insertion, the deletion, and the retrieval of data have a worst-case runtime of $\mathcal{O}(\log n)$. Hence, initializing the data structure with n states takes

$$\sum_{k=1}^{n} \mathcal{O}(\log(k)) = \mathcal{O}(\log(n!)) < \mathcal{O}(n \cdot \log(n))$$

steps. Additionally, traversing the whole automaton requires in a worst case to touch every single state and transition object. The worst-case runtimes of the named three common use cases are:

Access Specific States $\mathcal{O}(\log n)$ Accessing the `AnnotatedState` object for a given ID just takes a single read access on the map. The original data structure requires traversing the whole automaton for every single access.

Incoming Transitions $\mathcal{O}(1)$ Assuming that the `AnnotatedState` object is available, we only need to use its property `Intrans` in order to obtain the incoming transitions.
$\mathcal{O}(\log n)$ Assuming that we have to retrieve the object of the type `AnnotatedState`, we need a query to the map that stores all states of the automaton.

Filtering Transitions $\mathcal{O}(\log k) + \mathcal{O}(\log n)$ (for k neighbor states and n states in total) If the `AnnotatedState` object is given, we just need to query the map `Intrans` or `Outtrans`. Getting access to the `AnnotatedState` object takes again a read access in the map containing all states.

The given data structures do provide a faster and more convenient way to access and store data for the said use cases. Note that we chose a purely functional approach and hence immutable data structures – although `Fare.Automaton` itself uses mutable data structures. Therefore, the runtime for write access and the initialization relies on a compiler that works well and reuses the read-only data. We tried to support this with our coding style:

Loops are mostly implemented using built-in methods like map, reduce, and fold, which apply a given function on each element in a given collection. There are a few cases where those methods are not suitable. In these cases, we implemented recursive functions. All our recursive functions are tail-recursive. This means that the recursive function call is the last call of the function. Hence, the recursive call may safely modify, reuse, and override any data that was produced by the calling function.

We now introduce our implemented modules and their main functions. These introductions serve as quick overview and as a guide where to find which implementation of the implemented algorithms. For a more detailed explanation on every function and precise usage instructions, we refer to our source code and the contained documentation for each of the functions.

8.2 The Module `Intersect`: Over-Approximating the Intersection

The module `Intersect` implements the algorithms we motivated and introduced in Section 3. Its main purpose is to generate a finite automaton that accepts an over-approximation of the intersection of the languages of a list of automata.

We implemented the function `IntersectAutomataList` for this purpose. The input is a list of `Fare.Automaton` objects. The result is a single object of this type. The function itself splits the work into many tasks of intersecting and over-approximating two automata at once. This is done in a boustrophedonic way:

An inner recursive function is called with the input list and an empty list. The inner function calls another function to intersect and over-approximate two automata. The result will be added at the beginning of the second list. The processed two automata will be removed from the beginning of the first list and the recursive function calls itself. Once the first list is empty, the second list takes its place.

This implements the tree-like chaining as introduced in Section 3 and Figure 4. Adding the resulting automata to the beginning of the list instead of the end of the list does not only save the need to find the end of the list. It also reverses the result list automatically. In the tree-metaphor, this means that the layers of the tree are processed boustrophedonically.

This matters when the length of the input list is not a power of two. Then, there is a layer with an odd number of automata. The automaton that is not intersected in this layer will be the first one to be intersected in the next layer. This addresses the problem of automata that are quite seldom intersected and over-approximated.

The function `IntersectTwoAutomata` is used for the actual intersection and over-approximation. This function is the place where the intersection and the Hopcroft-minimization is done and the desired number of states is set. The actual reduction of states is delegated to the function `reduceastates`.

This function `reduceastates` initially calls the function `getSimilarityRating` to initialize the custom data structure containing the states and a table of all similarity ratings of pairs of states. The function then determines the best pair of states to be united using `findmostsimilarstatepair` and updates the similarity rating and the two representations of the automaton using `uniteStates`.

The step of uniting states is independent from the similarity rating: The function `uniteStates` just unites a pair of states that is given as a parameter. The update of the similarity ratings is delegated to an external function. This allows to independently change the way how states are selected to be united.

An interesting detail about the implementation is the similarity rating using the representation of transitions by `Fare.Transition`. The algorithm we discussed earlier simply counted the number of mismatching unique transitions of two states. The type `Fare.Transition` represents many transitions as a single object since these objects are not annotated with single letters but whole intervals in the Unicode alphabet.

Hence, we have two goals: We need to compare whole intervals of letters rather than single letters. If characters occur in multiple intervals, we may not count them twice. The first useful trick is our custom data structure. It allows to easily access incoming and outgoing transitions by the ID of the neighbor state. These transitions are given as interval lists. We can count the mismatching transitions of two states by comparing two interval lists for the same neighbor state ID. This is done by linearly parsing two sorted interval lists and summing up the lengths of exclusive subintervals. Since we require sorted lists either way, we can easily prevent double counting: We linearly pre-process the already sorted interval lists and unite each pair of overlapping intervals to a single one.

8.3 The Module `CounterExample`: Extracting an Elementary Bounded Counterexample Language

The module `CounterExample` implements the procedure we motivated in Section 4.1 and introduced in Section 5. Its purpose is to derive a regular Elementary Bounded Language consisting of potential counterexamples. The counterexamples are words that are in the over-approximation of the intersection and violate the specification. We assume the input to be given as a `Fare.Automaton`. The output is a representation of the Elementary Bounded Language as a list of tuples. The first component of the tuple is an infix of the Elementary Bounded Language. The second component is a Boolean value and denotes the number of possible repetitions of the infix: An infix with `false` as the second component has to occur precisely once. An infix with `true` can occur arbitrarily often. Intuitively, the Boolean value answers the question whether there is a Kleene star right of the infix in the representation as a regular expression.

The function for this purpose is `FindCounterExampleEBL`. It returns an `option` type: It uses a variant of a depth-first search on the automaton to find a path to an accepting state. If there is no such path, the return value is `None`. Otherwise, it is `Some outputEBL` where `outputEBL` is a regular Elementary Bounded Language that is contained in the language of the input automaton. It uses the representation as described above. The Elementary Bounded Language is obtained by splitting the counterexample word and adding looped infixes:

We start with the initial counterexample path and process every single state on the path. From every state, we try to find a loop back to the state itself. The infix that is read along this path is added as an Elementary Bounded Language infix with a Kleene star. The search for the loop is again implemented using the modified depth-first search.

We decided that the initial path may not be the empty path: If we apply the said procedure on just the empty path, the result is just of the form w^* with w a single word. Instead of the empty path, the run reading w could be chosen as the initial path. The language $w^* \setminus \{w^0\}$ would still be a subset of the resulting language as the loop can still be found at the initial state.

This design decision requires to treat the empty word separately: Since the empty word is not in any of the resulting languages, we need to check whether the initial state is an accepting state. The module offers the function `IsTheEmptyWordACounterexample`. Its first parameter is the language of all possible counterexamples as an object of the type `Fare.Automaton`. Its second parameter is the list of the `Fare.Automaton` objects whose intersection we want to determine. The function checks whether the initial states are accepting in the automaton of potential counterexamples and all automata that shall be intersected. The result is returned as a value of the type `bool`. A positive result is a proof that the empty word is a counterexample. A negative result means that the empty word is only a spurious counterexample. It can then be removed from the language of potential counterexamples. To this end, the module provides the function `EliminateAcceptingInitialState`. If the initial state is accepting, the function modifies the input automaton by setting a copy of the former initial state as the new initial state. The new copy is not accepting and has no incoming transitions. The incoming and outgoing transitions of the former initial state remain unchanged. The new initial state inherits all outgoing transitions of the former initial state.

The Custom Depth First Search A key technique in this procedure is the frequent use of a depth first search on the automaton. To this end, we have implemented a variant of the depth first search in the function `DepthFirstSearch`. The implementation is strictly functional, tail recursive and uses immutable data structures. The search runs directly on an object of the type `Fare.Automaton`. The automaton, its states and its transitions are never modified throughout the search.

The most important parameters when calling `DepthFirstSearch` are the function parameter criterion and the list of `Fare.State` objects `backwardpath`. Initially, the list `backwardpath` is a reversed representation of the beginning of the path that is already known. Typically, it is initialized with a list consisting only of the start state of the desired path. Throughout the recursive calls, the list contains the reverse prefix of the path that is currently examined. Hence, the list is also used for backtracking once all neighbor states of the current state, which is the first state in the list, have been visited. The backtracking step is done by just omitting the current first state of the list and calling the function recursively with the tail of the list. Since the list is organized reversely, adding and removing the current state can be done in constant time. A suitable path is found, once the current state satisfies `criterion`. This means that the function `criterion` returns `true` for the current state. The path is returned by just reversing `backwardpath` in linear time. Once all neighbor states of the start state have been examined without success, we know that there is no state satisfying the criterion and the function returns `None`.

We keep track of the visited states using the parameter `visitedstates`. It is a set of `Fare.State` objects and marks all states that have been visited. Initially, the function should be called with `visitedstates` being an empty set. Since sets in F# are stored as AVL trees, this produces a logarithmic overhead when checking or storing whether a state has been visited. A way to reduce the overhead is implemented in the Boolean parameter `revisit`: The parameter marks whether the current state had already been added to the `visitedstates` set before. This happens when a state is visited again during backtracking. The parameter must initially be set to `false` if the first state is initially not included in the set `visitedstates`. Setting `revisit` to `true` is safely possible and saves one redundant write access to the set `visitedstates` if the first state is initially included in this set.

As we prevent states from being revisited, this means that the paths are actual paths and never loops. However, we still use the depth first search to close the loops in order obtain infixes with Kleene stars for the Elementary Bounded Language. This is possible by adjusting the criterion for the search: The criterion does not check whether the tested state itself closes the loop but whether the state has an outgoing transition to a state closing the loop. The missing state is then added to the path that was found in the depth-first search. This however does not find self-loops since the result of the depth-first search cannot be an empty path. Thus, self-loops are checked separately.

Allowing More General Criteria As another way to speed up the depth-first search, we decided to widen the set of states that is accepted by the criterion of the depth-first search. When searching for an initial counterexample, all accepting states satisfy the criterion. When closing the loop, only a single state closes the loop, namely the state q where we started to search for the loop. Since we move along an initial path and add loops from every state, we know that there is a path to the current state q from any state that appears in the path before q. We use this fact to generalize the criterion for the depth-first search: We just search for a state q' on the initial path up to the current state q. We form the word along the loop by concatenating two words: The word that is read along the path found by the depth-first search and the word along the subpath of the initial path from q' to q.

However, a downside of this approach is that it increases the chance that the same loop is taken several times: If there exists a loop that includes the subpath of the initial path from q' to q, the depth-first search can find this loop whenever it is called from a state between q' and q on the initial path. By enforcing that a subpath of this path is taken, it is more likely that the said loop containing it is taken again. Hence, we do not let the criterion find any state in the previous path. The subpath that is found by the criterion is dynamically moved throughout the process.

Simplifying the Elementary Bounded Language The resulting Elementary Bounded Language might use more infixes than necessary. For example, the languages $b^*a^1a^1b^*$ and $b^*(aa)^1b^*$ are equivalent. However, the representation does make a difference: Each infix means a component in the vector when intersecting the languages using Parikh images. Furthermore, each infix of the Elementary Bounded Language requires a layer in the automaton that represents the intersection between an input automaton and the Elementary Bounded Language. Hence, we want to keep the number of infixes low by uniting two consecutive infixes that both have the exponent 1 to a single one with the same exponent.

We do this simplification on the fly with a constant overhead. It is implemented in `FindCounterExampleEBL` and takes place when building the Elementary Bounded Language. Whenever a new infix is added, this is done using an inner function. If an infix without a Kleene star is added, `FindCounterExampleEBL` calls its inner function `AddNonStarStringToReverseEBL`. This function has two parameters: The first parameter is the string of the infix that shall be added. The second parameter is the current list of tuples of infixes of the Elementary Bounded Language and the Boolean value denoting whether the exponent is the Kleene star. The list is stored in reverse order, which allows accessing the most recent infix in constant time. The result is represented in the same way. The function `AddNonStarStringToReverseEBL` checks whether the most recent infix has the exponent 1. In that case, it concatenates the most recent infix and the input prefix. The concatenated word replaces the former most recent infix in the final result of the function.

Similarly, there is a simplification that can be done when adding an infix with the Kleene star as the exponent: The languages b^* and b^*b^* are equivalent. The approach is quite similar to the previous case: An infix with the Kleene star as the exponent is added using the function `AddStarStringToReverseEBL`. This function checks whether the most recent infix has the same string and the same exponent as the input. In that case, the input will not be added to the Elementary Bounded Language.

In both cases, empty strings as infixes of the Elementary Bounded Language do not alter it at all. Hence, the inner functions `AddNonStarStringToReverseEBL` and `AddStarStringToReverseEBL` can just omit them: These functions return the unmodified input list if the given input string is empty.

8.4 The Module `CounterExampleSymbolifier`: Generating Automata with Word-Transitions

Once we have an Elementary Bounded Language of potential counterexamples, we want to check whether any of them are non-spurious counterexamples. In order to do so, we need to preprocess the input languages, which are supposed to be intersected, so that we can use the Parikh Image Presburger Formula to decide whether there are non-spurious counterexamples in the intersection. This preprocessing includes a transformation of the automaton and an intersection with the Elementary Bounded Language.

As presented in Section 6, the automaton is transformed in a way that the transitions do not represent single letters of the original alphabet but an infix of the Elementary Bounded Language. We introduce a fresh alphabet. Each letter of this alphabet represents a whole infix. The transitions of the transformed automaton read a letter of the fresh alphabet and go to a state that can be reached when reading the corresponding infix in the original automaton.

The intersection with the Elementary Bounded Language is hence not done using the literal language but its representation using letters from the fresh alphabet. The module `CounterExampleSymbolifier` implements these two steps and offers a function that coordinates the execution of these steps on a list of automata.

The main function for this purpose is `SymbolifyAutomata`. Its two parameters are the Elementary Bounded Language and a list of finite automata. The Elementary Bounded Language uses the same format as described in the previous chapter. The automata are all of the type `Fare.Automaton`. The result is also a list of `Fare.Automaton` objects.

The function uses the function `SymbolicRegExFromEBL` that converts the given Elementary Bounded Language to a regular expression of the type `Fare.RegExp` that represents each Elementary Bounded Language infix as a different unicode character. The used unicode range starts with the character with the ID that is specified by the constant `symbolicSymbolRangeStart`. The IDs increase by one for each infix. This regular expression is then converted to an automaton by a built-in method of `Fare`.

Each automaton in the list is intersected with the said automaton obtained from the regular expression. Before this can be done, each automaton needs to be processed using the function `SymbolicWordAutomatonFromEBL`. The input is an Elementary Bounded Language and an automaton of the type `Fare.Automaton`. The output is an automaton of the same type. The automaton is traversed using the function `traverseAutomatonAndSymbolize`. That function maintains a map from the ID of a state in the old automaton to the `Fare.State` object of the corresponding state in the transformed automaton. A worklist contains all states that occur when traversing the automaton. A set of already visited states prevents double visits. From every state, we read each of the infixes of the Elementary Bounded Language. We then add a transition to each new state that was reached after reading the infix. The transition reads the symbol that represents the current infix.

If a state is never reached throughout this traversing process, then it does not get a corresponding state in the new automaton. States in the transformed automaton are only created when the helper function `AddSymbolicCharacterTransition` introduces a transition between them.

Besides its main functionality, the module offers functions for easier usage of the results: The function `NegatedLiteralRegExFromEBL` negates the Elementary Bounded Language and returns the result as a regular expression. This is useful if the Elementary Bounded Language is entirely spurious. Then, an intersection of the language of all potential counterexamples with the said regular expression just ignores these spurious counterexamples. When generating the Parikh Image Presburger Formula, it is necessary to know the alphabet of the automata. One possibility is to extract it from the automata by traversing them and tracking all letters at the transition. The function `symbolicCharactersOfEBL` offers a more convenient and faster way: It generates the said fresh alphabet as a list of Unicode characters from a given Elementary Bounded Language.

8.5 The Module `CounterExampleParikhIntersecter`: Parikh Image as a Presburger Formula

In Section 7, we presented a technique to extract the Parikh Image of an automaton as a Presburger Formula. As motivated in Section 4.2, the formula is needed in order to check whether there is a non-spurious counterexample in the Elementary Bounded Language. The previously presented module transformed the input languages in a way that each Parikh vector uniquely corresponds to a word in the Elementary Bounded Language. Hence, we can decide whether there is a non-spurious counterexample by deciding whether there is a common vector in the Parikh images of each of the transformed input languages. To this end, the module `CounterExampleParikhIntersecter` provides functions that create Parikh Image Presburger Formulas. Their satisfiability can be checked by querying the Satisfiability Modulo Theories solver `Z3`.

The function `ParikhIntersectionFormula` returns a Presburger Formula that is satisfiable if and only the intersection of the Parikh Images of the given automata is not empty. The resulting formula is represented as a type of the library `Z3Fs`. The parameters of the function are the alphabet and a list of automata. The alphabet is a list of Unicode characters. The automata are of the type `Fare.Automaton`. The function creates a conjunct of the Parikh Image Presburger Formulas of each of the input automata and adds constraints for each of the letters to ensure that the corresponding Parikh variable is consistent with the variables that represent how often each transition is taken in a run.

The implemented formula differs slightly from the formula presented in Section 7: There is no existential quantifier. All variables are free variables. However, the variables of each automaton are distinct. Only the Parikh variables are shared between the automata. Hence, satisfiability is still the same as for the original formula. Only the assigned variables differ.

The function `ParikhIntersectionFormula` computes the conjunct of several constraints for each automaton. The constraints are generated by several functions. Each of these functions gets an object of the type `TransitionVariableTable`. This custom data structure stores the variables counting the usage of each transition in a run. It also stores the helper variables for the distance rating and the accepting state. It is created by the function `createIntVariablesForAutomaton`, which works on an object of the type `TransitionTable`. This type is generated from a `Fare.Automaton` object by the function `GetTransitionTable`. It is needed in order to access final states and transitions of certain states without needing to traverse the whole automaton every time.

The said functions for the constraints are:

`distanceVariableConstraints`	Enforces that only connected transitions may be used by using the distance rating variables of the states.
`createFlowEquationFormulas`	Requires each state to be left and entered equally often in the run.

`createFormulasForFiniteStates`	Ensures that the helper variables for the accepting states are positive and that their sum is 1.

These functions mainly form sums of variables and constants, build up equations and inequalities and return them embedded in logic formulas. Parts of those formulas are organized in inner functions for better readability. The variables of those formulas are just read from the `TransitionVariableTable` object. The actual creation of those variables is done by specific functions for each type of variables. This allows changes to the variable naming scheme at a single place in the source code.

8.6 A Note on Coding Style and the Module `Common`

After clarifying the purpose of the main modules and their functions, we will give a brief overview on the coding style and the functions provided by the module `Common`. These explanations are supposed to help extending the existing source code.

Composing Functions In order to produce more concise code, we tend to define functions implicitly. This means that the function is not implemented literally but composed out of other functions or the result of a higher-order function. An example is the function `ListCons`. It adds an element to the beginning of a list. It is hence equivalent to the built-in operator `::` but since it is a function, it can be composed with other functions, passed as a parameter, or partially evaluated to new functions.

Reduce if/else Control Flow Commands Manipulating the control flow using the binary control flow commands `if` and `else` breaks the readability of the code. It often introduces either redundancy or additional fields that store the intermediate result of the `if/else`-case distinction. Hence, we try to avoid such constructs. To this end, we introduce functions like `GetMapItemOrPlaceholder`. The parameters are a key, a placeholder and a map. The function always returns an item. If the map contains an item with the given key, it will be returned. If there is no such item, the placeholder will be returned. This is typically a neutral element. This allows a single composition of functions for both cases. We typically use this for maps whose elements are lists and want to extend the lists: We use the said function and use the empty list as a placeholder. The addition of an element to a non-existing list in the map becomes the initialization of the list with this element. If the list has already existed before, the element is added to the existing list as expected. Hence, whenever we need this kind of case distinction, a single function call takes care of that without explicitly differentiating these cases every time.

Immutable Data Structures The default data structures in F# are immutable. This means that operations on an immutable value or object do not manipulate the object but formally return a manipulated copy of the object. The idea is to avoid side-effects on these data structures. Whenever a function is called, its return value is the only value that was not present before the function call. However, the concept of immutability is mainly a model for the programmer: If the input value is never used again, the compiler may optimize such calls so that the data does not actually need to be copied. Libraries written in programming languages other than F# mostly encourage or require the use mutable data structures. Hence, objects related to `Fare` are still mutable in our library. However, our own data structures are immutable. An example: An object of the type `AnnotatedState` is immutable and all information about the object itself will hence never change. Yet, it does always point to the same `Fare.State` object although this object might change its values.

Built-In Repetition Commands Since F# is a multi-paradigm language, it is possible to use loops with `for` and `while` expressions. However, such expressions are mostly used for mutable data structures. Since we prefer immutable data structures, the repetitions are either modelled as recursive functions or as applications of higher-order functions. The advantage of built-in higher-order functions is that they are typically pre-optimized and reduce the possible overhead of immutable data structures. We commonly use functions like `map`, `fold`, and `filter`. These functions apply a given function on every element in a collection. The function `map` returns a collection containing the results of these function calls, `filter` returns a list containing those elements whose function calls returned `true`, and `fold` uses the return values of the function calls as one of the parameters of the next call and returns the last return value. An example in the module `common` is the function `SuccessorFold`. This function uses the built-in function `fold` to implement a variant that uses two consecutive elements of the list as parameters for the function call. An example for the usage of the function is the extraction of a word from a path of states: Both the current and the successor state are needed in order to determine which transitions could possibly be taken along this path.

Tail Recursive Functions Whenever built-in functions are not suitable, the needed repetitions are performed using recursive functions. Recursive functions perform better if they are tail recursive. If the recursive call is the last call in the function body, only the current call and not the whole history of recursive calls needs to keep its data in the memory. The needed time can also be reduced: The data of a previous call can safely be reused during the next calls. This allows compiler optimizations regarding the mutation of immutable objects.

Reversed Lists Adding an element to the end of a list typically costs linear time as the end of the list needs to be found every time. Since the beginning of a list can be retrieved in constant time, adding elements to the beginning of the list performs much better. We make use of this property whenever we construct a list: Throughout the process, we use a reversed list and add the elements to beginning. The final result will be inverted once in linear time and returned.

Inner Functions Inner functions have implicit access to the parameters of the outer function. One advantage is that this keeps the number of explicit parameters low. We use them to hide complexity: Many recursive functions and functions that are supposed to be used as a parameter of `fold` are implemented as inner functions. The outer function has less parameters and only prepares the call of `fold` or the initial call of the recursion. Inner functions also allow to split the task to several smaller inner functions without increasing the effort when using the function.

The function `SplitListAtCondition` is an example for the use of tail recursive inner functions and reversed lists. Its parameters are a condition and list. It splits the input list and returns a tuple of two lists. The first element of the input list that satisfies the given condition separates the two output list: It is the last element of the first list and the first element of the second list. This is implemented using the tail recursive inner function `SplitSequenceAtConditionHelper`. Its parameters are two lists. Initially, the first list is empty and the second list is the input list of the outer function. Each recursive call moves the first element of the second list to the beginning of the first list until the condition is satisfied. Once this happens, the first list is reversed. The return value is a tuple of the reversed first list and the second list.

A special case of lists are strings. The performance problem is similar: Appending characters or strings to strings usually requires to follow the first string to its end in linear time. Thus, the built-in class `System.Text.StringBuilder` is offered by the .net/mono environment. Its mutable objects collect strings that are passed through the method `Append`. Once the method `ToString()` is called, the whole string is created in linear time. We use this way of concatenating strings for example when creating regular expressions of negated Elementary Bounded Languages or when extracting words along paths in the automaton.

8.7 Using and Integrating the Tool

There are three ways to use the tool: The modules can be accessed using F# Interactive or embedded in other mono projects. There is also a binary file that randomly generates instances according to the command-line parameters examples and runs them with debug output on the time consumption.

The module `IntersectionCegar` provides the simplest way of accessing the modules. This module offers implementations of the full CEGAR loop for regular inclusion of regular intersections. The implementation is split into two kinds of functions. The function `IsEmptyAfterNRefinements` extracts the Elementary Bounded Language, checks it for non-spurious counterexamples, and refines the language for a given number of times. Besides the number of allowed repetitions, the list of intersected automata and an automaton are given as parameters. This automaton will be searched for counterexamples and refined iteratively. The result is `None` if the number of loops did not suffice to prove the existence or absence of counterexamples. If the number of loops does suffice, the returned `Some` value carries a bool value denoting whether the input language only contained spurious examples.

The other kind of functions is called with the input languages, the specification, and the desired number of repetitions. These functions create an initial counterexample automaton and pass it to `IsEmptyAfterNRefinements`. This design allows to solve our regular inclusion problem and a related problem:

The function `RegularInclusionsNStepsEmptinessVariant` implements the procedure as introduced: First, it creates an over-approximation of the intersection, then it intersects it with the complement of the specification. This language of potential counterexamples is then examined for non-spurious counterexamples.

The function `RegularIntersectionNSteps` does not solve regular inclusion but emptiness of regular intersections. It over-approximates the intersection of a given list of automata and directly passes the result to `RegularIntersectionNSteps` without intersecting with a complement of any specification. It is possible to use this function for regular inclusion: Instead of over-approximating just the intersection, the function over-approximates the language of counterexamples.

The function `RegularInclusionNStepsEmptinessVariant` complements the counterexample language, adds it to the list of languages to intersect, and passes the extended list to `RegularIntersectionNSteps`. The advantage is that this variant uses the fact that the over-approximation can preserve emptiness in some cases. The intersection itself is typically not empty. The counterexample language is per definition empty if and only if the intersection is included in the specification. Hence, this variant aims to prove the inclusion without encountering any spurious counterexample. The intuition is that this variant might already rule out spurious counterexamples at an earlier stage before much information is lost by the over-approximation.

However, this variant requires one additional intersection and over-approximation step. This can be costly depending on the location in the tree of intersections and over-approximations. In our implementation, the worst case happens when the number of languages to intersect is already a power of two. Then, there is a single language left in each layer that cannot be intersected and the additional intersection happens in the end. This means that the newly added intersection step adds more states that need to be compared and united than any other intersection in the original procedure.

The additional intersection and over-approximation step is of course comparably cheap if the number of all automata including the complemented specification automaton is a power of two. Then, the additional intersection is just an intersection between the negated specification and one of the input automata.

An Example in F# Interactive We now provide an easy example that illustrates how to use our tool directly from F# Interactive. We assume that the package `Fare`, the required external files `z3FS.dll`, `libz3.so`, and `Microsoft.Z3.dll`, and the used references are already sent to F# Interactive.

In this example, we intersect the languages $ab^*c^*d^*$, $a^*bc^*d^*$, $a^*b^*cd^*$, and $a^*b^*c^*d$. Then, we check whether the intersection $abcd$ is included in $(abcd)^*$, aa^*, and $\overline{aa^*}$. The second check is supposed to fail. The other two checks are intended to succeed.

We first load the needed modules of our tool. It is possible to automatically measure the time consumption of every command in F# Interactive. In this example, we chose to activate this feature:

```
> #load "Common.fs"
#load "Intersect.fs"
#load "CounterExample.fs"
#load "CounterExampleSymbolifier.fs"
#load "CounterExampleParikhIntersecter.fs"
#load "IntersectionCegar.fs"
#time "on";;
```

The output is a very long list of all signatures of loaded types, fields, and functions. We can now use the regular expression feature of Fare to create the example automata:

```
> let anostarbcd = Fare.RegExp("ab*c*d*").ToAutomaton()
let abnostarcd = Fare.RegExp("a*bc*d*").ToAutomaton()
let abcnostard = Fare.RegExp("a*b*cd*").ToAutomaton()
let abcdnostar = Fare.RegExp("a*b*c*d").ToAutomaton()
let spec = Fare.RegExp("((abcd)*)").ToAutomaton();;
```

These five commands create the desired automata from their regular expressions. Technically, Fare interprets the regular expression string, parses it to a regular expression object, and constructs an automaton from it. It is also possible to construct an automaton directly:

```
> let aastar = new Fare.Automaton()
let accstate = new Fare.State()
accstate.Accept <- true
accstate.AddTransition(new Fare.Transition('a', accstate))
```

```
aastar.Initial.AddTransition(new Fare.Transition('a', accstate))
let notaastar = aastar.Complement();;
```

This example illustrates two ways to construct automata: The `Fare.Automaton` object `aastar` represents an automaton accepting the language aa^*. It is created by making the `Fare.State` object `accstate` accepting, adding a `Fare.Transition` object to the state itself reading `'a'`, and a transition reading the same character from the initial state of the automaton to `accstate`. The `Fare.Automaton` object `notaastar` is not created state-by-state but as a composition of automata: It is the complement of `aastar`. We can now use these automata as the input of our procedure.

```
>  [ anostarbcd ; abnostarcd ; abcnostard ; abcdnostar ] |>
 IntersectionCegar.IntersectionCegar.RegularInclusionNSteps 100 spec;;
... Real: 00:00:00.003, CPU: 00:00:00.003, GC gen0: 0, gen1: 0
val it : bool option = Some true
```

We use the function `RegularInclusionNSteps` in order to perform our standard approach on the given automata. The parameters are the maximal number of repetitions of the CEGAR loop, the specification, and a list of the automata to intersect. The intersection is *abcd*. It is included in the specification $(abcd)^*$. Our algorithm took three milliseconds to agree on that. We can also use the approach that reduces the problem to emptiness of an intersection:

```
> [ anostarbcd ; abnostarcd ; abcnostard ; abcdnostar ] |>
 (IntersectionCegar.IntersectionCegar.
 RegularInclusionNStepsEmptinessVariant 100 notaastar);;
Real: 00:00:00.003, CPU: 00:00:00.002, GC gen0: 0, gen1: 0
val it : bool option = Some true
```

The function call is quite similar. The parameters work in the same way. This time, we chose to check the intersection against the specification $\overline{aa^*}$. The output shows that our algorithm took two milliseconds to show that the intersection does satisfy this specification as well. We can also check whether our intersection is empty:

```
> [ anostarbcd ; abnostarcd ; abcnostard ; abcdnostar ] |>
 IntersectionCegar.IntersectionCegar.RegularIntersectionNSteps 100;;
... [
 p_A = 1,
 f_1395 = 1,
 x_1394,A,1395 = 1,
 z_1395 = 2,
 z_1394 = 1,
... ]
Real: 00:00:00.029, CPU: 00:00:00.022, GC gen0: 0, gen1: 0
val it : bool option = Some false
```

This function call is similar to the previous two. The difference is that there is no more parameter for the specification. Our algorithm takes 22 milliseconds to find a counterexample showing that the intersection is not empty. The output also shows that Z3Fs prints the satisfying model for the formula that was generated by our algorithm. The output was shortened in order to just show variables concerning a single automaton of the four automata.

These automata represent the intersection of an input automaton with the Elementary Bounded Language of counter examples by representing each Elementary Bounded Language infix as a single letter. In this example, the only Parikh variable is p_A and it is set to 1. This means that the Elementary Bounded Language has a single infix. It is represented by the unicode character A. The state distance variables z_1395 and z_1394 tell us that the two states of the automaton have the IDs 1394 and 1395. The value 1 indicates that the state with the ID 1394 is the initial state. A transition from this initial state to the other state is apparently used in the run since the other state has the distance marking 2. The variable x_1394,A,1395 tells us that the only transition in the automaton leads from the initial state to the other state and reads A. This transition is taken exactly once. This is consistent with the Parikh variable p_A. The non-initial state is the only accepting state since f_1395 is the only f_... variable. The run also ends there since it is set to 1.

Running the Algorithm Step-By-Step The previous specifications did include the intersection. We now examine a specification that does not include the intersection. This can of course be done using the previously introduced functions. In order to demonstrate the algorithm and how its various functions are used, we apply these functions step-by-step. We show the difference between the positive case and the negative case: The intersection is included in $\overline{aa^*}$ but it is not included in aa^*.

We have already prepared the input automata, two specifications, and implicitly also their complements since the two specifications are complements of each other. First, we apply the IntersectAutomataList of the module Intersect on the list of automata:

```
> let overapproximatedIntersection =
    [ anostarbcd ; abnostarcd ; abcnostard ; abcdnostar ] |>
    Intersect.Intersect.IntersectAutomataList;;
...
Real: 00:00:00.003, CPU: 00:00:00.001, GC gen0: 0, gen1: 0
val overapproximatedIntersection : Fare.Automaton
```

The result is computed after a millisecond. It is a single Fare.Automaton object whose language contains the intersection. We now intersect the automaton with the complement of the specification. We have already shown that the intersection is included in $\overline{aa^*}$. If we want to check that manually, we need to intersect the over-approximation with the complement of the specification. The complement aa^* is already prepared as aastar. We can intersect the over-approximation with aa^* to get the language of potential counterexamples:

```
 > let counterex1 = overapproximatedIntersection.Intersection(aastar)
let numStatesOverapprox = overapproximatedIntersection.NumberOfStates
let numStatesCex1 = counterex1.NumberOfStates
let cex1initAccept = counterex1.Initial.Accept
let cex1ebl = counterex1 |>
    CounterExample.CounterExample.FindCounterExampleEBL ;;
...
Real: 00:00:00.008, CPU: 00:00:00.009, GC gen0: 1, gen1: 0
val counterex1 : Fare.Automaton
val numStatesOverapprox : int = 5
val numStatesCex1 : int = 1
val cex1initAccept : bool = false
val cex1ebl : (string * bool) list option = None
```

We used several built-in methods of Fare to intersect the automata and examine the result. The over-approximated intersection has five states. The automaton of the language of potential counterexamples has a single state. It is not accepting. Hence, the language is empty. If it was accepting, we would need to check whether the empty word is a counterexample and remove the initial accepting state. After that, we can use the module `CounterExample` to create an Elementary Bounded Language of potential Counter Examples. Since our counterexample automaton accepts no word, there is no Elementary Bounded Language either. Hence, the return value is `None`.

After proving again that the intersection is contained in $\overline{aa^*}$, we show that it is not contained in aa^*. To this end, we intersect the over-approximated intersection with the complement $\overline{aa^*}$. We apply the same steps as before:

```
 > let counterex2 = overapproximatedIntersection.Intersection(notaastar)
let numStatesCex2 = counterex2.NumberOfStates
let cex2initAccept = counterex2.Initial.Accept
let cex2ebl = (CounterExample.CounterExample.FindCounterExampleEBL
counterex2);;
...
Real: 00:00:00.000, CPU: 00:00:00.000, GC gen0: 0, gen1: 0
val counterex2 : Fare.Automaton
val numStatesCex2 : int = 5
val cex2initAccept : bool = false
val cex2ebl : (string * bool) list option = Some [("abcd", false)]
```

This time, the automaton of the language of potential counterexamples contains five states. The initial state is not accepting again. Yet, the language might not be empty if one the five states is accepting. Indeed, there is an Elementary Bounded Language of potential counterexamples. Its only infix is *abcd*. Since the infix is not looped, this is also the only word in the language. In order to demonstrate the procedure, we will not just check whether this word is included in all of the intersected languages but perform the actual procedure:

```
> let eblv = cex2ebl.Value
let alphabet = (CounterExampleSymbolifier.CounterExampleSymbolifier.
symbolicCharactersOfEBL eblv)
let oneAut::otherAut = (CounterExampleSymbolifier.
CounterExampleSymbolifier.SymbolifyAutomata eblv
[ anostarbcd ; abnostarcd ; abcnostard ; abcdnostar ])
let autNumOfStates = oneAut.NumberOfStates
let autInitAcc = oneAut.Initial.Accept
let trans = oneAut.Initial.Transitions |> Seq.head
let otherStateAcc = trans.To.Accept;;
... Real: 00:00:00.002, CPU: 00:00:00.002, GC gen0: 0, gen1: 0 ...
val otherAut : Fare.Automaton list =
 [Fare.Automaton; Fare.Automaton; Fare.Automaton]
val alphabet : char list = ['A']
val oneAut : Fare.Automaton
val autNumOfStates : int = 2
val autInitAcc : bool = false
val trans : Fare.Transition = A -> 0
val otherStateAcc : bool = true
```

Since the Elementary Bounded Language was encapsulated in an `option` type, we had to extract it first. Then, we could pass it to a function that prepares the automata for the intersection using the Parikh Image Presburger Formula. We inspected one of the four automata. It has a non-accepting initial state and an accepting state. A transition from the initial to the accepting state reads the Unicode character `A`. This character represents the only Elementary Bounded Language infix

We can now extract the Parikh Image Presburger Formula from the automata and the Elementary Bounded Language using the function `ParikhIntersectionFormula`:

```
> let parikh = oneAut::otherAut |> (CounterExampleParikhIntersecter.
CounterExampleParikhIntersecter.ParikhIntersectionFormula alphabet);;
Real: 00:00:00.001, CPU: 00:00:00.002, GC gen0: 0, gen1: 0
val parikh : Z3.FSharp.Bool.Bool = ...
```

The output is a massive Presburger formula in prefix notation. We can now check satisfiability of this formula:

```
> let z3result = Microsoft.Z3.FSharp.Bool.Z3.Solve parikh;;
[ p_A = 1 ... ]
Real: 00:00:00.013, CPU: 00:00:00.012, GC gen0: 0, gen1: 0
val z3result : Z3.Status = SATISFIABLE
```

The library `Z3Fs` outputs a satisfying assignment for every variable in the formula. Since we have already discussed how to interpret these solutions, only the assignment of the Parikh variable is shown. A counterexample is a word consisting of one occurrence of the Unicode character `A`. As mentioned before, this represents *abcd*.

If we had not found a counterexample, we would have needed to exclude the Elementary Bounded Language of spurious counterexamples from our language of potential counterexamples and search for another Elementary Bounded Language. This could be done in this way:

```
> let newCexAut = counterex2 |> ((CounterExampleSymbolifier.
CounterExampleSymbolifier.NegatedLiteralRegExFromEBL eblv).
ToAutomaton().Intersection);;
```

The function `NegatedLiteralRegExFromEBL` generates the negation of the Elementary Bounded Language as a regular expression. After a conversion to an automaton, it can be intersected with the specification. The result is a specification without the spurious counterexamples from that Elementary Bounded Language.

9 Benchmarks

We benchmarked our implementation against randomly generated instances of the problem. Three problems arise when benchmarking randomly generated automata:

It is difficult to group instances of notably large instances that are similarly challenging. For example, an instance might be solved quickly if the first intersection of two of the input automata is already empty. An instance of the same number of automata, states, letters in the alphabet, transitions, and accepting states might require considering the whole set of input automata and several refinement steps.

Instances are non-trivial once the tree over intersected and over-approximated languages has two or more layers. Furthermore, the automata should be chosen so that they are not immediately empty. Some of these instances already require hours of computation time. This makes it practically infeasible to run the benchmarks on high numbers of instances in order to compensate different difficulties of specific instances.

The state space in the classical approach to intersect regular languages grows exponentially with the number of automata to intersect. This means that it is also not feasible to use the classical approach to group problem instances of similar difficulty.

Hence, we decided to run benchmarks on various types of automata rather. The goal was to get insights rather on general properties of the algorithm than on exactly quantified runtime behavior.

Generation of Test Instances We implemented a random automata generator that created non-deterministic automata according to the Vardi-Tabakov model as introduced in [TV05]. Our function `VardiTabkov` needs five parameters: The size of the state set, the alphabet size, the density of accepting states, the density of transitions, and a `System.Random` object. The density of accepting states is given as a `double` value between `0.0` and `1.0`. It specifies the desired ratio between the number of accepting states and the number of all states. The transition density is also a non-negative `double value`. For each letter, it describes the average of outgoing transitions per states. Since it is fixed for all letters of the alphabet, this means that each letter is read by equally many transitions in the automaton. Intuitively, it is a measure for the average non-determinism: If the transition density is 2.0, then an average state has two outgoing transitions reading the letter the same letter.

The function generates an array of states, sets a random state in this array as the initial state, marks the desired number of states as accepting states, and adds random transitions between any pair of states until the given density of transitions is reached for each of the letters. It keeps track of the existing transitions for each of the letters in order to prevent the multiple generation of the same transition.

Whenever the algorithm needs a random number, it is generated using the random number generator that is provided as the `System.Random` object. If we generate two automata using the same parameters and two `System.Random` objects that are initialized with the same seed, we get two identical automata. This allows to just store the random seed while still being able to reproduce a test instance.

The problem is that the result is not always connected. This means that some states might not be reachable in a graph-theoretical sense. This influences benchmark results in two ways: Only the reachable parts of the automata are considered by our algorithms. If the function produces two automata for the same parameters, one of them could still consist of a single reachable state whereas all states in the other automaton are reachable. Another problem is the unpredictable distribution of the parameters: The given parameters described the global characteristics of the automaton. The characteristics of the reachable parts of the automaton might differ. Even if half of the states are accepting states, the language might still be empty if the set of reachable state is a subset of the non-accepting states.

The Benchmarking Set-Up The benchmarks were run in a virtual machine. We created an executable file that runs a variant of our implementation of the CEGAR loop. In contrast to the previously presented functions, it measures the time for the complete procedure and various specific steps. It does not only print the result of the inclusion check but also the reason that indicates at which step in the procedure the result was concluded. The executable file has several parameters. These are the number of automata to intersect and the parameters of our function `VardiTabkov`. The random number generator is initialized with a given integer parameter and then passed to the function `VardiTabkov` as a `System.Random` object. The automata to intersect and the specification automaton are generated with the same parameters. The random number generator will not be reset after the generation of each automaton. The automata in the instance are hence different.

A shell script calls the executable file with several different parameters and stores the parameters and the output. Since the instances are created from the given random seed, the results can be reproduced. The stored output allows to analyze the behavior of the algorithm on this instance.

The shell script combines two alphabet sizes $(2, 3)$, three accepting state densities $(1.6, 1.8, 2.0)$, two numbers of automata $(4, 8)$, two numbers of states $(5, 10)$, and sixteen random seeds $(0 - 15)$.

The virtual machine had eight gigabyte of RAM and eight virtual processors at $2.26GHz$. The host had eight gigabyte of RAM as well and two Intel processors, each of them with four cores at $2.27GHz$. The configuration of the memory was a problem: Since the host was running other services besides the virtual machine, it ran out of memory for the guest system so that some of the memory had to be swapped to the hard drives. Also, the guest system was short on virtual memory as well. The whole system got unacceptably slow. Hence, some of the planned benchmark instances had to be terminated without begin executed. Additionally, some instances were started but not finished. There were several reasons. Some executions were terminated externally due to a lack of resources. In others, the mono runtime crashed when trying to allocate even more memory. Sometimes, Z3 refused to continue due to a lack of memory.

Success Rate In total, our shell script was supposed to generate 1152 instances. Out of those, 881 instances were actually started. During their runtime, 55 instances did not finish successfully. Hence, 826 out of 1152 ($\sim 71.7\%$) instances were executed as intended. The limit of 100 refinement steps was reached in in 41 cases. This means that our tool was able to prove or disprove the claim in $\sim 94.4\%$ of the finished executions, $\sim 77.8\%$ of the started executions, and $\sim 59.5\%$ of the intended executions.

Interestingly, only a single instance was stopped during the over-approximation phase. The others were terminated after at least one Elementary Bounded Language had been extracted. There were 15 executions that could not finish the first refinement step.

Refinement Steps The refinement steps are expensive in the sense that every finished refinement requires an actual intersection. This leads to automata that get bigger and hence consume more resources. This might be part of the reason for the effect that 25 out of 55 aborted executions ended after refinements took place. Hence, we want to know how many refinement steps are actually needed.

Of the 826 finished executions, 773 ($\sim 93.6\%$) concluded a result during, before, or right after checking the first Elementary Bounded Language. As we mentioned before, 41 ($\sim 5\%$) of the executions did not manage to conclude a result using at most 100 refinement steps. This means that only 12 ($\sim 1.5\%$) executions started more than one refinement step and reached a result. The exact numbers of refinements in these cases were:

- one refinement: five times (counterexample: three times, no more counterexamples candidates: twice)
- two refinements: twice (no more counterexamples)
- $1 \times 22, 2 \times 23, 1 \times 46, 1 \times 48$ refinements (counterexample found)

Interestingly, this means that we could have reduced the number of allowed loops from 100 to 48 without losing any result. Furthermore, there are actually cases where we could rule out all counterexample candidates. However, this did not happen in any case that needed more than two refinement steps. A reason might be that there are languages of potential counterexamples that include infinitely many disjoint Elementary Bounded Languages.

In total, there are eight counterexamples that were found after the first refinement. There are 22 instances where the counterexample was found before refining the language. This means that our test instances consisted of mainly positive instances.

Time Consumption We computed the minimum, average, and maximum execution time of the finished test instances. The results are grouped by the parameters of the input sizes. The three columns represent the density of accepting states. The rows group the number by the parameters concerning the transitions: Each row addresses a different combination of transition density and alphabet size. Each different combination of parameters was tested on four input sizes. The different problem sizes are shown in different the lines per cell: The first line in a cell contains the results for four automata with five states, the second line four automata with ten states. The last two lines consider eight automata with five and ten states. There are three numbers in a line of a cell. These are the minimum, average, and maximum runtime in milliseconds of the up to 16 instances with the same parameters and input sizes.

	AD = 0.25	AD = 0.5	AD = 0.75
$\|\Sigma\| = 2$,	69, 619, 3157	76, 328, 1354	105, 9768, 68349
TD = 1.6	71, 443, 1407	84, 1039, 4006	116, 727, 2607
	70, 272, 816	78, 348, 1062	115, 338, 712
	71, 478, 1551	89, 2739, 24489	114, 695, 1584
$\|\Sigma\| = 2$,	69, 55714, 718981	84, 514, 4141	114, 21566, 208540
TD = 1.8	70, 393, 1026	77, 1489, 4510	116, 748, 2730
	70, 167, 312	79, 311, 1051	111, 596, 2588
	72, 433, 1698	75, 4199, 42445	114, 962, 2820
$\|\Sigma\| = 2$,	251, 39527, 284061	761, 38449, 178672	491, 17310, 95101
TD = 2.0	6856, 14393, 24201	10129, 14594, 26592	8379, 9128918, 80948391
	207, 359, 589	336, 9224, 69260	220, 1425, 8769
	12238, 23443, 32681	10274, 115519, 1147858	13920, 171923, 1855200
$\|\Sigma\| = 3$,	74, 730, 2157	81, 172328, 2042411	120, 13852, 87983
TD = 1.6	72, 1880, 6047	71, 1537, 2922	72, 2793, 8430
	74, 441, 1386	75, 364, 2164	73, 1189, 8682
	71, 1751, 5799	72, 1688, 3090	72, 4157, 11701
$\|\Sigma\| = 3$,	73, 343, 1268	74, 94751, 1124332	95, 10858, 82284
TD = 1.8	71, 1795, 5494	71, 1406, 2583	83, 2537, 6008
	74, 215, 576	74, 428, 2157	71, 974, 4875
	72, 1426, 3891	71, 1882, 3814	71, 3117, 5850
$\|\Sigma\| = 3$,	74, 33447, 299805	290, 87640, 647013	403, 18690, 68120
TD = 2.0	12141, 20165, 23438	9565, 23479, 45685	13687, 30634, 109120
	73, 390, 732	464, 56581, 516770	647, 4029, 26916
	12807, 232549, 2372890	9078, 34994, 53096	25708, 122052, 1155320

We observe several patterns about the minimal runtimes: For most parameter configurations, the minimum runtimes for different input sizes differ by less than ten milliseconds. Furthermore, the runtime increases in many of theses cases stronger when doubling the state set than when doubling the number of automata. This indicates that our algorithm is not affected by the exponential state space as it is the case for the conventional approach. The minimal runtimes are most likely to be caused by instances with empty intersections. This explains similar runtimes for doubled numbers of automata: If the intersection is empty after considering a few automata, it remains empty no matter how many other automata are left. Hence, the number of additional steps is rather low.

The extreme values for the maximum runtimes show an interesting, counter-intuitive behaivor: The extreme values often occur when only four automata are intersected. If we increase the number of intersected automata, the chances rises that the intersection becomes empty. Hence, the said cases are more likely to actually contain counterexamples. The search for these counterexamples is expensive – especially if this search requires many refinement steps. We can also observe remarkably longer runtimes if the transitions density is 2.0. If there are more transitions, the likeliness of unreachable states decreases. Hence, the relevant part of the automaton is bigger and the runtime increases. Furthermore, the chance of a non-empty intersection increases. We analyze this assumption by only considering the instances where our procedure found counterexamples:

	AD = 0.25	AD = 0.5	AD = 0.75
$\|\Sigma\| = 2$, TD = 1.6	$335, 1280, 3157$	$1354, 1354, 1354$	$511, 511, 511$
$\|\Sigma\| = 2$, TD = 1.8	$335, 383, 432$	$373, 373, 373$	$1152, 1152, 1152$
$\|\Sigma\| = 2$, TD = 2.0	$284061, 284061, 284061$	$761, 870, 979$	$736, 2133, 4694$ $12108, 20478929, 80948391$
$\|\Sigma\| = 3$, TD = 1.6	$2117, 2117, 2117$	$4184, 4184, 4184$	
$\|\Sigma\| = 3$, TD = 1.8	$358, 358, 358$	$376, 376, 376$	
$\|\Sigma\| = 3$, TD = 2.0	$299805, 299805, 299805$	$849, 1463, 2042$ $516770, 516770, 516770$	$8844, 8844, 8844$ $11021, 11021, 11021$

There are only two cases where a counterexample was found in an intersection of eight automata. This supports the assumption that these kinds of instances tend to have empty intersections. The assumption that the worst cases are the ones where a counterexample is found does not hold. However, the minimum runtimes for finding a counterexamples are still always several times higher than the minimum runtimes in general for comparable cases. This indicates that the successful search for a counterexample is indeed an expensive case although it it not always the most expensive one. Thus, we examine these instances that required 100 refinement steps without proving or disproving the inclusion. The table shows the runtimes of such unsuccessful executions:

	AD = 0.25	AD = 0.5	AD = 0.75
$\|\Sigma\| = 2$, TD = 1.6	2680, 2680, **2680**		46174, 57261, 68349 2607, 2607, 2607
$\|\Sigma\| = 2$, TD = 1.8	3286, 361133, 718981		46330, 127435, 208540 2730, 2730, 2730
$\|\Sigma\| = 2$, TD = 2.0	10977, 36007, **77284**	5512, 73235, 178672 29024, 49142, 69260	6118, 33111, 95101 201762, 201762, **201762**
$\|\Sigma\| = 3$, TD = 1.6		18701, 1030556, 2042411	50774, 69378, 87983
$\|\Sigma\| = 3$, TD = 1.8		10864, 567598, 1124332	32321, 57302, 82284
$\|\Sigma\| = 3$, TD = 2.0	64439, 64439, **64439**	7374, 195726, 647013 112418, 112418, **112418**	25358, 46739, 68120 109120, 109120, 109120

There are only five configurations of parameters and input sizes where the maximum runtime without a result is not the maximum runtime of the general case. We highlighted these runtimes. In all five highlighted configurations, the worst-case is one where a counterexample was found. Only four of the worst-cases when finding a counterexample were not exceeded by an unsuccessful execution but by a successful one that ruled out all counterexamples.

Longest Runtime The absolute worst-case of all finished executions is an instance that took $80,948,391$ milliseconds (~ 22.5 hours) to find a counterexample. It had the alphabet size 2, the transition density 2.0, and a density of accepting states of 0.75. The random number generator was initialized with 10. Four automata with ten states were intersected. For comparison, we intersected the automata using the conventional method. Intersecting the two pairs of automata took 41 milliseconds and generated two automata of 100 states. Intersecting these automata took 2410 milliseconds. The intersection automaton has 9615 states and 454482 transitions. Complementing the specification and intersecting it with the intersection took a minute and 14.129 seconds.

There is a mismatch between almost a day of runtime using our approach and about two minutes of runtime when using the conventional approach. We wanted know reasons for this drastic difference. Since there is a counterexample, the last step of the computation was solving the Presburger Formula. This happened after the 46th intersection step and took $80,464,352$ milliseconds. This means that all parts of the program except for the last one took about eight minutes. This impressively long runtime of Z3 can be explained by the number of variables: The satisfying solution assigns values to $589,958$ variables. However, 589.892 of them are set to 0. Interestingly, the formula contains only seven Parikh variables that are barely used. In total, there are 36 state distance variables and four accepting state variables. This means that there is a single accepting state in each of the four automata that were preprocessed for intersection with the Elementary Bounded Language. The massive size of the formula occurred apparently due to fail states: Thousand of transitions do only read letters for which there is no Parikh variable. This means that these letters are none of the freshly introduced letters. The said transitions are connected to two non-accepting states. Hence, the purpose of these transitions is filtering out illegal letters. We assume that this effect was introduced by the automaton that was generated from the regular expression representing the Elementary Bounded Language. However, we did not observe this effect in any other example. A possible way to address this problem is disallowing transitions reading letters that are not in the language of the automaton.

Although the longest runtime was caused by an edge-case, we can still learn from it. The initial over-approximation and the first eight refinement steps took less than eight seconds. Also, the over-approximated automaton was obtained using about a thousand states. The explicit computation of the exact counterexample language took about twice as long. The automaton consists of almost ten thousand states. Although the computation of the initial-over-approximation and the first refinement steps is faster than the exact solution, our approach loses the race against the conventional approach due to the refinement steps. For example, excluding the spurious counterexamples in the 35-th refinement step took 872 milliseconds.

The following section presents an overview how variants of the procedure could be designed. One of the goals is avoiding the expensive and wasteful refinement steps. If we could find the counterexample in the presented instance using only a few refinement steps, even this extreme case would perform better than the conventional one.

10 Conclusion and Future Work

This work presented a way to transfer the concept of Counterexample-Guided Abstraction Refinement to regular inclusion and intersection. Our contribution includes methods to over-approximate the intersection and to extract Elementary Bounded Languages of potential counterexamples. Our work makes use of the existing approach to intersect Parikh Images using satisfiability of Presburger Formula. We also derived methods that are necessary to integrate these components, such as the algorithm that replaces infixes of Elementary Bounded Languages by fresh letters in order to obtain unique Parikh vectors. We implemented the whole procedure and demonstrated that the concept allows to solve instances whose explicit solutions would require millions to billions of states.

The goal of the implementation was to show that the whole procedure is generally feasible. Hence, the implementation is not tailored for specific classes of instances. Actual applications using the algorithm could customize steps to make use of properties of the expected instances. In this section, we will give an overview on conceptual limitations of the procedure and potential improvements. We will also cover potential modifications and uses of the offered tools besides the original CEGAR-like algorithm. These modifications address the question whether the approach can be a (semi-)decider. Also, bottlenecks as shown in the previous sections on benchmarks are tackled.

Being Lost in Nested Loops Since our approach relies on the extraction of Elementary Bounded Languages, it can not be a decider for all classes of regular languages. If the language of potential counterexamples has a nested loop like $(ac^*b)^*$, our procedure would find infinitely many Elementary Bounded Languages of the form ac^*b, $(ac^kb)^*$, and concatenations of such languages. If all of these words are spurious, then the language of possible counterexamples will never be empty.

Even if there are actual counterexamples, our algorithm might still never terminate: The algorithm does not detect nested loops. Hence, there is no mechanism to ensure that the inner and outer loops are unrolled fairly. For example, the algorithm might find infinitely many Elementary Bounded Languages of the form $(acb)^*$, $(accb)^*$, $(acccb)^*$ $\ldots$ while the counterexample is the word $acbaccb$.

A similar problem occurs for loops containing any kind of choice operator: If the language is $a\,\{c,d\}^*\,b$, then the algorithm might produce the Elementary Bounded Languages $ac\,\{d\}^*\,b$, $acc\,\{d\}^*\,b$, $\ldots$ and never find the word $adcb$.

Conceptually, our algorithm can decide the problem in finite time if the language of potential counterexamples is a union of finitely many Elementary Bounded Languages. Practically, every refinement causes an additional intersection and therefore increases the state space. Thus, there are even conceptually decidable instances that still require more memory consumption than physically feasible.

Furthermore, our algorithm can also be embedded in a deciding procedure. This procedure executes our program repeatedly but alters several parameters: If an execution does not find a solution, the number of reduction steps will be reduced and the number of refinement steps will be increased. Eventually, number of allowed reduction steps will be reduced to 0. Then, the procedure is a decider since this is equivalent to the conventional approach. Practically, this is of course limited by physical boundaries.

Wrong Alphabets Actual implementations suffer from a problem that does not exist conceptually: In theory, every automaton has a specified alphabet. If we unite two automata, the alphabet is the union of the previous alphabets. If we intersect them, the alphabet is the intersection of the alphabets or a subset. Knowing the alphabet is important when introducing fail states. If a letter can never occur at a certain place in a run, then the fail states mark a dead end in the automaton. To this end, we introduce transitions to the fail states reading all letters of the alphabet that are not allowed.

A problem arises when the assumed alphabet is not the actual alphabet of the intersected languages: The automata might get transitions reading letters that never occur in any word that could possibly be in the intersection. Yet, these words are considered as potential counterexamples. Our algorithm then produces Elementary Bounded Languages that will be immediately rejected as spurious.

Since the underlying automata library Fare assumes the alphabet to be entire Unicode, this leads to two problems: It is impossible on actual hardware to actually refine the counterexample automaton until all Unicode character have been excluded. Some of the Unicode characters are control symbols or special characters for Regular Expressions. If the resulting regular expression is syntactically invalid, the counterexample language cannot be refined at all. If it is syntactically valid, the resulting regular expression might still exclude other counterexamples than intended. For example, parentheses at the place of letters change a language like abc^* to $(b)^* = b^*$. In the worst case, we would exclude actual counterexamples, which possibly renders the result wrong.

We work around these problems by only considering counterexamples on a defined, limited range of Unicode characters. Currently, this range is set to `a-z`. It can be set to any range that is suitable for the intended application if the range does not contain Unicode control symbols and special characters of regular expressions.

10.1 Possible Modifications and Alternative Use-Cases

We now discuss various options how our library and our techniques enable the development of tools aside the classical CEGAR loop. We also consider possible modifications and improvements of the procedure that were not covered in this work as the goal was to show feasibility for the general case.

Automatic Detection of Alphabets As said before, the implementation currently restricts the alphabet to a fixed character range when testing counterexamples. This is not an ideal solution as there are Elementary Bounded Languages that contain characters that never occur in the actual intersection. A better solution is a dynamic specification for the alphabet. The search for the Elementary Bounded Language could then be restricted to characters that are specified in the alphabets of every input automaton. The actual alphabet of the intersection might still be smaller: The languages a^*b and ba^* share the alphabet $\{a, b\}$ while the alphabet of the intersection b is just $\{b\}$. Yet, the intersection of the input automata is an over-approximation for the alphabet of the intersection.

However, it is not always clear which characters appear in the language of an automaton if the automaton is automatically generated. If the automaton is generated from a regular expression using complementation, the language might use letters that never occur in the regular expression itself. Thus, there are cases where an automatic detection of the used alphabet of an automaton is needed.

The alphabet could be determined by considering live transitions. A state is live if there is an accepting run using this state. The method `GetLiveStates` of the class `Fare.Automaton` determines these. A live transition is a transition between such states. The alphabet of a language is certainly the set of letters that are read along live transitions.

Parikh Image Pre-Check Parikh Images only consider the frequency of letters, not their order. In our procedure, we prepare the languages so that the order of letters is fixed and each Parikh vector represents a single word. To this end, we used Elementary Bounded Languages. They enforce the order of infixes.

We can use the procedure without over-approximation and Elementary Bounded Languages to rule out some negative cases. If we find empty intersections of the Parikh Images, then the actual intersection is empty as well. If the intersection of the Parikh Images is non-empty, then the actual intersection might still be empty although there are words in all of the intersected languages with the same frequency of the letters. We could use the existing module to extract of the Parikh Image Presburger Formula in a different way:

Before over-approximating the intersection $\mathcal{L}(Reg_1) \cap \cdots \cap \mathcal{L}(Reg_k)$, we can check $\mathcal{P}(\mathcal{L}(Reg_1)) \cap \cdots \cap \mathcal{P}(\mathcal{L}(Reg_k)) \cap \mathcal{P}\left(\overline{\mathcal{L}(Reg)}\right) \overset{?}{=} \emptyset$. If it holds, we know that the intersection satisfies the specification $\mathcal{L}(Reg)$.

If it does not hold, we need to compute the over-approximation and intersect it with the complement of the specification to get the language of counterexample candidates $\mathcal{L}_C$. Since the counterexample candidates are included in the complement of the specification $\mathcal{L}_C \subseteq \overline{\mathcal{L}(Reg)}$, we might have ruled out the words whose Parikh vectors are ambiguous. Thus, we check $\mathcal{P}(\mathcal{L}(Reg_1)) \cap \cdots \cap \mathcal{P}(\mathcal{L}(Reg_k)) \cap \mathcal{P}(\mathcal{L}_C) \overset{?}{=} \emptyset$ again.

If it holds, the specification was satisfied. If it does not, we just continue with the CEGAR-loop as before. The additional checks do not alter any of the languages, the rest of the procedure will continue in the same way. However, the checks might help to avoid the CEGAR-loop.

CEGAr (Less Refinement) Our procedure currently refines the language of potential counterexamples after every spuriousness check. The refinement is rather expensive as it requires an actual intersection in each step. The advantage is that certainly no potential counterexample is ever checked twice in the spuriousness check. Every Elementary Bounded Language is free of any words that had been in a previous Elementary Bounded Language. However, the required state space to keep track of the potential counterexamples might exceed the cross-product automaton of the actual intersection after a few loops.

Hence, it might pay off to run several spuriousness checks without refining the language. The idea is to enumerate paths to accepting states: There are finitely many accepting runs that use every state at most once. There are also finitely many repetition-free runs from each state that return to the state itself. We can then build Elementary Bounded Languages by adding the loops at the corresponding places in the words that are read by the accepting runs. If we limit the number of loops added at a single state, we get finitely many Elementary Bounded Languages for the spuriousness check.

A practical advantage is the ability to run the spuriousness checks in parallel: If we extract several Elementary Bounded Languages from the same automaton, then we can just copy the automata and test several Elementary Bounded Languages at once.

The question whether and how often we refine is a trade-off: We can easily generate more Elementary Bounded Languages in an automaton than we compute possible refinements. Yet, it might still be appropriate to refine the automaton occasionally in order to reduce the number of overlapping Elementary Bounded Languages.

A conceptual advantage is that a structured enumeration of all Elementary Bounded Languages in the language of potential counterexamples makes the procedure a semi-decider for non-inclusion: Whenever an intersection is not included in the specification, there is a shortest counterexample and an Elementary Bounded Language. If the procedure considers all Elementary Bounded Languages in a fixed order, the language containing the shortest counterexample will be found in finite time. A possible order is the lexicographic order on the regular expression of the Elementary Bounded Languages.

CEGaR (Less Abstraction) The Elementary Bounded Languages are subsets of a language of potential counterexamples. We obtained this language by over-approximating the intersection and intersecting it with the complement of the specification. The idea of the over-approximation is to restrict the Elementary Bounded Languages to counterexamples that are likely to be non-spurious. If we can increase the throughput of Elementary Bounded Languages in the spuriousness check, we can afford a lower likeliness of non-spurious counterexamples.

Hence, it might be worth trying to use the previously mentioned variant with less refinement steps together with a variant that works without any over-approximation: We extract the Elementary Bounded Language right from the complement of the specification. A high frequency of spuriousness checks might then help to still find counterexamples or even entirely rule out the counterexample language.

Elementary Bounded Anti-Specifications An application of a variant without an initial over-approximation and without refinement steps are Elementary Bounded Anti-Specifications. Instead of giving the specification as a regular language that needs to include every word in the intersection, we define finitely many languages that are forbidden. These anti-specifications need to be Elementary Bounded Languages. Hence, we can use the existing modules of the implementation to check whether any of the finitely many anti-specifications contains non-spurious counterexamples. Our tools therefore serve as a decider for this class of problems.

Non-Regular Elementary Bounded Languages Currently, our procedure only extracts regular Elementary Bounded Languages. The reason is the way we prepare the input languages before generating the Parikh Image Presburger Formula: We first replace the infixes of the Elementary Bounded Language by single letters. We then intersect the modified automata with Elementary Bounded Languages. We do so in order to enforce the correct order of the letters representing infixes. Since our implementation for the extraction of the Parikh Images relies on finite automata, we need the intersection to be regular. This cannot be guaranteed if the Elementary Bounded Language is non-regular.

However, we can still enforce the correct order of the infixes by intersecting with a superset of the Elementary Bounded Language: If the Elementary Bounded Language $\{w_1^{x_1} \dots w_l^{x_l} \mid (x_1, \dots, x_l) \in L\}$ is represented as $\{a_1^{x_1} \dots a_l^{x_l} \mid (x_1, \dots, x_l) \in L\}$, we intersect each of the input automata with $a_1^* \dots a_l^*$.

The resulting languages still represent words that are not in the Elementary Bounded Language although the order of the infixes is the same. We still need to consider the frequencies of the infixes and their relation. In order to intersect the Parikh Image of the intersection with the one of the Elementary Bounded Language, we need to modify the Presburger Formula: If the set L is semi-linear, we can express it by a Presburger Formula. The conjunct of the Parikh Image Presburger Formula of the intersection of the modified input automata and the Presburger Formula of L describes those words that are both in the intersection and in the Elementary Bounded Language.

A difficulty is the exclusion of spurious Elementary Bounded Languages. If they are non-regular, their complement will not be regular. Hence, we cannot always intersect the language of potential counterexamples with the complement of the Elementary Bounded Languages and still maintain a regular language of potential counterexamples. Thus, we either choose an approach with no refinement steps or refine by only excluding a regular subset of the Elementary Bounded Language.

Therefore, our procedure can be extended to also consider non-regular Elementary Bounded Languages of potential counterexamples.

We have shown that our procedure is extensible in many ways. It can easily be adapted to address related problems. The suggested variants offer opportunities to avoid the bottlenecks we identified. Together, the procedure itself and its variants extend the range of the practically computable instances of our language-theoretic problem. Hence, the techniques we introduced and suggested can be used to solve many classes of verification problems.

References

[AVL62] Georgy M. Adelson-Velsky and Evgenii M. Landis. An algorithm for the organization of information. In *Doklady Akademii Nauk SSSR*, volume 3, pages 1259–1263, 1962.

[CGJ$^+$00] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *International Conference on Computer Aided Verification*, pages 154–169. Springer, 2000.

[Coo72] David C Cooper. Theorem proving in arithmetic without multiplication. *Machine Intelligence*, 7(91-99):300, 1972.

[Esp97] Javier Esparza. Petri nets, commutative context-free grammars, and basic parallel processes. *Fundamenta Informaticae*, 31(1):13–25, 1997.

[Fan14] Dave Fancher. *Book of F# - Breaking Free with Managed Functional Programming*. No Starch Press, Munich, Germany, first edition edition, 2014.

[GS66] Seymour Ginsburg and Edwin Spanier. Semigroups, presburger formulas, and languages. *Pacific journal of Mathematics*, 16(2):285–296, 1966.

[HW73] Carl Hierholzer and Christian Wiener. Über die Möglichkeit, einen Linienzug ohne Wiederholung und ohne Unterbrechung zu umfahren. *Mathematische Annalen*, 6(1):30–32, 1873.

[JR93] Tao Jiang and Bala Ravikumar. Minimal nfa problems are hard. *SIAM Journal on Computing*, 22(6):1117–1141, 1993.

[LCMM12] Zhenyue Long, Georgel Calin, Rupak Majumdar, and Roland Meyer. Language-theoretic abstraction refinement. In *International Conference on Fundamental Approaches to Software Engineering*, pages 362–376. Springer, 2012.

[LOW15] Antonia Lechner, Joël Ouaknine, and James Worrell. On the complexity of linear arithmetic with divisibility. In *Proceedings of the 2015 30th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 667–676. IEEE Computer Society, 2015.

[Moo71] Frank R Moore. On the bounds for state-set size in the proofs of equivalence between deterministic, nondeterministic, and two-way finite automata. *IEEE Transactions on computers*, 100(10):1211–1214, 1971.

[Par66] Rohit J Parikh. On context-free languages. *Journal of the ACM (JACM)*, 13(4):570–581, 1966.

[Sca84] Bruno Scarpellini. Complexity of subcases of presburger arithmetic. *Transactions of the American Mathematical Society*, 284(1):203–218, 1984.

[TV05] Deian Tabakov and Moshe Y Vardi. Experimental evaluation of classical automata constructions. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 396–411. Springer, 2005.

[VSS05] Kumar Neeraj Verma, Helmut Seidl, and Thomas Schwentick. On the complexity of equational Horn clauses. In *International Conference on Automated Deduction*, pages 337–352. Springer, 2005.