What's Decidable about Availability Languages?

Parosh Aziz Abdulla[1], Mohamed Faouzi Atig[1], Roland Meyer[2], and Mehdi Seyed Salehi[3]

1 **Uppsala University, Uppsala, Sweden**
`{parosh,mohamed_faouzi.atig}@it.uu.se`
2 **University of Kaiserslautern, Kaiserslautern, Germany**
`meyer@cs.uni-kl.de`
3 **Sharif University of Technology, Tehran, Iran**
`seyedsalehi@ce.sharif.edu`

Abstract

We study here the algorithmic analysis of systems modeled in terms of availability languages. Our first main result is a positive answer to the emptiness problem: it is decidable whether a given availability language contains a word. The key idea is an inductive construction that replaces availability languages with Parikh-equivalent regular languages. As a second contribution, we solve the intersection problem modulo bounded languages: given availability languages and a bounded language, it is decidable whether the intersection of the former contains a word from the bounded language. We show that the problem is NP-complete. The idea is to reduce to satisfiability of existential Presburger arithmetic. Since the (general) intersection problem for availability languages is known to be undecidable, our results characterize the decidability border for this model. Our last contribution is a study of the containment problem between regular and availability languages. We show that safety verification, i.e., checking containment of an availability language in a regular language, is decidable. The containment problem of regular languages in availability languages is proven undecidable.

1998 ACM Subject Classification D.2.4 Software/Program Verification, F.4 Mathematical Logic and Formal Languages

Keywords and phrases Availability, formal languages, emptiness, decidability

Digital Object Identifier 10.4230/LIPIcs.FSTTCS.2015.192

1 Introduction

Availability is an important concept in the dependability analysis of unreliable reactive systems. In such systems, components may fail for some time and recover later. In other words, the system may be available for a certain amount of time during the observation period. The property of interest for us in such systems is (interval) availability which specifies the proportion of the time in which the system is available for use.

Availability for continuous systems has been extensively studied in the literature [4, 16]. Studying availability over a discrete domain, however, is quite new, but it can be useful. With appropriate approximations, it should be possible to model the availability characteristics of a system. With a discrete domain at hand, one may hope for automated analyses. This paper can be understood as providing evidence for the latter claim.

Regular availability expressions have been introduced in [11] as a model for discrete availability aspects. Availability expressions extend regular expressions by an additional operator. This so-called occurrence constraint associates a positive or negative availability with the symbols of the alphabet and poses requirements on the accumulated availability of words. To give an example, the availability expression

$$((up + down)^*\checkmark)_{\#up-\#down>0}$$

requires a system uptime of at least 50%. To achieve this, the semantics evaluates the occurrence constraint $\#up - \#down > 0$ every time the execution meets a symbol $\checkmark$.

Algorithmic analysis looks at system models from a language-theoretic point of view. Here, availability expressions are interesting because their language class is incomparable with basic classes like context-free languages. Therefore, language-specific properties like emptiness or intersection are non-trivial for them.

The earlier work [11] proved the undecidability of checking whether the intersection of two availability languages is empty or not. The *emptiness problem* itself, however, remained open. Emptiness may be considered the key problem in algorithmic verification: correctness requirements are typically stated as $\mathcal{L}_1 \subseteq \mathcal{L}_2$ and then rephrased as $\mathcal{L} = \emptyset$ with $\mathcal{L} = \mathcal{L}_1 \cap \overline{\mathcal{L}_2}$. The first contribution of this article is a positive answer to the emptiness problem of regular availability languages. We provide an algorithm that takes an availability expression *rae* and decides whether the associated language $\mathcal{L}(rae)$ is empty.

Technically, we show that availability languages have regular approximations that are exact with respect to emptiness. More precisely, given *rae* we construct a regular expression *reg* with the same Parikh image [14]: $\Pi(\mathcal{L}(rae)) = \Pi(\mathcal{L}(reg))$. The idea is to proceed by induction on the structure of availability expressions. In the base case, we directly construct a one-counter automaton M that captures the language of an expression $(reg)_{cstr}$. (Here, *cstr* is an occurrence constraint as in the above example.) With Parikh's result, we then obtain a regular language that is Parikh-equivalent with $\mathcal{L}(M)$. In the induction step, we apply this result to iteratively replace availability expressions by regular expressions. This leads to an algorithm for solving the emptiness problem with a non-elementary complexity. This is due to the exponential blow-up encountered, at each induction step, when computing the Parikh image of a one-counter automaton.

Our second contribution is a refined study of the (undecidable) intersection problem. Rather than inspecting the full intersection $\mathcal{L}(rae_1) \cap \mathcal{L}(rae_2)$, we restrict the search to words from a given bounded expression $bl = w_1^* \dots w_m^*$. This under-approximate verification technique is also known as pattern-based verification [6], and generalizes the popular idea of bounded countext switching [15]. Our finding is that the problem is only NP-complete.

Technically, we give a reduction to satisfiability of existential Presburger arithmetic [17]. We first rewrite $\mathcal{L}(rae_1) \cap \mathcal{L}(rae_2) \cap \mathcal{L}(bl)$ as $(\mathcal{L}(rae_1) \cap \mathcal{L}(bl)) \cap (\mathcal{L}(rae_2) \cap \mathcal{L}(bl))$, a trick due to Ginsburg and Spanier [9]. Then we show how to capture the latter intersection by a formula $\exists \tilde{x}.\varphi_1(\tilde{x}) \wedge \varphi_2(\tilde{x})$ with $\tilde{x} = x_1, \dots, x_m$. Intuitively, the task of φ_k with $k = 1, 2$ is to count the occurrences of w_1 to w_m in the intersection $\mathcal{L}(rae_k) \cap \mathcal{L}(bl)$.

The actual challenge in the proof is to construct φ_k. We again proceed by induction. To invoke the hypothesis, we guess the part $bl' = w_i^* \dots w_j^*$ of the bounded expression bl that will be traversed when rae_k passes through a top-level constraint $(rae')_{cstr'}$. To our surprise, parts of length one ($i = j$) were difficult to handle and needed an auxiliary construction. Such a part may be traversed multiple times. Hence, representing it by a Presburger formula would lead to non-linearity. We show how to compute a finite automaton that captures the language $\mathcal{L}(bl') \cap \mathcal{L}((rae')_{cstr'})$. Here, we need visibility arguments and study the boundedness behavior of the one-counter automata representing availability languages.

Our last contribution is a study of the containment problem between regular languages and availability languages. First, we show the decidability of $\mathcal{L}(rae) \subseteq \mathcal{L}(reg)$. Note that this inclusion is a common formulation of safety verification problems. The proof is by a reduction to the emptiness problem of the language $\mathcal{L}(rae) \cap \overline{\mathcal{L}(reg)}$. As key argument, we establish closure of the class of availability languages under regular intersection. Second, we show the undecidability of checking whether a regular language is included in an availability language. The proof is by a reduction from the halting problem for two-counter automata (which is known to be undecidable [13]).

Related Work. We already discussed the relation of our results to availability analysis. We now concentrate on the related work in formal languages and verification. The work [11] introduced regular availability expressions and proposed a corresponding automaton model that captures availability languages in an operational rather than a declarative way. Moreover, it gave a synthesis algorithm that determines a most liberal implementation of an availability requirement. The final result was the undecidability of intersection emptiness. We focus here on algorithmic problems of the latter form, and obtain positive results. We show the decidability of emptiness, NP-completeness of a restricted intersection problem, and decidability of safety verification. We believe that these positive results, in particular the low complexity of the intersection problem modulo bounded languages, should motivate further studies of availability languages. It should also be noted that we generalize the model [11] towards stronger occurrence constraints.

Availability expressions introduce occurrence constraints to influence the use of symbols. With this numeric aspect, Parikh images [14] proved to be a valuable tool in the manipulation of availability languages. There are other language-theoretic models that employ Parikh images [12, 3, 2, 1, 18]. In all these models (variants of so-called Parikh automata and Presburger regular expressions), the final acceptance of a word depends on the number of occurrences of letters. What is different in our model is that we admit intermediary occurrence checks. These checks can be used as guards to influence the future system behavior, as opposed to post-mortem acceptance checks. There is no bound on the number of such intermediary measurements so that there is no immediate reduction to the aforementioned models.

Concerning bounded languages, Ganty et al. [8] showed how to construct from a context-free language a context-free bounded language with the same Parikh image. Also in the context-free setting, Esparza and Ganty proposed the intersection problem modulo bounded languages [6], a work that inspired our second contribution. Hague and Lin generalized this result to pushdown automata with reversal-bounded counters [10]. Our underlying model is regular rather than context free but admits an unbounded number of checks on the counters.

Weighted languages form another line of related work [5]. The idea is to let an automaton manipulate weights from a semi-ring. Actually, weighted automata admit very general semi-rings while we focus on the occurrence of letters. In contrast, our occurrence constraints can influence the system behavior while weighted automata only provide an analysis.

2 Availability Languages

Regular availability expressions extend regular expressions by occurrence constraints on the letters. The model was introduced in [11] and is presented here in a generalized form. The idea of occurrence constraints is to require a specified ratio on the occurrence of letters. For example, the word $w = aab$ has more letters a than b, which means the occurrence constraint $\#a - \#b > 0$ holds. To be more precise, the occurrence constraint is checked at the places marked by a distinguished symbol $\checkmark$. It does not need to hold throughout the word. With this, the availability expression $(a^*b^*\checkmark)_{\#a-\#b>0}$ denotes the language $\{a^nb^m \mid n > m\}$. Throughout the paper, we assume an underlying finite alphabet Λ.

An *occurrence constraint cstr* takes the form

$$t + \sum_{a\in\Lambda} k_a \cdot \#a > 0 \qquad \text{with } t, k_a \in \mathbb{Z}. \tag{1}$$

The set of *regular availability expressions* is defined as follows:

$$rae \ ::= \ a \mid \varepsilon \mid \emptyset \mid \checkmark \mid rae + rae \mid rae.rae \mid rae^* \mid (rae)_{cstr} \ .$$

Here, $a \in \Lambda$, $\checkmark \notin \Lambda$ is the distinguished symbol, and *cstr* is defined as above. We use *reg* to indicate that an availability expression actually is a *regular expression*, which means it does not contain $\checkmark$ nor *cstr*. The *depth* of an availability expression is the nesting depth of occurrence constraints $(rae)_{cstr}$. An occurrence constraint $(rae)_{cstr}$ is *top-level* in rae' if it is not covered by another $(-)_{cstr'}$ in the syntax tree of rae'. The syntactic *size* of an availability expression *rae* is denoted by $|rae|$. The definition is as expected, every piece of syntax contributes to it.

The semantics of availability expressions is in terms of finite words, $\mathcal{L}(rae) \subseteq (\Lambda \cup \{\checkmark\})^*$, and defined inductively as follows:

$$\mathcal{L}(a) := \{a\} \qquad \mathcal{L}(\checkmark) := \{\checkmark\} \qquad \mathcal{L}(\varepsilon) := \{\varepsilon\}$$
$$\mathcal{L}(\emptyset) := \emptyset \qquad \mathcal{L}(rae_1 + rae_2) := \mathcal{L}(rae_1) \cup \mathcal{L}(rae_2) \qquad \mathcal{L}(rae^*) := \mathcal{L}(rae)^*$$
$$\mathcal{L}(rae_1.rae_2) \ := \mathcal{L}(rae_1).\mathcal{L}(rae_2) \qquad \mathcal{L}(rae_{cstr}) \ := \mathcal{L}(rae)_{cstr}.$$

To define the semantics of occurrence constraints, $\mathcal{L}(rae)_{cstr}$, we need Parikh images and projections. The *Parikh image* of a word w over $\Lambda \cup \{\checkmark\}$ is a function $\Pi(w) : \Lambda \to \mathbb{N}$ that returns for every $a \in \Lambda$ the number of occurrences of a in w, in symbols $\Pi(w)(a) := |w|_a$. Let $\Gamma \subseteq \Lambda \cup \{\checkmark\}$ and let w be a word over $\Lambda \cup \{\checkmark\}$. The *projection* of w to Γ, denoted by $\pi_\Gamma(w)$, is the result of removing all symbols outside Γ from w. Using these concepts, operator $\mathcal{L}_{cstr}$ checks that each prefix ending in $\checkmark$ satisfies the occurrence constraint, and projects the remaining words to Λ:

$$\begin{aligned}\mathcal{L}_{cstr} &:= \{\pi_\Lambda(w) \mid w \in \mathcal{L}^{\checkmark}_{cstr}\}\\ \mathcal{L}^{\checkmark}_{cstr} &:= \{w \in \mathcal{L} \mid t + \sum_{a\in\Lambda} k_a \cdot \Pi(w_1)(a) > 0 \text{ for all } w_1.\checkmark.w_2 = w\} \ .\end{aligned}$$

Availability languages are incomparable with context-free languages [11]. For example, the language $\{a^nb^nc^n \mid n \in \mathbb{N}\}$ can be generated by the regular availability expression

$$((((a^*b^*c^*\checkmark)_{1+\#a-\#b>0}\checkmark)_{1+\#b-\#a>0}\checkmark)_{1+\#c-\#a>0}\checkmark)_{1+\#a-\#c>0}$$

but it is not a context-free language. The language of words $w.w^{reverse}$ in turn cannot be represented by a regular availability expression.

3 Emptiness

The *emptiness problem* for availability languages consists in checking, for a given regular availability expression *rae*, whether $\mathcal{L}(rae) = \emptyset$. Our first main result is the decidability of the emptiness problem. The solution is inductive. We first discuss the emptiness problem for

$(reg)_{cstr}$. Via one-counter automaton, we show how to obtain a Parikh-equivalent regular expression. Then we use this result to turn a general availability expression rae into a Parikh-equivalent regular expression reg: $\Pi(\mathcal{L}(rae)) = \Pi(\mathcal{L}(reg))$. With this correspondence, the two languages have the same emptiness status: $\mathcal{L}(rae) = \emptyset$ iff $\mathcal{L}(reg) = \emptyset$.

3.1 One-Counter Automata

We use a variant of one-counter automata (1CM) with counters over the integers $\mathbb{Z}$ rather than the natural numbers $\mathbb{N}$. A 1CM is a non-deterministic finite state automaton equipped with a counter that can be incremented, decremented, and compared to zero. Formally, the automaton is a 5-tuple $M = (Q, \Lambda, \Delta, s, F)$ where Q is a finite set of states with initial state $s \in Q$ and final states $F \subseteq Q$. The transitions in $\Delta \subseteq Q \times (\Lambda \cup \{\varepsilon\}) \times Op \times Q$ are labelled by a letter from Λ and equipped with an operation from $Op := \{> 0, \leq 0\} \cup \{\mathsf{add}(m) \mid m \in \mathbb{Z}\}$. For the semantics, we define labelled transitions between configurations from $Q \times \mathbb{Z}$. The automaton accepts a word $w \in \Lambda^*$ if there is a sequence of configurations that is labelled by w and ends in a final state. The language $\mathcal{L}(M)$ is the set of all words accepted by M.

A 1CM M over the integers can be compiled down to a language-equivalent 1CM $M_{\mathbb{N}}$ over the natural numbers. To adjust the semantics, over the naturals an addition $\mathsf{add}(-m)$, $m \in \mathbb{N}$, is enabled only if the resulting counter value stays at least zero. The idea of the translaton is to duplicate each control state. So state q in M yields q_+ and q_- in $M_{\mathbb{N}}$. In q_+, the counter value represents a positive value in the original automaton. In q_-, the value represents the corresponding negative value in the original automaton. Clearly, a transition > 0 from q will be copied to q_+ and will be removed from q_-. A transition ≤ 0 from q will be copied to q_+. In q_-, we have the transition without a condition (represented by $\mathsf{add}(0)$). Consider a decrement $(q, a, \mathsf{add}(-m), q')$, $m \in \mathbb{N}$. Besides the corresponding transitions at q_+ and q_-, every pair $x, y \in \mathbb{N}$ so that $x + y = m$ yields a transition sequence $(q_+, a, \mathsf{add}(-x), p)(p, \varepsilon, \leq 0, p')(p', \varepsilon, \mathsf{add}(y), q'_-)$ where p, p' are two intermediary states used only in the simulation of the decrement transition. The construction is similar for increments.

The 1CM $M_{\mathbb{N}}$ in turn can be understood as pushdown automata with two stack symbols. One of them is used to mark the bottom of the stack, the other represents the counter value. As a consequence of the two constructions, the languages of 1CM over $\mathbb{Z}$ are context free. With Parikh's theorem, we can construct a regular language with the same Parikh image. A simple construction which takes a context-free language and returns a Parikh-equivalent finite automaton has been proposed in [7]. We summarize the argumentation.

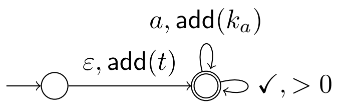
▶ **Lemma 3.1.** *Given a 1CM M, one can construct reg with* $\Pi(\mathcal{L}(M)) = \Pi(\mathcal{L}(reg))$.

1CM (over $\mathbb{Z}$ and over $\mathbb{N}$) are effectively closed under regular intersection and projection.

▶ **Lemma 3.2.** *Given 1CM M and reg, we can construct M' with $\mathcal{L}(M') = \mathcal{L}(M) \cap \mathcal{L}(reg)$. Given $\Gamma \subseteq \Lambda$, we can construct M' with $\mathcal{L}(M') = \pi_{\Gamma}(\mathcal{L}(M))$.*

3.2 From Availability to Regular Expressions

To check emptiness of $(reg)_{cstr}$, we first construct a 1CM M that accepts words over $\Lambda \cup \{\checkmark\}$ only based on the constraint $cstr$. With the notation from the previous section, it accepts all $w \in (\Lambda \cup \{\checkmark\})^*$ such that $\{w\}^{\checkmark}_{cstr} = \{w\}$. Assume $cstr$ takes the Form (1). Automaton M, depicted to the right, has two states s and f, the former being initial and the latter being final, respectively. There is an ε-labelled transition from s



to f with operation $\mathsf{add}(t)$. The transition initializes the counter with constant t from the constraint. For every letter $a \in \Lambda$, we have a loop at state f that adds coefficient $k_a \in \mathbb{Z}$. Moreover, for $\checkmark$ we have a loop at f with check > 0.

▶ **Lemma 3.3.** *Let $w \in (\Lambda \cup \{\checkmark\})^*$. Then $w \in \mathcal{L}(M)$ if and only if $\{w\}^{\checkmark}_{cstr} = \{w\}$.*

To take the regular expression *reg* into account, we use closure under regular intersection. So given *reg* and M above, we compute M' with $\mathcal{L}(M') = \mathcal{L}(M) \cap \mathcal{L}(reg)$. To remove symbol $\checkmark$, we project $\mathcal{L}(M')$ to Λ. This yields the language of $(reg)_{cstr}$.

▶ **Lemma 3.4.** $\pi_\Lambda(\mathcal{L}(M')) = \mathcal{L}((reg)_{cstr})$.

By closure under projection, $\mathcal{L}((reg)_{cstr})$ is again a one-counter language. With Lemma 3.1, the language can be represented by a regular language, up to Parikh-equivalence.

▶ **Proposition 3.5.** *One can construct reg' with $\Pi(\mathcal{L}((reg)_{cstr})) = \Pi(\mathcal{L}(reg'))$.*

For the general case, consider *rae* and focus on an occurrence constraint of maximal depth. By maximality, it has the shape $(reg)_{cstr}$. We apply Proposition 3.5 to construct a regular language reg' with the same Parikh image, $\Pi(\mathcal{L}(reg')) = \Pi(\mathcal{L}((reg)_{cstr}))$. We now replace $(reg)_{cstr}$ by reg' within *rae*, resulting in rae'. The languages of *rae* and rae' still coincide, up to Parikh-equivalence:

$$\Pi(\mathcal{L}(rae')) = \Pi(\mathcal{L}(rae)).$$

The reason is that the Parikh image of $\mathcal{L}((reg)_{cstr})$ keeps the number of symbols (but may not retain their order), and this is what is needed to evaluate further occurrence constraints in *rae*. We repeat the procedure inductively on rae'. Eventually, we have eliminated all occurrence constraints and hence arrived at a regular expression.

▶ **Theorem 3.6.** *One can construct reg with $\Pi(\mathcal{L}(rae)) = \Pi(\mathcal{L}(reg))$. Hence, the emptiness problem for availability languages is decidable.*

Observe that the procedure has a non-elementary complexity. This is due to the exponential blow-up encountered, at each induction step, when computing the Parikh image of a 1CM.

4 Intersection Modulo Bounded Languages

The intersection problem $\mathcal{L}(rae_1) \cap \mathcal{L}(rae_2) \neq \emptyset$ is known to be undecidable, already for two availability expressions [11]. We now show that the problem remains decidable, actually only NP-complete, if we require the words in the intersection to belong to the language of a bounded expression of the form $bl = w_1^* \dots w_m^*$. To be precise, we study the following problem that we will refer to as IBL, *intersection modulo bounded languages*: Given rae_1 to rae_n and bl, is $\bigcap_{i=1}^n \mathcal{L}(rae_i) \cap \mathcal{L}(bl) \neq \emptyset$? Our second main result is as follows.

▶ **Theorem 4.1.** IBL *is* NP*-complete for availability expressions of fixed depth.*

We explain the proof approach and elaborate on the side condition. The approach is inspired by Ginsburg and Spanier's [9] and has also been used in [6]. We first rewrite the intersection:

$$\left(\bigcap_{i=1}^{n} \mathcal{L}(rae_i) \right) \cap \mathcal{L}(bl) = \bigcap_{i=1}^{n} \left(\mathcal{L}(rae_i) \cap \mathcal{L}(bl) \right) . \tag{2}$$

Let $bl = w_1^* \dots w_m^*$. Our technical contribution (Proposition 4.8) is then to compute in polynomial time an existential Presburger formula $\varphi_i(x_1, \dots, x_m)$ that captures the words in an intersection $\mathcal{L}(rae_i) \cap \mathcal{L}(bl)$ as follows: $w_1^{k_1} \dots w_m^{k_m} \in \mathcal{L}(rae_i) \cap \mathcal{L}(bl)$ if and only if $(k_1, \dots, k_m) \models \varphi_i$. Intuitively, the Presburger formula counts how often each word in the bounded expression occurs. With this, the intersection between the languages $\mathcal{L}(rae_i) \cap \mathcal{L}(bl)$ on the right-hand side of Equation (2) is represented by the intersection of the solution spaces of the corresponding formulas φ_i. Indeed, there is a word in the intersection if and only if there are coefficients k_1 to k_m on which all formulas agree. This is equivalent to satisfiability of $\exists x_1 \dots \exists x_m : \bigwedge_{i=1}^{n} \varphi_i(x_1, \dots, x_m)$. Since the formulas are computable in polynomial time (Proposition 4.8) and satisfiability of existential Presburger is in NP [17], we obtain an upper bound for IBL.

For the polynomial-time computability to hold, we have to assume the depth of the availability expressions given as input to be bounded from above by a constant. The need for a fixed depth comes with the proof approach. We construct the Presburger formulas by an induction on the depth of availability expressions. Each step in this induction is polynomial-time computable. The composition of the polynomials, however, only stays polynomial if we assume it to be fixed.

For the lower bound, NP-hardness already holds in the case of regular rather than availability languages, a single letter alphabet, and a fixed pattern, due to [6].

4.1 Bounded Languages and Presburger Arithmetic

In the literature, bounded languages are defined as (potentially non-regular) subsets of $w_1^* \dots w_m^*$. As there is no risk of confusion, we decided to adopt the terminology. For our proofs, we will have to deal with leading and trailing words. So we will also refer to $bl = w_0.w_1^* \dots w_m^*.w_{m+1}$ as a bounded expression. The *length* of bl is the number of starred words, $m \in \mathbb{N}$. A *part* of bl is a language $u.w_i^* \dots w_j^*.v$ with $1 \leq i \leq j \leq m$, u a suffix of w_{i-1} or w_i, and v a prefix of w_j or w_{j+1}. Alternatively, $u.v$ may form an infix of some w_i and the iterated part is missing.

Presburger arithmetic is the first-order logic of the natural numbers with addition but without multiplication. Given a formula $\varphi(x_1, \dots, x_n)$ with free variables x_1 to x_n, we use $\mathcal{S}(\varphi)$ for the solution space: the set of valuations $(k_1, \dots, k_n)$ that satisfy the formula. We are interested in the *existential fragment* of Presburger, denoted by $\exists$PA and defined by

$$t ::= 0 \mid 1 \mid x \mid t_1 + t_2 \qquad \varphi ::= t_1 = t_2 \mid t_1 > t_2 \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \exists x.\varphi \,.$$

A result by Verma, Seidl, and Schwentick shows how to capture the Parikh images of context-free languages by existential Presburger.

▶ **Proposition 4.2** ([19]). *Given a context-free grammar G, one can compute in linear time an $\exists$PA formula φ satisfying $\mathcal{S}(\varphi) = \Pi(\mathcal{L}(G))$.*

4.2 NP Upper Bound

Our goal is to construct an $\exists$PA formula for $\mathcal{L}((rae)_{cstr}) \cap \mathcal{L}(bl)$. Roughly, the approach is to represent the intersection by a 1CM and then obtain the $\exists$PA formula with Proposition 4.2. More precisely, the construction is by induction on the depth of availability expressions, and the challenge is to handle the top-level occurrence constraints $(rae')_{cstr'}$ within $(rae)_{cstr}$. To invoke the hypothesis, the idea is to precompute the intersection of $(rae')_{cstr'}$ with parts bl' of the bounded language bl. This, however, requires care. We will have to treat parts of length at most one and parts of length at least two substantially different. As a result, we will have to invoke different hypotheses. We explain the difficulties.

For parts of length at most one, the intersection $\mathcal{L}((rae')_{cstr'}) \cap \mathcal{L}(bl')$ may be entered several times. Indeed, bl' may be $u.w_i^*.v$ with u a suffix of w_i and v a prefix of w_i. If we only had a Presburger formula to represent the intersection, re-entering the intersection would lead to a non-linear constraint. Instead, we show (in a separate induction, leading to Proposition 4.3) how to compute a finite automaton representing the intersection. This automaton can then be plugged into the overall 1CM construction.

For parts $bl' = u.w_i^* \dots w_j^*.v$ of length at least two, we cannot re-enter an intersection $\mathcal{L}((rae')_{cstr'}) \cap \mathcal{L}(bl')$, simply because $j > i$. This allows us to represent the intersection by an ∃PA formula, which we obtain by a simple invocation of the induction hypothesis. We incorporate this formula into the ∃PA formula for the overall construction. Note that the intersection with a bounded language of length at least two is not necessarily a regular language. This means the automaton trick for length at most one does not work.

4.2.1 Automata for Length at most One

In the following, we show how to construct, in polynomial time, a finite state automaton recognizing the intersection of an availability expression and a part of a bounded expression of length at most one.

▶ **Proposition 4.3.** *Consider availability expressions of fixed depth. Given such an expression $(rae)_{cstr}$ and a bounded language $bl = w_1.w^*.w_2$, one can compute in polynomial time an NFA A with $\mathcal{L}(A) = \mathcal{L}((rae)_{cstr}) \cap \mathcal{L}(bl)$.*

The rest of this section is devoted to the proof of Proposition 4.3 which is done by induction on the depth of the availability expression.

Base Case. We first prove the base case when the availability expression is actually a regular expression (see Lemma 4.4).

▶ **Lemma 4.4.** *Given $(reg)_{cstr}$ and $bl = w_1.w^*.w_2$, one can compute in polynomial time an NFA A with $\mathcal{L}(A) = \mathcal{L}((reg)_{cstr}) \cap \mathcal{L}(bl)$.*

To prove Lemma 4.4, we first construct a 1CM for the intersection of the languages and in a second step compile this automaton down to a finite automaton. In Section 3, we have shown how to turn $(reg)_{cstr}$ into a 1CM with the same language. Since 1CM are closed under regular intersection, we can also determine $M = (Q, \Lambda, \Delta, s, F)$ with $\mathcal{L}(M) = \mathcal{L}((reg)_{cstr}) \cap \mathcal{L}(bl)$. Note that the construction of M works in polynomial time. To turn M into a finite automaton, the key observation is that M satisfies the following property referred to as **(Bound)**. There is a constant $b \in \mathbb{N}$ so that
(Bound-U) once we exceed b in a configuration, the counter will never drop below zero,
(Bound-L) once we fall below $-b$, the counter will not increase above zero again.
For the formalization, we focus on **(Bound-U)**, **(Bound-L)** is similar:

$$\forall (q, c) \in Q \times \mathbb{Z} \text{ with } (s, 0) \rightarrow^* (q, c) \text{ and } c > b :$$
$$\forall (q', c') \in Q \times \mathbb{Z} \text{ with } (q, c) \rightarrow^* (q', c') : \quad c' > 0.$$

The following, lemma shows that indeed the 1CM M satisfies the property **(Bound)** .

▶ **Lemma 4.5.** *M satisfies* **(Bound)** *with $b \in \mathbb{N}$ of size polynomial in $|rae|+|bl|$.*

We defer the proof of Lemma 4.5 for a moment and show that property **(Bound)** implies Lemma 4.4: the correspondingly bounded 1CM accept regular languages. The reason is that we only have to track the counter value precisely as long as it stays in the interval $[-b, b]$. Once this range is left, **(Bound-L)** and **(Bound-U)** indicate how to evaluate guards.

▶ **Lemma 4.6.** *Assume 1CM M satisfies* **(Bound)** *with $b \in \mathbb{N}$. There is an NFA A of size polynomial in $|M|+b$ with $\mathcal{L}(A) = \mathcal{L}(M)$.*

To prove Lemma 4.5, recall that $\mathcal{L}(M) = \mathcal{L}(bl) \cap \mathcal{L}((reg)_{cstr})$ with $bl = w_1.w^*.w_2$ and $cstr$ of the Form (1). The main observation is that M is a *visibly* 1CM in the following sense. A letter $a \in \Lambda$ always has the effect of adding the coefficient $k_a \in \mathbb{Z}$ to the counter, independent of the transition. This means no matter which transition sequence the automaton takes to process the word w in $bl = w_1.w^*.w_2$, the effect on the counter is always constant. We refer to it as $\mathit{effect}(w) \in \mathbb{Z}$. Note that the effect is homomorphic, $\mathit{effect}(w.v) = \mathit{effect}(w) + \mathit{effect}(v)$. We then do a case distinction according to whether w has a positive or a negative effect. Assume $\mathit{effect}(w) \geq 0$. We define the constant $b \in \mathbb{N}$ to be $|t| + \mathit{max}\{|\mathit{effect}(u)| \mid u$ an infix of $w_a.w_b$ where $w_a \in \{w_1, w\}$ and $w_b \in \{w, w_2\}\}$.

For **(Bound-L)**, we show that the counter never drops below $-b$ and hence the property trivially holds. For **(Bound-U)**, we consider a configuration with counter value $c > b$ and argue that the value stays above zero in every continuation of the transition sequence.

Induction step. Next, we show the induction step for the proof of Proposition 4.3. The induction step is established using the following lemma:

▶ **Lemma 4.7.** *Assume Proposition 4.3 holds for availability expressions of depth at most $n \in \mathbb{N}$. Consider $(rae)_{cstr}$ of depth $n+1$ and $bl = w_1.w^*.w_2$. One can compute in polynomial time an NFA A with $\mathcal{L}(A) = \mathcal{L}((rae)_{cstr}) \cap \mathcal{L}(bl)$.*

Proof. The idea is to consider every part of the bounded expression $w_1.w^*.w_2$ that may be traversed when $(rae)_{cstr}$ passes through a top-level occurrence constraint. For example, $\mathcal{L}((rae)_{cstr}) \cap \mathcal{L}(bl)$ may traverse $bl' = u.w^*.v$ while being in $(rae')_{cstr'}$, with u a suffix of w and v a prefix of w. Each part bl' is again a bounded expression of the form assumed by the lemma. This means for each combination of top-level constraint $(rae')_{cstr'}$ and part bl', we can apply the hypothesis and compute a finite automaton $A_{(rae')_{cstr'},bl'}$ representing the intersection. We now modify the given availability expression $(rae)_{cstr}$ to $(\tilde{rae})_{\tilde{cstr}}$ by replacing every top-level constraint $(rae')_{cstr'}$ with $\bigcup_{bl' \text{ part of } bl} A_{(rae')_{cstr'},bl'}$. This replacement is sound and complete in the sense that

$$\mathcal{L}((rae)_{cstr}) \cap \mathcal{L}(bl) \;=\; \mathcal{L}((\tilde{rae})_{\tilde{cstr}}) \cap \mathcal{L}(bl) \;. \tag{3}$$

Soundness holds by $\mathcal{L}(A_{(rae')_{cstr'},bl'}) = \mathcal{L}((rae')_{cstr'}) \cap \mathcal{L}(bl') \subseteq \mathcal{L}((rae')_{cstr'})$. Completeness is because we consider every part of bl. The finite automaton of interest is constructed from the right-hand side of Equation (3) by going through 1CM, as in Lemma 4.4. Note that $(\tilde{rae})_{\tilde{cstr}}$ indeed has the form $(reg)_{\tilde{cstr}}$ so that the argument from the base case applies.

To see that the construction is polynomial time, note that the number of parts of bl is quadratic. We avoid computing regular expressions for the automata $A_{(rae')_{cstr'},bl'}$ but directly incorporate them into the 1CM construction. ◀

4.2.2 Presburger for the General Case

With the previous automaton construction at hand, we are now prepared to address our actual goal: computing an ∃PA formula that characterizes the intersection of a regular availability language with a bounded language. Our main result is the following proposition:

▶ **Proposition 4.8.** *Consider availability expressions of fixed depth. Given such an expression $(rae)_{cstr}$ and a bounded language $bl = w_0.w_1^* \dots w_m^*.w_{m+1}$, one can compute in polynomial time an $\exists$PA formula $\varphi(x_1, \dots, x_m)$ so that for all $k_1, \dots, k_m \in \mathbb{N}$:*

$$(k_1, \dots, k_m) \models \varphi \qquad \text{if and only if} \qquad w_0.w_1^{k_1} \dots w_m^{k_m}.w_{m+1} \in \mathcal{L}((rae)_{cstr}) \cap \mathcal{L}(bl).$$

The rest of this section is dedicated to the proof of Proposition 4.8. The proof is done by induction on the depth of the availability expression.

Base Case. We first prove the base case when the availability expression is actually a regular expression (see Lemma 4.9).

▶ **Lemma 4.9.** *Given $(reg)_{cstr}$ and $bl = w_0.w_1^* \dots w_m^*.w_{m+1}$, one can compute in polynomial time an $\exists$PA formula $\varphi(x_1, \dots, x_m)$ as required.*

We introduce fresh letters to the bounded language: $bl' := w_0.(w_1.a_1)^* \dots (w_m.a_m)^*.w_{m+1}$. Now an occurrence of a_i signals a full occurrence of w_i. We compute the product with the 1CM for $(reg)_{cstr}$ and apply Proposition 4.2. It yields an $\exists$PA formula which, after existential quantification of the variables for the original letters, is as required by Lemma 4.9.

Induction Step. Next, we show the induction step for the proof of Proposition 4.8. The induction step is established using the following lemma.

▶ **Lemma 4.10.** *Assume Proposition 4.8 holds for availability expressions of depth at most $n \in \mathbb{N}$. Consider $(rae)_{cstr}$ of depth $n+1$ and $bl = w_0.w_1^* \dots w_m^*.w_{m+1}$. One can compute in polynomial time an $\exists$PA formula $\varphi(x_1, \dots, x_m)$ as required.*

This is the proof where we need the two hypotheses: that we can compute a finite automaton representing an intersection with a bounded language of length at most one, and that we can construct an $\exists$PA formula characterizing an intersection with a bounded language of length at least two. We shall assume $m \geq 2$, for otherwise we can apply Lemma 4.9 to the automaton from Proposition 4.3.

Proof. We first define a modification of the given availability expression. It will involve adding new letters that indicate the occurrence of an intersection with a bounded language of length at least two. In a following step, we turn the bounded expression into a finite automaton that takes the fresh letters into account. Then we determine the $\exists$PA formula of interest. The proof concludes with an estimation of the complexity.

Modifying the availability expression Consider every top-level constraint $(rae')_{cstr'}$. For every part bl' of the bounded language that has length at most one, we apply Proposition 4.3. It yields a finite automaton $A_{(rae')_{cstr'}, bl'}$ with language $\mathcal{L}((rae')_{cstr'}) \cap \mathcal{L}(bl')$. Moreover, for every part bl' of length at least two, we introduce a fresh letter $\sharp_{(rae')_{cstr'}, bl'}$. We now replace $(rae')_{cstr'}$ by the regular language

$$\bigcup_{\substack{bl' \text{ part of } bl \\ \text{of length} \leq 1}} A_{(rae')_{cstr'}, bl'} \quad \cup \quad \bigcup_{\substack{bl' \text{ part of } bl \\ \text{of length} > 1}} \sharp_{(rae')_{cstr'}, bl'} \ .$$

The modified availability expression is the result of all these replacements.

Turning the bounded expression into a finite automaton An occurrence of $\sharp_{(rae')_{cstr'}, bl'}$ will represent an occurrence of $\mathcal{L}((rae')_{cstr'}) \cap \mathcal{L}(bl')$. The task of the automaton associated with bl is to enforce that such an intersection is not traversed twice. The construction is

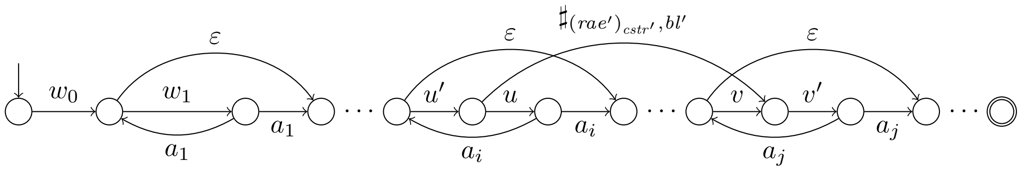


Figure 1 Illustration of A_{bl}.

illustrated in Figure 1. First, we introduce fresh letters counting w_1 to w_m. This gives $w_0.(w_1.a_1)^* \ldots (w_m.a_m)^*.w_{m+1}$. When represented as a finite automaton, we have a state q_w for every prefix w of $w_0.w_1.a_1 \ldots w_{m+1}$. Consider the part $bl' = u.w_i^* \ldots w_j^*.v$ with u a suffix of $w_i = u'.u$ and v a prefix of $w_j = v.v'$. For every top-level availability constraint $(rae')_{cstr'}$ we add a transition labelled by $\sharp_{(rae')_{cstr'},bl'}$ from $q_{w_0 \ldots u'}$ to $q_{w_0 \ldots v}$. Note that such a transition can be taken only once. The result is A_{bl}.

Computing the ∃PA formula The modification of the given availability expression $(rae)_{cstr}$ is of the form $(reg)_{cstr}$. We turn it into a 1CM and add loops to all states to guess the occurrences of the fresh letters a_1 to a_m. Let the result be M. Since A_{bl} is a finite automaton and 1CM are closed under regular intersection, we can compute the product $M \times A_{bl}$. We turn this product into a context-free grammar and apply Proposition 4.2. This gives us an ∃PA formula of the form $\psi(\tilde{p}, \tilde{y}, \tilde{z})$. The vectors of variables are as follows:

- $\tilde{p}$ has one variable p_a for every letter $a \in \Lambda$,
- $\tilde{y}$ has one variable y_i for every word $w_i \in \{w_1, \ldots, w_m\}$,
- $\tilde{z}$ has one variable $z_\sharp$ for each $\sharp = \sharp_{(rae')_{cstr'},bl'}$, $(rae')_{cstr'}$ top-level, and bl' of length ≥ 2.

Let $\sharp$ refer to a pair of top-level occurrence constraint $(rae')_{cstr'}$ and part of length at least two $bl' = u.w_i^* \ldots w_j^*.v$. By the hypothesis, we can compute an ∃PA formula $\psi_\sharp(x_i, \ldots, x_j)$ for $\mathcal{L}((rae')_{cstr'}) \cap \mathcal{L}(bl')$. We modify the formula to $\varphi_\sharp(z_\sharp, x_1^\sharp, \ldots, x_m^\sharp)$:

$$\exists x_i \ldots \exists x_j \ : \ \left(z_\sharp = 0 \ \wedge \ \bigwedge_{k=1}^{m} x_k^\sharp = 0 \right) \ \vee$$
$$\left(z_\sharp = 1 \ \wedge \ \bigwedge_{k<i \vee k>j} x_k^\sharp = 0 \ \wedge \ x_i^\sharp = x_i + 1 \ \wedge \ \bigwedge_{i<k\leq j} x_k^\sharp = x_k \ \wedge \ \psi_\sharp(x_i, \ldots, x_j) \right).$$

If $z_\sharp$ is zero, the transition in A_{bl} labelled by $\sharp$ is not taken. This means the intersection $\mathcal{L}((rae')_{cstr'}) \cap \mathcal{L}(bl')$ does not contribute to the occurrences of w_1 to w_m. Therefore, we require the fresh variables $x_k^\sharp$ to be zero. If $z_\sharp$ is not zero, it has to be one because the $\sharp$-labelled transition can be taken at most once. Since $bl' = u.w_i^* \ldots w_j^*.v$, the intersection $\mathcal{L}((rae')_{cstr'}) \cap \mathcal{L}(bl')$ still does not contribute to the occurrences of w_k with $k < i$ or $k > j$. For w_i we have $x_i + 1$ occurrences. The additional occurrence is for the suffix u. For w_k with $i < k \leq j$, we have precisely x_i occurrences of w_i. Note that we do not have to count the half occurrence of v. Automaton A_{bl} will later see an a_i signalling the occurrence of the composed $v.v' = w_j$.

With this, we can define the overall formula $\varphi(x_1, \ldots, x_m)$:

$$\exists \tilde{p} \ \exists \tilde{y} \ \exists \tilde{z} \ \exists_\sharp \tilde{x}^\sharp \ : \ \bigwedge_{i=1}^{n} x_i = y_i + \sum_\sharp x_i^\sharp \quad \wedge \quad \bigwedge \varphi_\sharp(z_\sharp, x_1^\sharp, \ldots, x_m^\sharp) \quad \wedge \quad \psi(\tilde{p}, \tilde{y}, \tilde{z}).$$

The formula sums up the occurrences of w_i in a new free variable x_i. These occurrences are given by y_i for the outer constraint and for the top-level occurrence constraints that are intersected with a bounded language of length at most one. For the occurrences of w_i in an intersection with a bounded language of length at least two, we sum up the variables $x_i^\sharp$. Note that they are set to zero in case an intersection $\sharp$ is not taken. The remainder adds the formulas $\varphi_\sharp$ for the intersections $\sharp$ of top-level constraints with bounded languages of length at least two, and also adds the formula ψ for the overall intersection. Existential quantifiers hide all the auxiliary variables. We note that $\varphi_\sharp$ as well as ψ are existential Presburger formulas that may contain quantifiers. Since they are surrounded by conjunctions and disjunctions, the scope of these quantifiers can be extruded without harm.

Time complexity of the construction The automata representing the intersections with bounded languages of length at most one can be computed in polynomial time by Proposition 4.3. Similarly, the conversion of bl into A_{bl} can be computed in polynomial time. Indeed, the number of top-level occurrence constraints is bounded by the size of the input. Moreover, there is at most a quadratic number of pairs $w_i^* \dots w_j^*$ and again a quadratic number of prefixes and suffixes. Altogether, there is a polynomial number of symbols that we add. As a result, also the product of 1CM and A_{bl} can be computed in polynomial time, and similar for the Presburger formula $\psi(\tilde{p}, \tilde{y}, \tilde{z})$. It remains to add ∃PA formulas for a polynomial number of intersections $\sharp$. Each such formula can be determined in polynomial time by the hypothesis of the lemma. As a result, we have an overall polynomial time construction. ◀

One can optimize the construction by considering a more general notion of parts $U.w_i^* \dots .w_j^*.V$ where U is the union of all suffixes of w_i and w_{i-1} and V is the union of all prefixes of w_j and w_{j+1}. This does not change the overall complexity.

5 Containment

We study the problem of whether an availability language is contained in a regular language and vice versa. For the former problem, we show that availability languages are closed under regular intersection.

▶ **Theorem 5.1.** *Given rae and reg, we can construct rae′ with $\mathcal{L}(rae') = \mathcal{L}(rae) \cap \mathcal{L}(reg)$. With Theorem 3.6, $\mathcal{L}(rae) \subseteq \mathcal{L}(reg)$ is decidable.*

For the proof, we represent the regular language by a finite automaton. Then we compute, for each pair of entry and exit state, the intersection of the corresponding regular language with the top-level occurrence constraints. This gives an inductive construction.

For the reverse inclusion, we show the undecidability by a reduction from the halting problem for two-counter automata (2CM) [13]. 2CM are defined like the 1CM in Section 3 but use two counters. We can assume them to only add 1 or -1, and will use inc and dec, instead. So the overall alphabet is $\Lambda := \{\mathsf{inc}(i), \mathsf{dec}(i), \mathsf{zero}(i) \mid i = 1, 2\}$.

The idea of the reduction is to understand a 2CM as a finite automaton. The automaton only reflects the control-flow but does not take into account the semantics of counters. This means the language is regular, let it be $\mathcal{L}(reg)$. We define an availability language $\mathcal{L}(rae)$ that contains all words over Λ violating the semantics of two-counter automata. Together,

$$\mathcal{L}(reg) \subseteq \mathcal{L}(rae) \qquad \text{iff} \qquad \mathcal{L}(reg) \cap \overline{\mathcal{L}(rae)} = \emptyset.$$

Language $\mathcal{L}(reg) \cap \overline{\mathcal{L}(rae)}$ restricts the regular control-flow language to words respecting the semantics of counters. This language is empty if and only if the 2CM does not halt.

▶ **Theorem 5.2.** *$\mathcal{L}(reg) \subseteq \mathcal{L}(rae)$ is undecidable, even for rae of depth 1.*

Proof. It remains to define rae. The expression is a choice $rae := rae_1 + rae_2$ where rae_1 reflects the bad behavior on counter 1, and similar for rae_2. There are two choices for bad behavior: we decrement a counter below zero (see $rae_{1,1}$) or a test for zero fails (see $rae_{1,2}$):

$$\begin{aligned} rae_1 &:= (\Lambda^*.\mathsf{zero}(1) + \varepsilon).(rae_{1,1} + rae_{1,2}).\Lambda^* \\ rae_{1,1} &:= ((\ \mathsf{inc}(1) + \mathsf{dec}(1) + \mathsf{inc}(2) + \mathsf{dec}(2) + \mathsf{zero}(2)\ )^*.\checkmark)_{\#\mathsf{dec}(1) > \#\mathsf{inc}(1)} \\ rae_{1,2} &:= ((\ \mathsf{inc}(1) + \mathsf{dec}(1) + \mathsf{inc}(2) + \mathsf{dec}(2) + \mathsf{zero}(2)\ )^*.\checkmark.\mathsf{zero}(1))_{\#\mathsf{inc}(1) > \#\mathsf{dec}(1)}. \end{aligned}$$

◀

6 Concluding Remarks and Future Work

Availability languages extend regular languages by occurrence constraints on the letters [11]. The extension increases expressiveness and leads to a class of languages incomparable with the context-free ones. In this paper, we contributed positive results to the algorithmic analysis of availability languages. Our first result is the decidability of the emptiness problem that was left open in [11]. Our solution is inductive and combines an explicit one-counter automata construction with Parikh's theorem. Our second result is NP-completeness of the intersection problem modulo bounded languages. The idea is to reduce to satisfiability of existential Presburger arithmetic. The reduction needs arguments about the boundedness behavior of the one-counter automata representing availability languages. Finally, we study regular containment. We obtain a positive result for safety verification $\mathcal{L}(rae) \subseteq \mathcal{L}(reg)$ and a negative result for the reverse inclusion $\mathcal{L}(reg) \subseteq \mathcal{L}(rae)$.

For future work, we see practical as well as theoretical avenues. On the practical side, we plan to study the use of availability languages in model checking. Although we have shown safety verification to be decidable, the question remains how to check the inclusion efficiently in practice. On the theoretical side, it should be beneficial to compare availability languages with other models. It would be attractive to have a uniform understanding of Parikh automata, Presburger languages, and availability languages. Extensions of monadic second-order logic designed to capture availability requirements would also be interesting. Finally, there is no omega-theory of availability.

References

1 M. Cadilhac, A. Finkel, and P. McKenzie. On the expressiveness of Parikh automata and related models. *arXiv:1101.1547 [cs]*, 2011.

2 M. Cadilhac, A. Finkel, and P. McKenzie. Affine Parikh automata. *RAI*, 46(4):511–545, 2012.

3 M. Cadilhac, A. Finkel, and P. McKenzie. Bounded Parikh automata. *International Journal of Foundations of Computer Science*, 23(8):1691–1709, 2012.

4 E. de Souza e Silva and H. R. Gail. Calculating availability and performability measures of repairable computer systems using randomization. *JACM*, 36(1):171–193, 1989.

5 M. Droste, W. Kuich, and H. Vogler, editors. *Handbook of Weighted Automata*. EATCS Monographs. Springer, 2009.

6 J. Esparza and P. Ganty. Complexity of pattern-based verification for multithreaded programs. In *POPL*, pages 499–510. ACM, 2011.

7 J. Esparza, P. Ganty, S. Kiefer, and M. Luttenberger. Parikh's theorem: A simple and direct automaton construction. *IPL*, 111(12):614–619, 2011.

8 P. Ganty, R. Majumdar, and B. Monmege. Bounded underapproximations. *FMSD*, 40(2):206–231, 2012.

9 S. Ginsburg and E. H. Spanier. Semigroups, Presburger formulas, and languages. *Pacific Journal of Mathematic*, 16(2):285–296, 1966.

10 M. Hague and A. W. Lin. Synchronisation- and reversal-bounded analysis of multithreaded programs with counters. In *CAV*, volume 7358 of *LNCS*, pages 260–276. Springer, 2012.

11 J. Hoenicke, R. Meyer, and E.-R. Olderog. Kleene, Rabin, and Scott are available. In *CONCUR*, number 6269 in LNCS, pages 462–477. Springer, 2010.

12 F. Klaedtke and H. Rueß. Monadic second-order logics with cardinalities. In *ICALP*, volume 2719 of *LNCS*, pages 681–696. Springer, 2003.

13 M. L. Minsky. *Computation: Finite and Infinite Machines.* Prentice-Hall, 1967.

14 R. J. Parikh. On context-free languages. *JACM*, 13(4):570–581, 1966.

15 S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *TACAS*, volume 3440 of *LNCS*, pages 93–107. Springer, 2005.

16 G. Rubino and B. Sericola. Interval availability distribution computation. In *Fault-Tolerant Computing*, pages 48–55, 1993.

17 B. Scarpellini. Complexity of subcases of Presburger arithmetic. *Transactions of the AMS*, 284(1):203–218, 1984.

18 H. Seidl, T. Schwentick, and A. Muscholl. Numerical document queries. In *PODS*, pages 155–166. ACM, 2003.

19 K. N. Verma, H. Seidl, and T. Schwentick. On the complexity of equational horn clauses. In *CADE*, volume 3632 of *LNCS*, pages 337–352. Springer, 2005.