Kleene, Rabin, and Scott are available

Jochen Hoenicke[1], Roland Meyer[2], and Ernst-Rüdiger Olderog[3] ⋆

[1] University of Freiburg
email: `hoenicke@informatik.uni-freiburg.de`

[2] LIAFA, Paris Diderot University & CNRS
email: `roland.meyer@liafa.jussieu.fr`

[3] University of Oldenburg
email: `olderog@informatik.uni-oldenburg.de`

Abstract. We are concerned with the availability of systems, defined as the ratio between time of correct functioning and uptime. We propose to model guaranteed availability in terms of *regular availability expressions* (rae) and *availability automata*. We prove that the intersection problem of rae is undecidable. We establish a Kleene theorem that shows the equivalence of the formalisms and states precise correspondence of flat rae and simple availability automata. For these automata, we provide an extension of the powerset construction for finite automata due to Rabin and Scott. As a consequence, we can state a complementation algorithm. This enables us to solve the synthesis problem and to reduce model checking of availability properties to reachability.

1 Introduction

Traditional approaches to system verification rely on idealistic assumptions, e.g., that each component will work perfectly all the time. However, in many applications such assumptions are unrealistic. Think of a sensor network where some of the sensors fail and recover. Is then the whole information accumulated by the network invalid?

Our paper is motivated by the desire to establish correctness properties of unreliable reactive systems where components may fail for some time, or phrased positively, are available only for a certain amount of time during an observation interval. This property is known as *(interval) availability*. It is often studied in the context of stochastic systems where one calculates the *probability* that a component or system has a certain interval availability but it may also be studied in the context of timed systems [dSeSG89,RS93,Tri01]. For continuous time models, availability can be formalised using integrals. Letting $sys(t)$ represent proper system functionality ($\{0,1\}$-valued) at time t, the expression

$$\frac{1}{n} \cdot \int_0^n sys(t)\ dt \geq k$$

⋆ This work was partly supported by the German Research Council (DFG), grant SFB/TR 14 AVACS. The second author was supported by the French ANR projects Averiss and Veridyc. In Proc. of CONCUR, volume 6269 of LNCS. Springer, 2010.

states that the *ratio* of accumulated time the system is functioning as desired to total uptime n is at least $k \in [0, 1]$. Thus, during the observation interval $[0, n]$ the system is available for fraction k of the time.

We discovered that availability can be studied already in the simpler setting of discrete time, in terms of formal languages and automata-theoretic models. This is what this paper is about. We express availability as the ratio of letters from a set A in a word $w = a_1 \cdots a_n$ to the length n of the word. Formula

$$\frac{1}{n} \cdot \sum_{t=1}^{n} \chi_A(a_t) \geq k$$

states that in word w the letters of the set A are available for at least fraction k of the time. When modelling, the set A may contain desired system states or receipt actions of messages. Our contributions are as follows:

1. We introduce (in Section 2) the class of *regular availability expressions* (*rae* for short). We prove that the problem whether the intersection of finitely many raes denotes the empty language is *undecidable*. The proof is by reduction of the termination problem of Minsky machines.
2. We introduce (in Section 3) *availability automata* and establish a *Kleene theorem*. It states that the class of languages denoted by flat raes coincides with the class accepted by *simple* availability automata. We derive a correspondence between intersections of raes and availability automata.
3. We extend (in Section 4) the *powerset construction* for finite automata [RS59]. For a nondeterministic *simple* availability automaton it computes an equivalent deterministic version. As a consequence, we derive a *complementation* algorithm for simple automata that solves the *synthesis* problem and reduces *model checking* of availability properties to reachability.

2 Regular availability expression

We define availability for *finite words* $w \in \Sigma^*$ over an alphabet Σ. We denote by $|w|$ the length of the word. For $A \subseteq \Sigma$ we denote by $\pi_A(w)$ the projection of w to the alphabet A, i. e., the word that is derived from w by removing all letters that are not in A. This notation allows for a concise definition of availability that avoids division by zero in case $w = \varepsilon$:

$$\frac{1}{n} \cdot \sum_{i=1}^{n} \chi_A(a_i) \geq k \quad \text{iff} \quad |\pi_A(w)| \geq k|w| \quad \text{with } w = a_1 \ldots a_n.$$

Definition 1 (Syntax of raes). *The set of* regular availability expressions (rae) over an alphabet Σ *is inductively defined as follows:*

$$rae ::= a \mid rae + rae \mid rae.rae \mid rae^* \mid \checkmark \mid rae_{A \gtrsim k}$$

with $a \in \Sigma$, $A \subseteq \Sigma$, $\gtrsim \in \{\geq, >\}$, and $k \in [0, 1]$.

The symbol ✓ marks the positions at which the required availability is checked. At these positions, the expression $rae_{A\geq k}$ ensures that the letters in A are available for at least fraction k. A highly available network may be specified by the expression $((up + down)^*.✓)_{\{up\}>0.99}$. We abbreviate $rae_{(\Sigma\setminus A)\geq 1-k}$ by $rae_{A\leq k}$.

An rae is *flat* if the operator $rae_{A\gtrsim k}$ does not appear nested. Flat raes are the intuitive model one expects. The system behaviour is captured by a regular expression. For analysis purposes, availability constraints are added to restrict the traces to the average behaviour. As the system typically does not react to the occurrence number of events, nesting of availability operators is hardly needed.

Raes are given a semantics in terms of languages $\mathcal{L}(rae) \subseteq (\Sigma \cup \{✓\})^*$:

$$\begin{aligned} \mathcal{L}(a) &:= \{a\} & \mathcal{L}(rae_1 + rae_2) &:= \mathcal{L}(rae_1) \cup \mathcal{L}(rae_2) \\ \mathcal{L}(rae^*) &:= \mathcal{L}(rae)^* & \mathcal{L}(rae_1.rae_2) &:= \mathcal{L}(rae_1).\mathcal{L}(rae_2) \\ \mathcal{L}(✓) &:= \{✓\} & \mathcal{L}\left(rae_{A\gtrsim k}\right) &:= \mathcal{L}(rae)_{A\gtrsim k}. \end{aligned}$$

Operator $\mathcal{L}_{A\gtrsim k}$ collects all words in language $\mathcal{L}$, where each prefix ending in ✓ satisfies the availability $A \gtrsim k$ as discussed above (not counting ✓-symbols). The operator also removes these symbols. Formally,

$$\mathcal{L}_{A\gtrsim k} := \{\pi_\Sigma(w) \mid w \in \mathcal{L} \text{ and } |\pi_A(w_1)| \gtrsim k|\pi_\Sigma(w_1)| \text{ for all } w_1.✓.w_2 = w\}.$$

Raes and classical regular expressions differ in their properties. Raes have an undecidable language *intersection problem*

$$\mathcal{L}(rae_1) \cap \mathcal{L}(rae_2) = \emptyset. \qquad (\textbf{Intersect})$$

Theorem 1. Intersect *is undecidable.*

Proof. The proof of Theorem 1 is by reduction of the termination problem for Minsky machines to the intersection problem for raes. Since Minsky machines are Turing complete [Min67, Theorem 14.1-1], termination is undecidable.

A Minsky machine $M = (c_1, c_2, inst)$ has two *counters* c_1 and c_2 that store arbitrarily large natural numbers and a *finite set inst of labelled instructions* $l : op$. There are two kinds of operations op. The first, denoted by $inc(c, l')$, increments counter $c \in \{c_1, c_2\}$ by one and then jumps to the instruction labelled by l'. The second command, denoted by $dect(c, l', l'')$, is called a *decrement and test*. It checks counter c for being zero and, in this case, jumps to instruction l'. If the value of c is positive, the counter is decremented and the machine jumps to l''. We use $Locs(inst) := \{l \mid l : op \in inst\}$ to refer to the *set of control locations* in M. It contains an *initial instruction* $l_I \in Locs(inst)$ that starts the computation of the Minsky machine. A *final label* $l_F \notin Locs(inst)$ may appear only as the destination of an instruction and terminates the computation.

Given a Minsky machine $M = (c_1, c_2, inst)$ we define two raes so that the intersection of their languages is non-empty iff the computation of M terminates.

Construction. We start by splitting each instruction in $inst$ into two parts, one part for each counter. This yields two new sets of instructions $inst_1$ and $inst_2$. The parts added to $inst_1$ only affect counter c_1 and jump to the second part of the instruction in inst_2. The parts added to inst_2 affect counter c_2 and jump to the first part of the next instruction. Since every instruction in inst only changes one counter, we need a new operation $\mathit{goto}(l)$ to jump when no change is made:

$$\begin{aligned} l_i : \mathit{inc}(c_1, l_j) \in \mathit{inst} &\quad\rightsquigarrow\quad l_i : \mathit{inc}(l_i') \in \mathit{inst}_1 \wedge l_i' : \mathit{goto}(l_j) \in \mathit{inst}_2 \\ l_i : \mathit{dect}(c_1, l_j, l_k) \in \mathit{inst} &\quad\rightsquigarrow\quad l_i : \mathit{dect}(l_i', l_i'') \in \mathit{inst}_1 \wedge l_i' : \mathit{goto}(l_j) \in \mathit{inst}_2 \\ &\qquad\qquad \wedge\, l_i'' : \mathit{goto}(l_k) \in \mathit{inst}_2. \end{aligned}$$

The translation for the second counter is similar. By this transformation, two adjacent instructions $l : op$, $l' : op$ always change first the counter c_1 and then c_2. We encode the computation steps of the machines as $l.c^v.a.r^{v'}.\, l'.c^u.b.r^{u'}$. The sequence of $v \in \mathbb{N}$ letters c encodes the valuation of the counter c_1 before the first instruction. The following symbol a is either i, d, or ε depending on whether operation $l : op$ increments the first counter, decrements it, or does not act on it. An ε is also used if op tests the first counter for being zero. The result of the operation is stored as a sequence of result letters $r^{v'}$. We ensure that $v' = v + 1$ if a is an increment (similarly for decrement and goto) by an availability expression *av1*, indicated by a brace in Formula 1 below. Then the operation on c_2 starts, indicated by its label l'. In the encoding of the second counter, b represents the operation to be performed.

The crucial issue is to transfer the result $r^{v'}$ of an operation to the next instruction that is executed. Again, we use an availability expression *av2*, which now connects the encodings of two subsequent computation steps. To sum up, a terminating computation of the machine is encoded by the word

$$\big[l_I.(\overbrace{c^{v_0}.a_0.r^{v_1}}^{\mathit{av1}}).l_I'.(c^{u_0}.b_0.r^{u_1})\big].\ \big[l_1.(c^{v_1}.a_1.r^{v_2}).l_1'.(c^{u_1}.b_1.r^{u_2})\big]\ldots l_F. \qquad (1)$$

$$\underbrace{r^{v_1}) \ldots (c^{v_1}}_{\mathit{av2}}$$

To encode the effect of a *goto*, we demand equality between the number of c and r symbols by stating that the availability of c is precisely $^1/_2$. This is achieved by the regular availability expression

$$\mathit{GOTO} := \big((c^*.r^*.\checkmark)_{\{c\}\geq ^1/_2}.\checkmark\big)_{\{c\}\leq ^1/_2}.$$

To avoid clutter we abbreviate this by $(c^*.r^*.\checkmark)_{\{c\}=^1/_2}$. Note that this trick is only valid if there is exactly one $\checkmark$-symbol at the end of the expression. The following availability expressions implement increment and decrement operations:

$$\mathit{INC} := (c^*.i.r^*.\checkmark)_{\{c,i\}=^1/_2} \qquad \mathit{DEC} := (c^*.d.r^*.\checkmark)_{\{c,i\}=^1/_2}.$$

In the increment expression, the symbol i is counted like c and has to match an additional r symbol, which ensures that the number of r symbols is by one larger than the number of c symbols. Likewise in the decrement expression, the symbol d has to match an additional c symbol, so the result value encoded by the number of r symbols is by one smaller than the number of c symbols. Thus all expressions encoding operations have the shape

$$(c^*.\mathit{CMD}.r^*.\checkmark)_{\{c,i\}=^1/_2} \quad \text{with } \mathit{CMD} := i + d + \varepsilon.$$

With these definition, we define an availability expression $rae(l : op)$ for every instruction $l : op$. It ensures correct computation and control flow. Note that tests for zero need to be implemented by ε instead of $GOTO$:

$$rae(l : inc(l')) := l.INC.l' \qquad rae(l : dect(l', l'')) := l.l' + l.DEC.l''$$
$$rae(l : goto(l')) := l.GOTO.l'.$$

Combining all commands on a counter, we define

$$OP_i := \Sigma_{l:op \in inst_i} rae(l : op) \quad \text{for } i = 1, 2.$$

We rely on two availability expressions that define a word corresponding to a terminating execution of the Minsky machine. The first expression imitates instructions on the first counter and copies contents of the second. It also ensures that the execution ends at the final label and that the second counter is initialised to zero. The second expression executes instructions on the second counter and copies the contents of the first. It also starts the execution at the initial label and initialises the first counter with zero:

$$rae_{cmp1}(M) := OP_1.\varepsilon.CMD.\left[(r^*.OP_1.c^*.\checkmark)_{\{c,i\}\cup Locs(inst_1)={}^1\!/_2}.CMD\right]^*.r^*.l_F$$
$$rae_{cmp2}(M) := l_I.\varepsilon.CMD.\left[(r^*.OP_2.c^*.\checkmark)_{\{c,i\}\cup Locs(inst_1)={}^1\!/_2}.CMD\right]^*.r^*.OP2.$$

To copy the result value r^* from one computation step to the counter value c^* in the next, we again employ availability expressions, i.e., we implement *av2* in Formula 1. One difficulty is that this copy operation is interrupted by an OP. However, the definitions of INC, DEC, and $GOTO$ guarantee an availability of ${}^1\!/_2$ for the letters c, i between the labels. There are two labels in OP; since those from $Locs(inst_1)$ are in the availability set, the full OP-command guarantees an availability of exactly ${}^1\!/_2$. As a result, we obtain equality between r^* and c^*.

One can show that M terminates if and only if the intersection

$$\mathcal{L}\left(rae_{cmp1}(M)\right) \cap \mathcal{L}\left(rae_{cmp2}(M)\right) \neq \emptyset.$$

is non-empty. The details can be found in the appendix. □

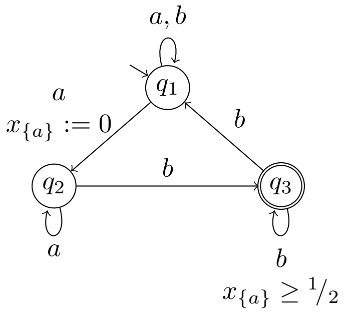
Remark 1. The intersection problem remains undecidable for eight flat raes. Each of the raes in the previous construction can be expressed by four flat versions (note that an equality requires two constraints).

Theorem 1 shows that raes correspond to an automaton model that is strictly more expressive than finite automata. We investigate it in the following section.

3 Availability automata

We define *availability automata* as transition labelled finite automata enriched by *(availability) counters* that determine the presence of certain letters within a run. Each counter represents a *check* operation of an availability $A \gtrsim k$ with $A \subseteq \Sigma$ and $k \in [0, 1]$, and transitions in the availability automaton are guarded by these constraints. Additionally, each transition carries a *reset* operation $Y := 0$ that denotes the counters that are reset so as to restart the measurement afterwards.

Definition 2 (Availability automata). *Let Σ be an alphabet. An* availability automaton over *Σ is a tuple $\mathcal{A} = (Q, Q_I, Q_F, X, \rightarrow, c)$ with* states *Q,* initial states *$Q_I \subseteq Q$, and* final states *$Q_F \subseteq Q$. The* availability counters *are given by X. The* counter labelling function *$c : X \rightarrow (\mathbb{P}(\Sigma)\times\{\geq, >\}\times[0,1])$ assigns to each counter $x \in X$ a constraint $A \gtrsim k$.* Transitions *$\rightarrow \subseteq Q\times\Sigma\times\mathbb{P}(X)\times\mathbb{P}(X)\times Q$ are labelled by Σ, check the constraints of their counters, and reset some counters.*



Consider the availability automaton to the right. We employ the following notation. States are drawn as nodes. Initial states have an incoming arc, final states carry a double circle. The first part $A \subseteq \Sigma$ of the counter labelling $c(x) = (A \gtrsim k)$ is given as index x_A of the counter. If two counters are indexed by the same set A, we use different variables x_A and y_A. A transition $(q_1, a, C, Y, q_2) \in \rightarrow$ is drawn as directed arc from q_1 to q_2 labelled by a. A check operation $C = \{x_A\}$ is written as $x_A \gtrsim k$, revealing the remaining part of the counter labelling. A reset set $Y = \{x_A\}$ is denoted by $x_A := 0$.

To give an operational semantics, we exploit the following equivalence that highlights the contribution of a single action to an availability expression:

$$|\pi_A(w)| \gtrsim k|w| \quad \text{iff} \quad |\pi_A(w)| - k|w| \gtrsim 0.$$

To check the availability $A \gtrsim k$, we compare the value of $|\pi_A(w)| - k|w|$ against zero. More importantly, this value can be computed incrementally by adding 1 for each occurrence of a symbol of A and subtracting k for every symbol. The observation suggests the use of *counter valuations* $\gamma : X \rightarrow \mathbb{R}$. They assign to each counter the value $|\pi_A(w)| - k|w|$ of the word w, which has been read since the last reset. For $a \in \Sigma$, we denote by $\chi(a) : X \rightarrow \mathbb{R}$ the counter valuation that assigns $\chi_A(a) - k$ to counter x checking $c(x) = A \gtrsim k$. As usual, χ_A is the characteristic function. This enables us to describe the *update* when processing the character a by $\gamma' = \gamma + \chi(a)$. The *reset* $\gamma' = \gamma[Y := 0]$ yields $\gamma'(x) = 0$ if $x \in Y$ and $\gamma'(x) = \gamma(x)$ otherwise.

The semantics of an availability automaton $\mathcal{A} = (Q, Q_I, Q_F, X, \rightarrow, l)$ is given in terms of runs in the set $\mathcal{R}(\mathcal{A})$. A *run* r is a sequence

$$r = q_0.\gamma_0.a_1.q_1.\gamma_1.a_2.q_2.\gamma_2 \ldots a_n.q_n.\gamma_n \in \mathcal{R}(\mathcal{A})$$

subject to the following constraints. Initially, all counters are zero, $\gamma_0(x) = 0$ for all $x \in X$, and the run starts in an initial state, $q_0 \in Q_I$. For every step $q_{i-1}.\gamma_{i-1}.a_i.q_i.\gamma_i$ there is a transition $(q_{i-1}, a_i, C, Y, q_i) \in \rightarrow$ such that $\gamma_i = (\gamma_{i-1} + \chi(a_i))[Y := 0]$ and for each constraint $x \in C$, $\gamma_{i-1}(x) + \chi(a_i)(x) \gtrsim 0$. By the above transformation, this guarantees the desired availability.

Runs contain internal information about states and counter valuations. We abstract them away to obtain the usual notion of the language of an automaton.

Definition 3 (Language). *The* language of $\mathcal{A} = (Q, Q_I, Q_F, X, \rightarrow, l)$ *is the projection of all runs that end in a final location to their labels*

$$\mathcal{L}(\mathcal{A}) := \{a_1 \dots a_n \mid q_0.\gamma_0.a_1.q_1.\gamma_1 \dots a_n.q_n.\gamma_n \in \mathcal{R}(\mathcal{A}) \text{ with } q_n \in Q_F\}.$$

Like for finite automata, ε-transitions do not contribute to the expressiveness of availability automata but are convenient when encoding raes.

Lemma 1. *For every $\Sigma \cup \{\varepsilon\}$-labelled availability automaton $\mathcal{A}$, there is a Σ-labelled availability automaton $\mathcal{A}'$ with $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}')$.*

As we will show later, the language emptiness problem is undecidable for general availability automata. However, for automata with only one active counter in each state emptiness becomes decidable. We call them *simple.*

Definition 4 (Simple availability automata). *An availability automaton $\mathcal{A} = (Q, Q_I, Q_F, X, \rightarrow, c)$ is called* simple*, if there is at most one active counter in each state. Formally, there is a mapping $\mu : Q \rightarrow X$ such that each transition $(q, a, C, Y, q') \in \rightarrow$ is only constrained by the counter of q, $C \subseteq \{\mu(q)\}$, and resets the counter of q' if it differs from the counter of q, $\{\mu_{\mathcal{A}}(q')\} \subseteq \{\mu_{\mathcal{A}}(q)\} \cup Y$.*

Simple automata capture precisely the languages of flat raes and are thus of particular interest. The following two sections are devoted to the proof of this statement. In general, availability automata are equivalent to intersections of raes — modulo renaming.

3.1 From expressions to automata

To encode raes into availability automata, we define operators on the automata that mimic the operations on raes. *Choice* $\mathcal{A}_1 + \mathcal{A}_2$, *sequential composition* $\mathcal{A}_1.\mathcal{A}_2$, and *iteration* $\mathcal{A}^*$ correspond to the constructions for finite automata. Choice is defined by union, sequential composition introduces ε-transitions from the final states of the first to the initial states of the second automaton, and iteration introduces ε-transitions back to a fresh initial and final state. Different from the classical definitions, counters are reset when a new automaton is entered in $\mathcal{A}_1.\mathcal{A}_2$ and $\mathcal{A}^*$. All these operations produce simple availability automata if the input automata are simple.

An *availability constraint* $\mathcal{A}_{A \gtrsim k}$ is reflected by a relabelling of the automaton. The idea is to let every former $\checkmark$-labelled transition check the constraint $A \gtrsim k$. To this end, a new counter x_A is added to the automaton and every $\checkmark$-transition is relabelled to ε and augmented by the counter's constraint $x_A \gtrsim k$. For flat raes, the operation results in a simple automaton, since the input is a finite automaton without counters.

To reflect the intersection of languages — an essential ingredient in automata-theoretic verification procedures — we define a *synchronous product* $\mathcal{A}_1 \parallel \mathcal{A}_2$. It multiplies the states $Q^1 \times Q^2$, takes the pairs of initial states as initial $Q^1_I \times Q^2_I$, and likewise as final states $Q^1_F \times Q^2_F$. Transitions synchronise on the label and combine the guards. Note that simple availability automata are not closed under synchronous product. The operators reflect their semantics counterparts.

Proposition 1 (Semantic correspondence).

$$\begin{aligned} \mathcal{L}(\mathcal{A}_1.\mathcal{A}_2) &= \mathcal{L}(\mathcal{A}_1).\mathcal{L}(\mathcal{A}_2) & \mathcal{L}(\mathcal{A}^*) &= \mathcal{L}(\mathcal{A})^* \\ \mathcal{L}(\mathcal{A}_1 + \mathcal{A}_2) &= \mathcal{L}(\mathcal{A}_1) \cup \mathcal{L}(\mathcal{A}_2) & \mathcal{L}\left(\mathcal{A}_{A \gtrsim m}\right) &= \mathcal{L}(\mathcal{A})_{A \gtrsim m} \\ \mathcal{L}(\mathcal{A}_1 \parallel \mathcal{A}_2) &= \mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2). \end{aligned}$$

Proposition 1 paves the way for a compositional definition of the automaton representation $\mathcal{A}[\![rae]\!]$ of an rae. A single action a is translated to the automaton that has an a-transition from an initial to a final state. The remaining operators are replaced homomorphically by their automata-theoretic counterparts.

Proposition 2 (Kleene's first half). $\mathcal{L}(rae) = \mathcal{L}(\mathcal{A}[\![rae]\!])$. *Moreover, if rae is flat, then* $\mathcal{A}[\![rae]\!]$ *is simple.*

3.2 From automata to expressions

We start with a simple availability automaton $\mathcal{A} = (Q, Q_I, Q_F, X, \rightarrow, c)$ and its counter mapping $\mu : Q \rightarrow X$. Without changing the accepted language, we can strip resets of counters that are not active in the successor state. We compute a corresponding flat rae in two steps. First, we compute $rae_{q,q'}$ that describes all words from q to q' that obey the counter's availability constraint. More precisely, the rae models all words that are accepted by $\mathcal{A}$ when starting in q with a zero counter and reaching q' with a reset only on the last edge. Therefore, the concatenation $rae_{q,q'}.rae_{q',q''}$ again corresponds to some partial run of $\mathcal{A}$. Likewise, we determine availability expressions rae_q for the language from q (with a zero counter and without passing reset edges) to a final state.

In a second step, we construct a finite state automaton which has these raes as transition labels. When using Kleene's classical result to compute the regular expression corresponding to the automaton, we obtain a flat rae. It has precisely the language of $\mathcal{A}$. Since we recorded the measurements as raes, there is no need to add further availability constraints.

Phase 1 For every pair of states $q, q' \in \mathcal{A}$, we construct a finite automaton[4] $A_{q,q'}$. We take the graph of $\mathcal{A}$, make q the initial state, and add a fresh final state. Resetting transitions that previously ended in q' are redirected to the final state. Thus, the measurement between q and q' ends with a reset. The remaining resetting transitions are removed. Since $\mathcal{A}$ is simple, this makes all states with a different active counter unreachable, so they can be removed together with their outgoing edges. The resulting automaton has the single counter $x := \mu(q)$. To reflect measurements of $c(x)$ on an a-labelled transition, we split the edge. The first new transition is labelled by a, the second by $\checkmark$. Let $\rightarrow_x$ denote the outgoing transitions from a state where x is active, transitions $\rightarrow_{res} \subseteq \rightarrow_x$ reset the counter, and $\rightarrow_{c(x)} \subseteq \rightarrow_x$ have $c(x)$ as guard. By $\rightarrow_{std}$ we refer to unguarded

[4] A finite automaton is a tuple $A = (Q, Q_I, Q_F, \rightarrow)$ with the typical interpretation as *states*, *initial states*, *final states*, and *transition relation* $\rightarrow \subseteq Q \times \Sigma \cup \{\varepsilon, \checkmark\} \times Q$.

and reset-free transitions in $\rightarrow_x \setminus (\rightarrow_{res} \cup \rightarrow_{c(x)})$. They can be understood as transitions of a finite automaton:

$$A_{q,q'} := (\mu^{-1}(x) \cup \{q_e \mid e \in \rightarrow_{c(x)}\} \cup \{q_\nu\}, \{q\}, \{q_\nu\}, \rightarrow_{std} \cup \rightarrow_{split} \cup \rightarrow_{redir}),$$

with $\rightarrow_{split} := \{(p, a, q_e), (q_e, \checkmark, p') \mid e = (p, a, \{x\}, \emptyset, p') \in \rightarrow_{c(x)}\}$ and $\rightarrow_{redir} := \{(p, a, q_\nu) \mid (p, a, \emptyset, \{\mu(q')\}, q') \in \rightarrow_{res}\}$. If the resetting transition to q' is guarded, also the redirected edge is split up.

The finite automaton A_q is constructed similarly. It has q as initial state, uses the final states of the original automaton, and removes all resetting edges. Guarded transitions are again split up and decorated by $\checkmark$:

$$A_q := (\mu^{-1}(x) \cup \{q_e \mid e \in \rightarrow_{c(x)}\}, \{q\}, Q_F, \rightarrow_{std} \cup \rightarrow_{split})$$

We compute an ordinary regular expression $re_{q,q'}$ that accepts the language of $A_{q,q'}$. Adding the constraint $c(x)$ yields $rae_{q,q'} := [re_{q,q'}]_{c(x)}$ that reflects the measurement between q and q'. We also construct re_q and define $rae_q := [re_q]_{c(x)}$.

Phase 2 From $\mathcal{A}$ we construct a finite automaton A that describes its language. Again, we preserve the states, keep the initial states, and add a fresh final state. The main idea is to summarise all paths between two states q and q' by a single transition that is labelled by $rae_{q,q'}$. The expression takes care of the required availability constraint. Similarly, from every state q we have a transition to the new final state that is labelled by rae_q:

$$A := (Q \cup \{q_\nu\}, Q_I, \{q_\nu\}, \{(q, rae_{q,q'}, q') \mid q, q' \in Q\} \cup \{(q, rae_q, q_\nu) \mid q \in Q\}).$$

Let $rae[\![\mathcal{A}]\!]$ denote the regular expression for A. Due to the flat raes as labels, the expression itself is again flat. It correctly represents the automaton's language.

Example 1. Applying the algorithm to the automaton from the beginning of Section 3 yields $((a+b)^*a)_{\{a\}\geq 1/2}((a^*b(b\checkmark)^*b(a+b)^*a)_{\{a\}\geq 1/2})^*(a^*b(b\checkmark)^*)_{\{a\}\geq 1/2}$.

Proposition 3 (Kleene's second half). $\mathcal{L}(\mathcal{A}) = \mathcal{L}(rae[\![\mathcal{A}]\!])$.

Proposition 2 and Proposition 3 establish our second main result. Flat regular availability expressions and simple availability automata are equally expressive.

Theorem 2 (Kleene theorem for availability). *A language is recognised by a flat rae if and only if it is accepted by a simple availability automaton.*

An availability automaton that is not simple can be decomposed into simple automata. Take a *free version* $\mathcal{A}_f$ of the automaton $\mathcal{A}$ where transitions e have unique labels, say a_e. The original language can be obtained by removing the indices with the homomorphism $h(a_e) = a$. Thus, $h(\mathcal{L}(\mathcal{A}_f)) = \mathcal{L}(\mathcal{A})$ holds. The free automaton $\mathcal{A}_f$ is decomposed into several simple availability automata $\mathcal{A}_f^x$ with x as single counter. This decomposition projects away the constraints on the other counters and leaves states and transitions unchanged.

Lemma 2. *Consider $\mathcal{A}_f$ with counters X. Then $\mathcal{L}(A_f) = \bigcap_{x \in X} \mathcal{L}\left(A_f^x\right)$.*

With the previous result we derive the following correspondence.

Corollary 1. *Up to a homomorphism, a language is recognised by an intersection of raes if and only if it is accepted by an availability automaton.*

4 A powerset construction

The well-known powerset construction of Rabin and Scott that constructs deterministic finite automata from non-deterministic ones can be extended to simple availability automata. Like for finite automata, the key idea is to record in the state of the deterministic automaton the possible states the non-deterministic automaton can be in, e.g. $\{p, q\}$ would be a state of the deterministic automaton. For availability automata, also the possible values of the active counter in different runs need to be represented. Therefore, the following observation is crucial to our construction. For simple automata, it is sufficient to record the highest availability for this counter. Thus, we record one value for each state of the original automaton, which can be achieved by $|Q|$ counters. For the state $\{p, q\}$, this yields counters x_p and x_q.

When the non-deterministic automaton changes its state without resetting the active counter, say from p to p', we need to set the counter $x_{p'}$ of the new state to the counter of the old state. As the syntax of availability automata does not allow for counter assignments, we use a mapping μ as part of the state of the deterministic automaton. It yields for each state in the current set of states the availability counter that stores the highest possible availability. In the running example, p' is assigned counter x_p. We shall also need to compare two counters. If p' is also reached from q, we need to know whether x_p is higher than x_q. Therefore, we keep an order $\succ$ on the counters as part of the deterministic state.

Given a simple availability automaton $\mathcal{A} = (Q, Q_I, Q_F, X, \rightarrow, c)$ with its active counter mapping $\mu_{\mathcal{A}} : Q \rightarrow X$, we construct an equivalent deterministic automaton $det(\mathcal{A})$. As argued, we need a counter for each state, $X^d = Q$. Additionally, we keep a copy of the counters $\overline{X}^d$. While x measures $A \gtrsim k$, counter $\bar{x}$ observes $A \lesssim k$. This allows for explicit checks of violations of an availability constraint. States Q^d of the deterministic automaton are triples $(\boldsymbol{q}, \mu, \succ)$ where

- $\boldsymbol{q} \subseteq Q$ is the set of possible states of the non-deterministic automaton like in the construction of Rabin and Scott.
- $\mu : \boldsymbol{q} \rightarrow X^d$ assigns to each possible state the counter that stores the corresponding availability. If a state $q \in \boldsymbol{q}$ is mapped to a counter $\mu(q) = x_{q'}$ corresponding to a state $q' \in Q$, we additionally require that q and q' have the same active counter $\mu_{\mathcal{A}}(q) = \mu_{\mathcal{A}}(q')$.
- $\succ \subseteq \mu(\boldsymbol{q}) \times \mu(\boldsymbol{q})$ is a order on the counters. First, the order is compatible with the counter valuations γ in a run, i.e., $y \succ x$ guarantees $\gamma(y) \geq \gamma(x)$. Additionally, $\succ$ is a linear order on those counters that correspond to the same counter in the original automaton $\mathcal{A}$. More precisely, for $q, q' \in \boldsymbol{q}$ with $\mu(q) \neq \mu(q')$, the corresponding counters are ordered ($\mu(q) \succ \mu(q')$ or $\mu(q') \succ \mu(q)$) if and only if the states q and q' have the same active counter $\mu_{\mathcal{A}}(q) = \mu_{\mathcal{A}}(q')$.

We define the powerset automaton as

$$det(\mathcal{A}) = (Q^d, Q^d_0, Q^d_F, X^d \cup \overline{X}^d, \rightarrow^d, c^d).$$

The initial state is $Q^d_0 = \{(Q_I, \mu_0, \succ_0)\}$ with the initial mapping μ_0 that assigns to each initial state $q \in Q_I$ the corresponding counter $x_q \in X^d$ and $\succ_0$ arbitrary. The final states are the states that contain a final state of the original automaton, $Q^d_F = \{(\boldsymbol{q}, \mu, \succ) \mid \boldsymbol{q} \cap Q_F \neq \emptyset\}$. The constraints c^d on the counters in X^d are taken from $\mathcal{A}$, $c^d(x_q) = c(\mu_{\mathcal{A}}(q))$ for $x_q \in X^d$. The counter $\overline{x}$ measures the inverse constraint. Since $A < k$ is equivalent to $\Sigma \setminus A > 1 - k$, we set $c^d(\overline{x}) = (\Sigma \setminus A > 1 - k)$ if $c^d(x) = (A \geq k)$, and similar for $A > k$.

In the deterministic automaton, the edges are labelled by checks C^d that contain for each $x \in X^d$ either x or $\overline{x}$. Furthermore, we require consistency of C^d with the order $\succ_1$ of the source location:

$$y \succ_1 x \quad \text{and} \quad x \in C^d \quad \text{implies} \quad y \in C^d.$$

Since the counters $x, \overline{x}$ measure opposite constraints and are reset at the same time, we have $\gamma(x) \gtrsim 0$ iff $\gamma(\overline{x}) \lesssim 0$ for every reachable counter valuation γ. This means at most one set of constraints C^d is satisfied by γ. Moreover, as was discussed, reachable γ are compatible with the order in the state. So, for $y \succ_1 x$ we obtain $\gamma(y) \geq \gamma(x)$ and thus $\gamma(x) \gtrsim 0$ implies $\gamma(y) \gtrsim 0$. Therefore, for each γ there is exactly one enabled C^d.

To define the transition relation $\rightarrow^d$ of the deterministic automaton we define the unique successor state $(\boldsymbol{q}_2, \mu_2, \succ_2)$ for each state $(\boldsymbol{q}_1, \mu_1, \succ_1)$, each symbol $a \in \Sigma$, and each constraint set C^d. The enabled transitions of the non-deterministic automaton are the outgoing a-labelled transitions from states in $\boldsymbol{q}_1$ for which the guard is true,

$$\begin{aligned} \mathit{enabled} := \{&(q_1, a, \emptyset, Y, q_2) \in \rightarrow \mid q_1 \in \boldsymbol{q}_1\} \\ &\cup \{(q_1, a, \{\mu_{\mathcal{A}}(q_1)\}, Y, q_2) \in \rightarrow \mid q_1 \in \boldsymbol{q}_1, \mu_1(q_1) \in C^d\}. \end{aligned}$$

The first part of the successor state $\boldsymbol{q}_2$ is computed as for Rabin and Scott. A state q_2 is in the set $\boldsymbol{q}_2$ if there is an enabled transition leading to it,

$$\boldsymbol{q}_2 := \{q_2 \in Q \mid \exists q_1, C, Y : (q_1, a, C, Y, q_2) \in \mathit{enabled}\}.$$

The difficult part is computing the new counter valuations for each state $q_2 \in \boldsymbol{q}_2$. If an enabled transition to q_2 resets the counter $\mu_{\mathcal{A}}(q_2)$ we may need a fresh counter in $\mathit{det}(\mathcal{A})$ that is reset on the transition. The counter of q_2 can also be inherited from a source location if there is an edge in the non-deterministic automaton that does not reset the counter. Let x_{q_2} denote a fresh counter for q_2, then the new counter used as $\mu_2(q_2)$ is the counter with the highest value from the set of candidates

$$\begin{aligned} \mathit{cand}(q_2) := \{&x_{q_2} \mid \exists q_1, C, Y : (q_1, a, C, Y, q_2) \in \mathit{enabled}, \mu_{\mathcal{A}}(q_2) \in Y\} \\ &\cup \{\mu_1(q_1) \mid \exists C, Y : (q_1, a, C, Y, q_2) \in \mathit{enabled}, \mu_{\mathcal{A}}(q_2) \notin Y\}. \end{aligned}$$

The set $\mathit{cand}(q_2)$ is non-empty for all $q_2 \in \boldsymbol{q}_2$ and contains only counters x_q with $\mu_{\mathcal{A}}(q) = \mu_{\mathcal{A}}(q_2)$. The latter holds by definition of simple automata. A non-resetting transition from q_1 to q_2 requires the states to have the same active counter, $\mu_{\mathcal{A}}(q_1) = \mu_{\mathcal{A}}(q_2)$. The fresh counters x_{q_2} are inserted into the order $\succ_1$ based on the guard C^d. This yields a new order $\succ_2'$. For every state $q_1 \in \boldsymbol{q}_1$ with the same active counter in $\mathcal{A}$, we check whether $\mu_1(q_1)$ is positive, which means $\mu_1(q_1) \in C^d$. In this case, we add $\mu_1(q_1) \succ_2' x_{q_2}$, otherwise $x_{q_2} \succ_2' \mu_1(q_1)$.

The new counter for q_2 is the one from $cand(q_2)$ with the largest value,

$$\mu_2(q_2) := \max_{\succ_2'} cand(q_2) \quad \text{for } q_2 \in \boldsymbol{q}_2 \,.$$

This counter is well-defined since $cand(q_2)$ is a non-empty finite set on which $\succ_2'$ is a linear order. We obtain $\succ_2$ from $\succ_2'$ by removing all unused counters,

$$\succ_2 := \succ_2' \cap (\mu_2(\boldsymbol{q}_2) \times \mu_2(\boldsymbol{q}_2)).$$

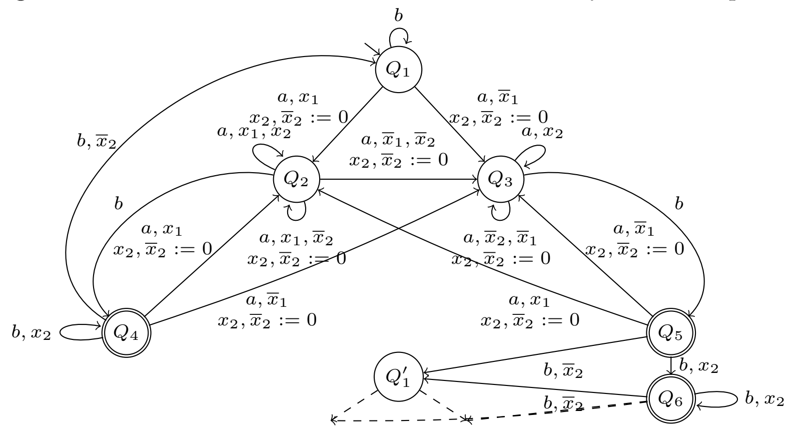
For the fresh counter x_{q_2}, we can use any counter x_q that is not used in $\mu_2(\boldsymbol{q}_2)$ for other purposes. Moreover, this counter should belong to a state q with the same active counter as q_2. If multiple states q_2, q_2' with the same active counter $\mu_{\mathcal{A}}(q_2) = \mu_{\mathcal{A}}(q_2')$ need a fresh counter, we choose the same fresh counter $x_{q_2} = x_{q_2'}$ in X^d. Since each state only uses a single counter, and since we have as many counters as states, there must always be a free counter available if $\mu_2(q_2)$ needs a fresh one. Finally the edge $((\boldsymbol{q}_1, \mu_1, \succ_1), a, C^d, Y^d, (\boldsymbol{q}_2, \mu_2, \succ_2))$ is added to the transition relation $\rightarrow^d$, where $Y^d = \{x_{q_2}, \overline{x}_{q_2} \mid q_2 \in \boldsymbol{q}_2, \mu_2(q_2) = x_{q_2}\}$ is the set of fresh counters that are used in the new state.

Example 2. We apply the powerset construction to the automaton from Section 3. The resulting deterministic automaton is depicted on the next page. It has twelve reachable states $Q^d = \{Q_i, Q_i' \mid 1 \leq i \leq 6\}$ of which seven are given in the figure. The states are $Q_1 = (\{q_1\}, \{q_1 \mapsto x_1\}, \emptyset), Q_2 = (\{q_1, q_2\}, \{q_1 \mapsto x_1, q_2 \mapsto x_2\}, x_1 \succ x_2), Q_3 = (\{q_1, q_2\}, \{q_1 \mapsto x_1, q_2 \mapsto x_2\}, x_2 \succ x_1), Q_4 = (\{q_1, q_3\}, \{q_1 \mapsto x_1, q_3 \mapsto x_2\}, x_1 \succ x_2), Q_5 = (\{q_1, q_3\}, \{q_1 \mapsto x_1, q_3 \mapsto x_2\}, x_2 \succ x_1), Q_6 = (\{q_1, q_3\}, \{q_1 \mapsto x_2, q_3 \mapsto x_2\}, \emptyset)$. The states Q_i' are the states Q_i where the counters x_1 and x_2 are swapped. For space reasons we depict a check on a counter by labelling the edge with the counter. The counters x_1, x_2 observe the constraint $\{a\} \geq 1/2$, while $\overline{x}_1, \overline{x}_2$ check the inverse constraint $\{b\} > 1/2$.

The initial state Q_1 corresponds to the initial state q_1 in the original automaton. Under the input symbol b only the loop edge $(q_1, b, \emptyset, \emptyset, q_1)$ is enabled. Therefore for every counter constraint C^d the successor state is Q_1 again. We combined the edges $(Q_1, b, \{x_1\}, \emptyset, Q_1)$ and $(Q_1, b, \{\overline{x}_1\}, \emptyset, Q_1)$ to a single edge without counter constraints $(Q_1, b, \emptyset, \emptyset, Q_1)$. For the symbol a there are two enabled edges in the original automaton leading to q_1 and q_2. The edge to q_2 resets the counter. Therefore, we introduce a fresh counter x_2 for q_2. For computing the order between x_2 and x_1, we check the sign of x_1. The successor is state Q_2 with $x_1 \succ x_2$ if x_1 is positive and Q_3 with $x_2 \succ x_1$, otherwise.

Now we consider the outgoing edges from state Q_5. The a-labelled edges are the same as from Q_1, since there is no a-labelled edge starting from q_3. For symbol b the successor state depends on C^d. If $\overline{x}_2 \in C^d$, there are two enabled edges in the original automaton, namely $(q_1, b, \emptyset, \emptyset, q_1)$ and $(q_3, b, \emptyset, \emptyset, q_1)$. Both enter q_1 and do not reset the counter. The candidates for the new counter of q_1

are x_1 or x_2. By the linear order, x_2 is larger in Q_5, hence this is the counter used in the successor state Q_1'. If $x_2 \in C^d$, then the edge $(q_3, b, \{x_{\{a\}}\}, \emptyset, q_3)$ of the original automaton is enabled. The successor state Q_6 contains q_1 and q_3.


Q_1
Q_2
Q_3
Q_4
Q_5
Q_6
Q_1'
b
a, x_1
$x_2, \overline{x}_2 := 0$
$a, \overline{x}_1$
$x_2, \overline{x}_2 := 0$
a, x_1, x_2
a, x_2
$b, \overline{x}_2$
$a, \overline{x}_1, \overline{x}_2$
$x_2, \overline{x}_2 := 0$
b
b
a, x_1
$x_2, \overline{x}_2 := 0$
$a, x_1, \overline{x}_2$
$x_2, \overline{x}_2 := 0$
$a, \overline{x}_2, \overline{x}_1$
$x_2, \overline{x}_2 := 0$
$a, \overline{x}_1$
$x_2, \overline{x}_2 := 0$
$a, \overline{x}_1$
$x_2, \overline{x}_2 := 0$
a, x_1
$x_2, \overline{x}_2 := 0$
b, x_2
$b, \overline{x}_2$
b, x_2
$b, \overline{x}_2$
b, x_2

Proposition 4. *$\mathcal{L}\left(det(\mathcal{A})\right) = \mathcal{L}\left(\mathcal{A}\right)$.*

The proof is given in the appendix. A deterministic automaton is complemented by inverting the set of final states, $\overline{det(\mathcal{A})} = (Q^d, Q_0^d, Q^d \setminus Q_F^d, X^d \cup \overline{X}^d, \rightarrow^d, c^d)$.

Proposition 5. *$\mathcal{L}\left(\overline{det(\mathcal{A})}\right) = \overline{\mathcal{L}\left(det(\mathcal{A})\right)}$.*

The construction shows that the set of languages accepted by simple availability automata is a subset of the languages accepted by deterministic availability automata, which in turn is a subset of the languages of all availability automata. As simple automata can be simulated by one-counter machines, their emptiness problem is decidable. The intersection problem is not, since it is not decidable for flat raes. Hence, simple automata are not closed under intersection, and thus not closed under complementation. General availability automata are also not closed under complementation. Like for timed automata [AD94] one can argue that the complement of $(a + b + c)^*.c.((a + b + c)^*)_{a=1/2}.c.(a + b + c)^*$ is not accepted by any availability automaton. Since deterministic automata are closed under complementation this shows that their expressiveness is strictly between simple and general automata.

4.1 Application to Verification

Equipped with the previous complementation algorithm, we are able to tackle the following verification problems for discrete systems under availability constraints.

$$(\textbf{Syn}) \quad ? \models \mathbb{B}_{rae} \qquad\qquad (\textbf{MC}) \quad \mathcal{A} \models \mathbb{B}_{rae}.$$

The *synthesis* problem **Syn** asks for the most general availability automaton that satisfies a Boolean combination of availability expressions $\mathbb{B}_{rae}$. As usual, satisfaction is defined in terms of language inclusion. The model checking problem **MC** takes an availability automaton $\mathcal{A}$ modelling the system of interest and an availability expression $\mathbb{B}_{rae}$ that formalises the correctness condition. It reduces the problem whether $\mathcal{A}$ is a model of $\mathbb{B}_{rae}$ to a reachability query.

Our complementation algorithm is restricted to *simple* availability automata. Relying on the negation normal form for $\mathbb{B}_{rae}$, the following theorem is an immediate consequence of our previous efforts.

Theorem 3. *Consider a Boolean combination of flat raes. There is an algorithm that solves* ***Syn****.* ***MC*** *is reducible to reachability in availability automata.*

By Theorem 1, reachability in availability automata is of course undecidable. The definition of runs however suggests to view availability automata as particular counter automata. Therefore, Theorem 3 allows us to use state-of-the-art tools like SLAM [BR02] or BLAST [BHJM07] to solve **MC**. They support abstraction-aided reachability analysis and often successfully tackle this undecidable problem in practically relevant cases.

5 Conclusion

We defined an extension of regular expressions to specify the availability of systems. We developed a corresponding automaton model and established a full Kleene theorem. It shows that the two denote the same formal languages. An undecidability proof places the models between finite automata and counter automata. Finally, we give a complementation algorithm for the restricted simple availability automata. It yields a fully automated synthesis procedure for a practically significant class of regular availability expressions. Moreover, it allows for a reduction of availability model checking to reachability analysis.

Related work. Various extensions of regular expressions have been proposed. For example, in [ACM02] timed regular expressions are introduced and proven equivalent to timed automata of [AD94] by a timed analogue of Kleene's theorem. The availability formula in the introduction can be stated directly in real-time logics like the Duration Calculus [CH04] or investigated operationally in suitable subclasses of hybrid automata like stopwatch automata [CL00] or priced timed automata [LR08]. The advantage of the discrete setting we chose is that the essentials of availability can be studied in isolation without being overwhelmed with the technicalities of continuous timed and hybrid systems.

Availability automata are also closely related to the work on weighted automata [DKV09]. There, the alphabet is equipped with a weight function that assigns each letter a weight in some semiring. Examples include the interesting semiring $([0,1], max, \cdot, 0, 1)$. It can be used to determine the *reliability* of a word by assigning a probability $k \in [0,1]$ to each letter. Crucially different from our model, weighted automata do not have guards and thus the measurement does not influence the system behaviour. We employ checks and resets on the availability to mimic loop invariants as they are standard in programming languages.

In Presburger regular expressions [SSM03], a regular language is constrained by additional Presburger formulae. Our initial example of a network with 99 % availability can be specified in this formalism as $(up + down)^* \wedge x_{up} \geq 99x_{down}$.

Again, as opposed to our approach, conditional executions based on intermediary valuations are not supported. Moreover, our resets allow for an unbounded number of measurements whereas Presburger regular expressions only have a finite number of arithmetic constraints.

Future work. Our results are only a first step in the study of quantitative system properties. We plan to extend the work to ω-regular and to timed languages. Also logical characterisations of availability languages seem interesting.

Practically, we envision the following application of the presented technique. By stochastic techniques, we establish availability constraints $rae_{A \gtrsim k}$ that model the components of a distributed system [dSeSG89,RS93]. When reasoning about their interaction, we abstract away the stochastic information and rely on our new availability models. Proximity to integer programs allows us to reuse efficient software model checkers for their analysis [BR02,BHJM07].

Our undecidability proof of the intersection problem (Theorem 1) requires two rae. We leave open decidability of the *emptiness problem* for a single rae.

References

[ACM02] E. Asarin, P. Caspi, and O. Maler. Timed regular expressions. *Journal of the ACM*, 49:172–206, 2002.

[AD94] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[BHJM07] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker BLAST. *STTT*, 9(5–6):505–525, 2007.

[BR02] T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. In *Proc. of POPL*, pages 1–3. ACM, 2002.

[CH04] Z. Chaochen and M. R. Hansen. *Duration Calculus: A Formal Approach to Real-Time Systems*. EATCS Monographs. Springer, 2004.

[CL00] F. Cassez and K. G. Larsen. The impressive power of stopwatches. In *Proc. of CONCUR*, volume 1877 of *LNCS*, pages 138–152. Springer, 2000.

[DKV09] M. Droste, W. Kuich, and H. Vogler, editors. *Handbook of Weighted Automata*. EATCS Monographs. Springer, 2009.

[dSeSG89] E. de Souza e Silva and H. R. Gail. Calculating availability and performability measures of repairable computer systems using randomization. *Journal of the ACM*, 36(1):171–193, 1989.

[LR08] K. G. Larsen and J. I. Rasmussen. Automata-theoretic techniques for modal logics of programs. *Theoretical Computer Science*, 390:197–213, 2008.

[Min67] M. Minsky. *Computation: Finite and Infinite Machines*. Prentice Hall, 1967.

[RS59] M. O. Rabin and D. S. Scott. Finite automata and their decision problems. *IBM Journal of Research*, 3(2):115–125, 1959.

[RS93] G. Rubino and B. Sericola. Interval availability distribution computation. In *Proc. of FTCS*, pages 48–55. IEEE, 1993.

[SSM03] H. Seidl, T. Schwentick, and A. Muscholl. Numerical document queries. In *Proc. of PODS*, pages 155–166. ACM, 2003.

[Tri01] K. S. Trivedi. *Probability and Statistics with Reliability, Queuing, and Computer Science Applications*. Wiley, 2nd edition, 2001.