A Theory of Register Monitors

Thomas Ferrère
IST Austria

Thomas A. Henzinger
IST Austria

N. Ege Saraç
Sabancı University

ABSTRACT

The task of a monitor is to watch, at run-time, the execution of a reactive system, and signal the occurrence of a safety violation in the observed sequence of events. While finite-state monitors have been studied extensively, in practice, monitoring software also makes use of unbounded memory. We define a model of automata equipped with integer-valued registers which can execute only a bounded number of instructions between consecutive events, and thus can form the theoretical basis for the study of infinite-state monitors. We classify these *register monitors* according to the number k of available registers, and the type of register instructions. In stark contrast to the theory of computability for register machines, we prove that for every $k \geq 1$, monitors with $k+1$ counters (with instruction set $\langle +1, = \rangle$) are strictly more expressive than monitors with k counters. We also show that adder monitors (with instruction set $\langle 1, +, = \rangle$) are strictly more expressive than counter monitors, but are complete for monitoring all computable safety ω-languages for $k = 6$. *Real-time monitors* are further required to signal the occurrence of a safety violation as soon as it occurs. The expressiveness hierarchy for counter monitors carries over to real-time monitors. We then show that 2 adders cannot simulate 3 counters in real-time. Finally, we show that real-time adder monitors with inequalities are as expressive as real-time Turing machines.

CCS CONCEPTS

• **Theory of computation** → *Quantitative automata*; *Logic and verification*; • **Computer systems organization** → *Real-time system specification*;

ACM Reference Format:
Thomas Ferrère, Thomas A. Henzinger, and N. Ege Saraç. 2018. A Theory of Register Monitors. In *LICS '18: LICS '18: 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, July 9–12, 2018, Oxford, United Kingdom.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3209108.3209194

1 INTRODUCTION

The safety of reactive systems [27] can be guaranteed through the use of several techniques, such as rigorous design principles, systematic testing, or formal verification. Run-time monitoring is one such technique [26]. It conjoins the system with a module dedicated to ensuring that the sequence of events produced by the system is correct. The monitor works on-line, and upon detection of incorrect behavior can react, e.g., by interrupting the system.

LICS '18, July 9–12, 2018, Oxford, United Kingdom

ACM ISBN 978-1-4503-5583-4/18/07…$15.00
https://doi.org/10.1145/3209108.3209194

In theoretical terms, any monitor recognizes a safety ω-language [39]. Safety languages are characterized by the fact that an invalid sequence of events can always be identified by a finite prefix. An important subclass is that of ω-regular safety languages. These languages can be recognized by deterministic finite automata without acceptance condition, and their applications in formal verification and monitoring have been extensively studied, see e.g., [7, 17, 40].

Many interesting properties of reactive systems are not ω-regular. For instance, consider a server that can receive requests with event a, issue grants with event b, be activated and deactivated with event c. Property ψ_1 of this system is that while active, every grant must be matched by an earlier request. It corresponds to the language of ω-words in which every finite subword beginning on an odd occurrence of c and ending before the next occurrence of c features more a's than b's. Property ψ_1 is evidently not finite-state due to a potentially unbounded number of pending requests. Real-life examples of properties of this kind are plenty.

We propose an automaton model that captures the task of monitoring safety properties that lie beyond ω-regular. In practice, runtime monitors are programs that have access to a large, potentially unbounded memory unit. Our machine model uses integer-valued registers as for the register machines from computability theory [29, 38]. This is particularly suited to model the monitoring of safety properties expressed in terms of some quantities involved in the computation of the observed system, such as time, energy, or other functional indicators. The resulting notion of monitorability is very different from computability, where two counter registers suffice for Turing computation [30].

The register monitor shown in Figure 1 recognizes property ψ_1 of our example. It runs as long as the property is satisfied, and halts in case of violation (this is indicated by no transitions being available). Deactivating the server causes pending requests to be

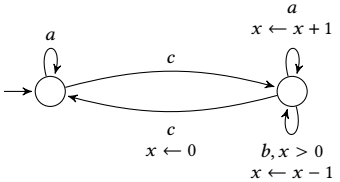

Figure 1: A register monitor for property ψ_1.

dropped, which is reflected by the update $x \leftarrow 0$. Requests and grants are processed using increments and decrements.

Consider now more powerful arithmetic operations such as *addition*. Property ψ_2 of the server requires that the average over time of the number of pending requests never exceeds 5. Requests are assumed to be granted in the order with which they arrive. A register monitor recognizing this safety property is shown in Figure 2. The sum of pending requests is maintained in a register r with additive

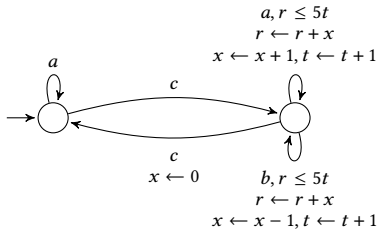

Figure 2: A register monitor for property ψ_2.

updates $r \leftarrow r + x$. Currently pending requests x are counted as previously, and time is counted in t.

Observe that monitors of both Figure 1 and Figure 2 halt in *real time*, in the sense that every safety violation causes the monitor to halt as soon as it occurs. This is not strictly required by our definitions, we also allow for a monitor to delay its verdict as necessary to complete its computation.

Consider a variant of our server example, in which every request and grant is preceded by a key communicated in binary form using letters d and e; the key is broadcast during the inactive period of the server. Property ψ_3 requires that every grant and request use the key set during the previous inactive period of the server. A register monitor recognizing this safety property is shown in Figure 3. During a period of inactivity, the monitor uses register x to store the key and register y to store its length. During an active period, the monitor uses alternate registers x' and y' to encode every key. The length of the key being read should not exceed that of the correct key ($y' \leq y$), and upon occurrence of a request or grant the two keys should match ($x' = x$). The monitor of Figure 3 halts in linear-time in the worst case, it does not halt on the first erroneous bit. This should be sufficient in practice.

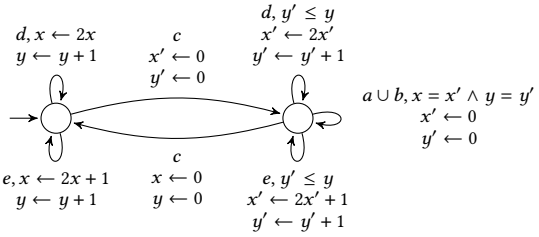

Figure 3: A register monitor for property ψ_3.

In this work we mainly focus on monitors that halt in real time, also called real-time monitors for short. We obtain the following results.

Counter monitors are defined by considering the instruction set $\langle +1, = \rangle$. We prove that $(k+1)$-counter monitors are more expressive than k-counter monitors for every $k \geq 1$. This holds both for general monitors and for the class of monitors that halt in real time. We also show that in counter monitors, resetting variables to zero as in Figure 1 is equivalent to copying variables, and is an essential operation. In general, *copyless* counter monitors are less expressive.

Adder monitors are defined by considering the instruction set $\langle 1, +, = \rangle$. Adder monitors with 6 registers are complete for the class of Turing computable safety languages, so unlike the counter hierarchy, the adder hierarchy collapses at $k = 6$. This is because using 2 adders, the pending part of an input ω-word can be encoded in real time, and using a further 4 counters an arbitrary Turing computation can be done asynchronously in parallel. We conjecture that on the contrary the real-time adder hierarchy does not collapse at any k. As a first step in this direction, we show that in real-time monitors, 3 counters are more powerful than 2 adders.

Linear register monitors are then defined by considering linear arithmetic updates and inequality tests $\leq$. We obtain that k counters can be simulated by 4 linear registers in real time, for any $k \geq 0$. We finally define *polynomial register monitors* by enabling the multiplication operation. This seemingly powerful model still cannot recognize in real time the language of words in which "no number repeats", consisting of words $\#w_1\#w_2\#w_3\#\ldots$ in which $w_i \neq w_j$ for every $i \neq j$. The above language could specify a security protocol where every key should be unique, also known as a *nonce*. It is provably not recognizable by real-time polynomial register monitors. When the numbers are presented in unary, the resulting language is recognizable by a linear register monitor in real time.

Related Work

Register machines were introduced in [29] as an abstract model of computation. The notion of real-time computation was, to our knowledge, first studied in [41]. Real-time Turing machines were defined and investigated by [33], showing that 2-tape machines are more expressive than 1-tape machines. A language that is not real-time recognizable was described explicitly for the first time in [20]. Real-time computation in general was also studied in [35].

Defining real-time counter machines as language recognizers and sequence generators, [14, 15] characterize the power of counter machines relative to Turing machines and establish a hierarchy of $k + 1$ versus k real-time counters. A hierarchy of $k + 1$ versus k tapes for real-time Turing machines was proved in [1]. Following the introduction of the information-theoretic approach by [31], this result was extended to other models of on-line computations [28].

Formal verification problems using temporal logic and arithmetic registers were surveyed in [9]. Reachability in register machines with polynomial updates was studied by [13], and is undecidable for more than one register. The use of formal languages in the context of monitoring was advocated by [16]. Due to the predominance of linear temporal logic in formal verification, the study of formal monitoring has largely been focused on finite-state languages [26]. The field of run-time verification [25] is also concerned with software engineering issues, which need not involve real-time constraints. A notable exception is the model proposed by [8] for on-line monitoring with Boolean and integer-valued stream variables.

The term *register automata* sometimes refers to computational models with register variables ranging over an infinite input alphabet, see [10, 19, 22]. The use of register automata for run-time verification was studied in [18]. In [34] was proposed a monitor model consisting of automata in which registers, ranging over input values, are bound by first-order quantifiers. We refer the reader to [21] for a survey of the monitoring of data languages. Unlike all of the above, in this work we assume a finite input alphabet of events.

Automata with registers were also defined as studied in [3] as a generalization of weighted automata. We borrow the notion of *copyless* registers from [3]. The extension of cost registers to arithmetic operations was studied in [6]. Cost-register automata also aim at providing foundations for regular stream processing languages [4]. The application of stream processing to network monitoring was considered in [42]. In general, the work on stream processing focuses on the algebraic specification, and efficient compilation of on-line stream processors. We instead study the notion of infinite-state monitorability.

To our knowledge, this paper is the first systematic theoretical study of register machines as on-line and real-time monitors.

2 DEFINITIONS

Let Σ be a finite alphabet of events. The length of a finite word $w \in \Sigma^*$ is denoted $|w|$. Given words $u \in \Sigma^*$ and $w \in \Sigma^* \cup \Sigma^\omega$, we write $u \prec w$, and say that u is a (strict) prefix of w, when there exists v with $|v| > 0$ such that $uv = w$.

2.1 Safety

A word $u \in \Sigma^*$ is a *bad prefix* for some language $L \subseteq \Sigma^\omega$ when for all words $w \in \Sigma^\omega$, if $u \prec w$ then $w \notin L$. A language $L \subseteq \Sigma^\omega$ is a *safety language* when for all $w \notin L$ there exists a bad prefix $u \in \Sigma^*$ for L with $u \prec w$. These definitions conform to [2].

Safety constitutes a privileged class of properties, for which a monitor is able to produce a permanent violation verdict, when a bad prefix is observed. Note that in general other kinds of properties can be considered to be amenable to monitoring [11]. Allowing such properties immediately raises the question of what type of verdict or reaction is expected of a monitor, since a seemingly bad prefix can sometimes be prolonged into a satisfying execution. In this paper we restrict our attention to safety properties.

Not all safety properties ought to be considered monitorable, for the simple reason that they may not be computable in a reasonable sense. Since we work with ω-words, we will naturally consider non-terminating computations and take the following definition: A safety language L is said to be *computable* when there exists a Turing machine that halts precisely on the infinite words not in L.

2.2 Register Monitors

Take X to be a set of integer variables, called *registers*. Let T be a set of functions and relations over the integers, called *instruction set*. An *update* is a mapping from variables to terms over T. A *test* is a conjunction of atomic formulas over T and their negation. The set of updates and tests on variables X are respectively denoted $\Gamma(X)$ and $\Phi(X)$. For any register valuation $v : X \to \mathbb{Z}$ and update $\gamma \in \Gamma(X)$, we define the updated valuation $v[\gamma] : X \to \mathbb{Z}$ by letting $v[\gamma](x) = v(\gamma(x))$ for all $x \in X$. For any $\phi \in \Phi(X)$ we write $v \models \phi$ when ϕ holds true under valuation v.

Definition 2.1 (Monitor). A (deterministic) *register monitor* is a tuple $(\Sigma, X, Q, s, \Delta)$ where Σ is an alphabet, X is a set of registers, Q is a set of control locations, $s \in Q$ is the initial location, and $\Delta \subseteq Q \times \Sigma \times \Phi(X) \times \Gamma(X) \times Q$ is a set of edges such that for every $(q, \sigma, \phi_1, \gamma_1, r_1) \neq (q, \sigma, \phi_2, \gamma_2, r_2) \in \Delta$ the formula $\phi_1 \wedge \phi_2$ is unsatisfiable. The sets Σ, X, Q, Δ are assumed finite.

Let $\mathcal{A} = (\Sigma, X, Q, s, \Delta)$ be a register monitor. A *configuration* of $\mathcal{A}$ is a pair (q, v) of location $q \in Q$ and valuation $v : X \to \mathbb{Z}$. Let $\sigma \in \Sigma$ be an event. A *transition* $\xrightarrow{\sigma}$ of $\mathcal{A}$ is a relation between configurations defined by $(q, v) \xrightarrow{\sigma} (q', v')$ iff $v' = v[\gamma]$ and $v \models \phi$ for some edge $(q, \sigma, \phi, \gamma, q') \in \Delta$. A *run* of automaton $\mathcal{A}$ over some word $w = \sigma_1\sigma_2\sigma_3 \ldots$ is a valid sequence of transitions

$$(q_0, v_0) \xrightarrow{\sigma_1} (q_1, v_1) \xrightarrow{\sigma_2} (q_2, v_2) \xrightarrow{\sigma_3} \ldots$$

labeled by w where $q_0 = s$ and $v_0(x) = 0$ for all $x \in X$. An infinite word w is *accepted* by $\mathcal{A}$ when $\mathcal{A}$ has a run over w. We say that $\mathcal{A}$ *errs* over a finite word w when $\mathcal{A}$ had no run over w. We call language of $\mathcal{A}$ and denote by $L(\mathcal{A})$ the set of infinite words accepted by $\mathcal{A}$.

Let $\tau : \mathbb{N} \to \mathbb{N}$ be a monotone function. A register monitor $\mathcal{A}$ is said to halt *in time* τ when for every bad prefix w of $L(\mathcal{A})$, there exists a suffix $v \succeq w$ such that $|v| \leq \tau(|w|)$ and $\mathcal{A}$ errs on v. We say that $\mathcal{A}'$ *simulates* $\mathcal{A}$ in time τ' when $L(\mathcal{A}) = L(\mathcal{A}')$ and for every τ such that $\mathcal{A}$ halts in time τ, $\mathcal{A}'$ halts in time $\tau' \circ \tau$. Monitors $\mathcal{A}$ and $\mathcal{A}'$ are said to be *time-equivalent* when they simulate each other in real time. Here and throughout this paper, in *real time* means in time $n \mapsto n$. Observe that two monitors are time-equivalent iff they err on the same set of prefixes.

2.3 Closure Properties

Like the class of safety languages that contains it, the class of register monitorable languages is closed under positive Boolean operations. This is easily shown using standard product constructions over deterministic automata.

Theorem 2.2. *For any instruction set T and function τ, the set of languages recognizable by register monitors in time τ is closed under union and intersection.*

The set of register monitorable languages is obviously not closed under complement. Over the alphabet $\{a, b\}$, the language a^ω is register monitorable but its complement is not safety. For a given instruction set T and function τ, the set of languages monitorable in time τ is usually not closed under projection.

Register monitors have a finite set of edges, so that any given monitor only executes boundedly many instructions per input symbol. Each register instructions can be seen as an oracle, taking constant computation time. Let us call *size* of a term the height of its syntax tree.

Definition 2.3 (Rate). A register monitor has *rate* c when the size of terms it features is at most c.

The rate of a monitor is intuitively related to its maximum number of instructions per test or update.

Define *register ϵ-monitors* similarly as register monitors, but with transitions taken in $Q \times (\Sigma \cup \{\epsilon\}) \times \Phi(X) \times \Gamma(X) \times Q$ and terms of size at most 1. *Silent* transitions, labeled with ϵ, do not consume any input symbol and can occur at arbitrary positions in a run.

Definition 2.4 (Delay). A register ϵ-monitor has *delay* c when its maximum number of consecutive silent transitions is at most $c - 1$.

Theorem 2.5. *Let c be a positive integer. Register ϵ-monitors with delay c and register monitors with rate c are time-equivalent formalisms.*

3 FINITE-STATE MONITORS

A register monitor with an empty instruction set, or equivalently, without registers, is called a *finite-state* monitor. They constitute a broad class of monitors which capture well-studied properties.

Example 3.1. Consider the language $(a \cup ba)^\omega$ over events a and b, consisting of infinite words in which there are no consecutive b events. This safety language can be recognized by the finite-state monitor of Figure 4.

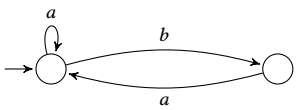

Figure 4: A finite-state monitor for the language $(a \cup ba)^\omega$.

Finite-state monitors correspond to the subclass of Büchi automata with trivial acceptance condition, also known as *safety automata*. The use of finite and Büchi automata as a monitor model was considered before in multiple related research efforts, see [16] in particular. For the case of safety languages, we make the following observation:

Theorem 3.2. *For any finite-state monitor $\mathcal{A}$ with n states, there exists a* real-time *finite-state monitor $\mathcal{A}'$ with $n' \leq n$ states equivalent to $\mathcal{A}$.*

Proof. We say that a location q of $\mathcal{A}$ is *doomed* if no cycle in the transition graph of $\mathcal{A}$ is reachable from q. Removing all doomed states of $\mathcal{A}$ yields $\mathcal{A}'$. One can easily check that $L(\mathcal{A}) = L(\mathcal{A}')$ (a prefix w is bad for $L(\mathcal{A})$ iff w is bad for $L(\mathcal{A}')$) and that $\mathcal{A}'$ is real-time ($\mathcal{A}'$ errs on all its bad prefixes). □

This contrasts with the situation of general ω-regular languages, including co-safety ones. The construction of a recognizer of bad prefixes from a nondeterministic Büchi automaton involves an unavoidable exponential increase in the number of states [24], even when using nondeterminism.

The applications of finite automata for monitoring and run-time verification are numerous and well-studied, see [7, 17] in particular. The special status of ω-regular languages is related to the decidability of their inclusion problem, upon which many formal verification results rely. For monitoring purposes, more expressive formalisms can and should be investigated.

4 COUNTER MONITORS

Definition 4.1 (Counter Monitors). A register monitor with the instruction set $\langle +1, = \rangle$ is called a *counter monitor.*

Let us give a typical example of safety language which is counter monitorable, but not finite-state monitorable. Given σ in some alphabet Σ, and $w \in \Sigma^*$, we denote by $|w|_\sigma$ the number of occurrences of event σ in the word w.

Example 4.2. Let $\Sigma = \{a, b\}$, and consider the language

$$L_1 = \{w \in \Sigma^\omega \mid \forall u \prec w, |u|_a \geq |u|_b\}.$$

If a stands for a request and b for a grant, language L_1 requires that every grant is matched by an earlier request (but not all requests may be granted). This language is recognized by the two-counter automaton of Figure 5, which counts occurrences of a in x and occurrences of b in y, and runs as long as $x \geq y$.

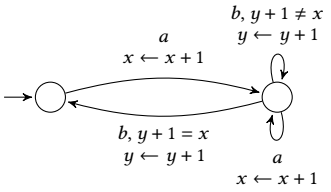

Figure 5: A counter monitor for the language L_1.

4.1 Counter Hierarchy

It was shown by [15] that in counter automata over finite words, every new counter creates additional expressive power. We show that this result carries over to our counter monitor model. For this we take the following language L_k as a witness. This language generalizes L_1 of Example 4.2 by taking an alphabet $\Sigma_k = \{0, 1, \ldots, k\}$ of $k + 1$ letters, and letting

$$L_k = \{w \in \Sigma_k^\omega \mid \forall i < k, \forall u \prec w, |u|_{i+1} \geq |u|_i\}.$$

It consists of all words in which every occurrence of event $i + 1$ must be matched by the occurrence of an earlier event i. For an alphabet with $k + 1$ letters, exactly $k + 1$ counters are needed.

Let L be a language over Σ. Prefixes $u_1, u_2 \in \Sigma^*$ are said to be *equivalent* relative to L, denoted $u_1 \sim_L u_2$, when $u_1 w \in L$ iff $u_2 w \in L$ for all infinite words $w \in \Sigma^\omega$. Let $\mathcal{A}$ be a counter monitor over the alphabet Σ. Configurations (q_1, v_1) and (q_2, v_2) are said to be *equivalent* when $(q_1, v_1) \xrightarrow{u} (q, v)$ iff $(q_2, v_2) \xrightarrow{u} (q, v)$ for all finite words $u \in \Sigma^*$. Here by $(q, v) \xrightarrow{u} (q', v')$ we denote a sequence of transitions from (q, v) to (q', v') and labeled with u.

Theorem 4.3. *For every $k \geq 1$, there exists a real-time $(k + 1)$-counter monitor without any equivalent k-counter monitor.*

Proof. Let us fix a number $k \geq 1$ of registers, and consider the language L_k. Notice that L_k is recognizable by a $(k + 1)$-counter monitor constructed similarly as for Example 4.2. We assume towards a contradiction that L_k is also recognizable by a k-counter monitor, and show that the number of inequivalent prefixes of length up to n relative to L_k is strictly greater than the number of possible configurations that a k-counter monitor can reach after reading a prefix of length up to n.

A simple condition suffices to characterize the equivalence relative to L_k: it holds that $u_1 \sim_{L_k} u_2$ iff there is an integer p for which $|u_1|_i = |u_2|_i + p$ holds for all $0 \leq i \leq k$. We represent each equivalence class of $\sim_{L_k}$ by a string u such that $|u|_i = 0$ for some i. Then, the number of equivalence classes of prefixes of length up to n, computed as a difference of two binomial coefficients, is bounded from below as follows:

$$\binom{n+k+1}{k+1} - \binom{n}{k+1} > \frac{(n+1)^{k+1} - n^{k+1}}{(k+1)!} > \frac{n^k}{k!}. \qquad (1)$$

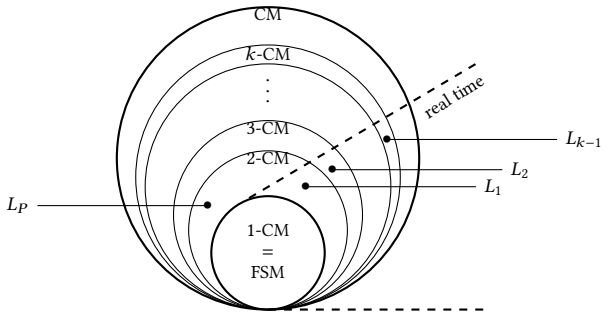

Figure 6: Counter hierarchy. FSM and CM stand for finite-state monitor and counter monitor, respectively.

For configurations of counter monitors we observe a similar condition: (q_1, v_1) and (q_2, v_2) are equivalent iff $q_1 = q_2$ and there exists an integer p for which $v_1(x) = v_2(x) + p$ holds for all $x \in X$. A configuration (q, v) denotes an equivalence class if $v(x) = 0$ for some $x \in X$. Now, consider an arbitrary k-counter monitor with m locations and rate c. The number of equivalence classes of configurations after it reads a prefix of length up to n is computed similarly and bounded from above as follows:

$$m\left[\binom{cn+k}{k} - \binom{cn}{k}\right] < m\left[\frac{(cn+k)^k}{k!} - \frac{(cn-k+1)^k}{k!}\right] < \frac{m(2k-1)}{(k-1)!}(cn+k)^{k-1}. \quad (2)$$

For sufficiently large n, the lower bound in (1) exceeds the upper bound in (2): There are more inequivalent prefixes of length up to n relative to L_k than configurations a k-counter monitor can possibly reach after reading such prefixes. Thus no k-counter monitor can recognize L_k. □

In Theorem 4.3, the k-counter simulator is not required to halt in real time. Observe that there exist languages counter-monitorable but not in real time. Such languages can be found in the form of $L_S = \bigcup_{n \in S} a^n b^\omega \cup a^\omega$ where S is a computable set with complexity more than exponential. For example, take P to be the set of valid Presburger arithmetic sentences given in binary notation. Since the complexity of deciding Presburger arithmetic is doubly exponential, it follows that L_P is not real-time counter monitorable. This is because over the prefix a^n, computations are linear-time relative to n (exponential-time relative to the length of n). The language L_P can be monitored by reading n over the prefix a^n and deciding the validity of formula number n over a suffix b^m for m exponential in n (doubly-exponential time in the length of n).

The results of this subsection are summarized in Figure 6.

4.2 Trading Rate for Locations

By Theorem 4.3 the number of registers in a counter monitor cannot be reduced through increasing its rate or its number of locations. On the contrary, for fixed number of registers, the rate can always be reduced at the expense of increasing the number of locations. This can be shown as an immediate application of the "counter compression" idea of [14] that we recall for the sake of completeness.

Theorem 4.4. *Every k-counter monitor with m locations and rate $c > 1$ has a time-equivalent k-counter monitor with mc^k locations and rate 1.*

Proof. Let $\mathcal{A} = (\Sigma, X, Q, s, \Delta)$ be a counter monitor with $X = \{x_1, \ldots, x_k\}$ and rate c. We construct a time-equivalent counter monitor $B = (\Sigma, X', Q', s', \Delta')$ with rate 1 by delaying the updates and remembering them using the finite state control. Let $X' = \{y_1, \ldots, y_k\}$, and $Q' = Q \times \{0, \ldots, c-1\}^k$ with initial state $s' = (s, 0, \ldots, 0)$. During a run, we store in y_i the integer division of x_i by c, and use the additional component $\{0, \ldots, c-1\}^k$ in locations of Q' to store the remainders. Formally, we maintain between each pairs of configuration of $\mathcal{A}$ and $\mathcal{A}'$ the invariant $x_i = cy_i + d_i$ where d_i is the $(i+1)$st component of the location, for all registers $i \in \{1, \ldots, k\}$. We proceed to construct the set of edges Δ'. For each original edge $(q, \sigma, \phi_x, \gamma, q') \in \Delta$ and values of $d_1, \ldots, d_k$, we add to Δ' an edge from $(q, d_1, \ldots, d_k)$ to $(q', d'_1, \ldots, d'_k)$ labeled σ with test ϕ_y and update as follows. For $a < c - d_j$, an update $x_i \leftarrow x_j + a$ translates as $y_i \leftarrow y_j$. For $a \geq c - d_j$, it translates as $y_i \leftarrow y_j + 1$. In both cases we let $r'_i \equiv r'_j + a \mod c$. The test ϕ_y is obtained from ϕ_x by replacing every atomic formula $x_i + a = x_j + b$ by $y_i = y_j$ if $d_i + a = d_j + b$, by $y_i = y_j + 1$ if $d_i + a = d_j + b + c$, by $y_i + 1 = y_j$ if $d_i + a + c = d_j + b$, and by $\bot$ otherwise. □

Since k locations can be emulated by a single register with rate k, we have that in a counter monitor, locations and rate are interchangeable resources.

4.3 Counter Variants

We show in the following that copying can be seen as some form of reset. Let us first observe that inequality tests do not increase the expressive power of counter monitors. Without loss of generality, assume a rate of 1. The changes of truth status of an inequality will always be preceded by an equality becoming true and/or an increment of one of its variables. Some additional state component thus suffices to track the current truth value of the inequality.

Theorem 4.5. *Register monitors with instruction set $\langle +1, = \rangle$ and $\langle +1, \geq \rangle$ are equally expressive.*

As for counter machines, we also obtain equivalent definitions by considering registers that are incremented, decremented, and tested for zero. Every register x in a monitor with instruction set $\langle -1, +1, =0 \rangle$ can be counted as the difference of a positive part x^+ and a negative part x^-, and has value zero when $x^- = x^+$. Conversely the difference $x - y$ between every pair of registers x and y in a monitor with instruction set $\langle +1, = \rangle$ can be counted in some register r_{x-y}, such that $r_{x-y} = 0$ iff $x = y$. Thus:

Theorem 4.6. *Register monitors with instruction set $\langle +1, = \rangle$ and $\langle -1, +1, =0 \rangle$ are equally expressive.*

The above simulations are real-time, and in the case of Theorem 4.6 preserve the rate and number of locations.

A significant difference between our model and standard counter automata is the ability to duplicate registers. A *copy* is an update γ featuring a variable $y \in X$ that appears more than once in $\gamma(X)$. In other words, a copy update features a variable which occurs on the right-hand side of more than one assignment. Let us give

an example of counter monitor that makes essential use of a copy update.

Example 4.7. Let Σ' be the alphabet of events obtained from $\Sigma = \{a, b\}$ by letting $\Sigma' = \Sigma \cup \{\#\}$. Consider the following language:

$$L_1' = L_1 \cup \left\{ w \in \Sigma'^{\omega} \mid \forall u \in \Sigma'^*, \forall v \in \Sigma^*, u\#v \prec w \Rightarrow |v|_a \geq |v|_b \right\}$$

where L_1 is the language of Example 4.2. It describes words such that every occurrence of event b is matched by an earlier occurrence of event a after the last event #. A simple modification of the automaton of Figure 5 enables to recognize the language L_1'. For this we simply add two edges labeled $\#, x \leftarrow y$ going back to the initial state.

Using the terminology of [3], we call *copyless* a register monitor without copy updates. We now show that removing the ability to copy results in a loss of expressiveness.

Suppose, towards a contradiction, the existence of a copyless counter monitor $\mathcal{A}$ with k registers, m locations and such that $L(\mathcal{A}) = L_1'$. It is easy to check that the copyless quality of a monitor is preserved by the translation of Theorem 4.6. Thus we assume without loss of generality that $\mathcal{A}$ has instruction set $\langle -1, +1, = 0 \rangle$. We further assume with Theorem 4.4 that $\mathcal{A}$ has rate 1.

Lemma 4.8. *For any $d \in \mathbb{N}$ and initial configuration (q_0, v_0) there exists an integer $n \leq (2d+1)^k + 1$ such that $\mathcal{A}$ has a run from (q_0, v_0) on a^n ending in a configuration (q_n, v_n) which satisfies $|v(x)| > d$ for at least one register $x \in X$.*

Using the above lemma, we show that there is no copyless counter monitor to recognize L_1' by constructing a prefix that is misclassified by $\mathcal{A}$ as described above. It will follow that:

Theorem 4.9. *Copyless counter monitors are strictly less expressive than counter monitors.*

Proof. We assume the existence of a register monitor $\mathcal{A}$ with k registers and m locations that recognizes L_1' and satisfying the assumptions above. For technical convenience we also assume $m \geq 2$. Informally, we say that a register $x \in X$ is *inactive* if its absolute value exceeds a bound such that it cannot be restored back to 0 for a certain class of inputs. Note that when $\mathcal{A}$ has one of its k registers inactivated, it is equivalent to a monitor with $k-1$ registers over that class of inputs.

The idea is to construct a prefix u in such a form that the run of $\mathcal{A}$ over u results in a configuration where all registers hold a value above $2m$. Then $\mathcal{A}$ cannot process all continuations of u correctly, since it can distinguish inequivalent prefixes of length $0 \leq l \leq 2m$ over the alphabet $\{a, \#\}$ by using its finite state memory consisting of m locations while blindly updating its counters.

The string u will be of the form $u_k\#u_{k-1}\# \dots u_2\#u_1$ where $u_i = a^{n_i}$. We choose each n_i such that after reading $u_k\# \dots u_{k-i-1}\#$ at least i registers are inactive. For this we rely on Lemma 4.8. Each upper bound $2m(2d_i+1)^{k_i}+1$ for inactivity depends on the number k_i of registers and the length d_i of the remaining part of the prefix. Therefore, we choose $d_1 = 2m+1$ as previously indicated, and construct u from right to left.

Considering u_1 we have $d_1 = 2m+1$ and $k_1 = 1$. By Lemma 4.8 we can choose n_1 such that $n_1 \leq 2m(2d_1+1)^{k_1}+1 = m_1$. We ensure that the single active register becomes inactive after reading u_1 by choosing the appropriate $n_1 \leq m_1$. Then, for u_2 we require $d_2 = m_1 + d_1 + 1$, $k_2 = 2$, and $n_2 \leq m_2 = 2m(2d_2+1)^{k_2}+1$. By induction on i we can obtain $d_i = m_{i-1} + d_{i-1} + 1$ and $n_i \leq m_i = 2m(2d_i+1)^i + 1$ satisfying the desired assumptions. At the ith separator $\mathcal{A}$ behaves as a $(k-i)$-counter monitor $\mathcal{A}_i$. Applying Lemma 4.8 on $\mathcal{A}_i$ for the next separator we obtain that another register becomes inactive. This gives us the sequence u.

We now consider suffixes of the form uw where w is a finite word in the language $F = \bigcup_{l=0}^{2m} (a \cup \#)^l$. There exists a pair of special words in F of the form $w = w_1\#w_2\# \dots \#w_l$ and $w' = w_1'\#w_2'\# \dots \#w_l'$ with $w_j, w_j' \in a^*$ for $1 \leq j \leq l$ satisfying the following condition: there exists $1 \leq i < l$ and a non-empty word v such that (1) $w_i = w_i'v$ and $w_l' = w_lv$; (2) the monitor $\mathcal{A}$ is in the same location before and after reading v at such positions in uw and uw'; (3) for all $1 \leq j < l$ with $j \neq i$ we have $w_j = w_j'$. Then since counter values are larger than $2m$ after reading u, the runs of $\mathcal{A}$ over uw and uw' are the same except for one extra loop over v at positions i and l, respectively. Additive counter updates are commutative, so that $\mathcal{A}$ is in the same configuration after reading uw and uw'. But $u\#wb^p\#a^{\omega} \in L_1'$ while $u\#w'b^p\#a^{\omega} \notin L_1'$ for $p = |w_l|$, so that $\mathcal{A}'$ does not recognize L_1', a contradiction. □

Definition 4.10 (Reset). We call *reset-counter monitor* a register monitor with the instruction set $\langle 0, +1, = \rangle$.

The reset operation can replace the ability to copy without any loss of expressive power. To show this, we proceed by translating a counter monitor in two steps using a simple variant of Theorem 4.6. The first step applies the translation and yields a copyless monitor with instruction set $\langle 0, +1, -1, = 0 \rangle$. For this, it suffices to notice that duplicating a value in unsigned counters with $x \leftarrow z, y \leftarrow z$ has the effect of a reseting to zero in signed counters, emulated with $r_{x-y} \leftarrow 0$. The second step applies the reverse translation and yields a copyless monitor with instruction set $\langle 0, +1, = \rangle$.

Theorem 4.11. *For any counter monitor, there exists a time-equivalent copyless reset-counter monitor.*

5 ADDER MONITORS

We now enhance counter monitors with the ability to increment the content of one register by the content of another.

Definition 5.1 (Adder Monitors). A register monitor with the instruction set $\langle 1, +, = \rangle$ is called an *adder monitor*.

5.1 Expressiveness

The ability to compute sums of registers gives adder monitors dramatically more expressive power, and notably the ability to encode the prefixes of a word in real time.

Example 5.2. Let Σ' be an alphabet of events defined by letting $\Sigma' = \Sigma \cup \{\#\}$, where $\Sigma = \{a, b\}$ as previously. Consider the safety language $M_1 = \bigcup_{w \in \Sigma^*} \#(w\#)^{\omega} \cup \#(w\#)^{\omega}\Sigma^{\omega}$. It consists of infinite sequences starting with # in which one unique finite word over Σ repeats, each consecutive pair of occurrences separated by #. The language M_1 is recognized by the real-time adder monitor with 2 registers of Figure 7. The part of a word before a separator, if any, is encoded in x using a binary representation. Later occurrences of

finite words are encoded in y. At every separator the encoding of the two words must match.

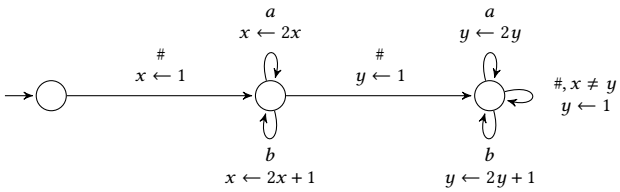

Figure 7: An adder monitor recognizing language M_1.

It is easy to see that the language M_1 requires registers growing at least exponentially fast in value with the length of the word. Since counter monitors have registers only growing linearly (and their product polynomially) fast, we have:

Theorem 5.3. *Adder monitors are more expressive than counter monitors.*

This strict separation between adders and counters holds for both real-time monitors, and for monitors that do not halt in real time. The extent of the expressiveness of real-time adder monitors is unknown. In particular, we do not know if a hierarchy, analogous to the one of Section 4 exists. Consider the following set of languages.

Example 5.4. Let $\Sigma' = \Sigma \cup \{\#\}$, where $\Sigma = \{a, b\}$ as previously. Events a and b now represent bit values and event # serves as a delimiter. We define the safety language

$$M_k = \bigcup_{w_1,\ldots,w_k \in \Sigma^*} \#(w_1\# \cup \ldots \cup w_k\#)^\omega \cup \#(w_1\# \cup \ldots \cup w_k\#)^*\Sigma^\omega$$

consisting of sequences of words between separators, in which at most k unique words repeat. This could model some communication channel in which processes periodically broadcast some unique identifier, and the safety property being that at most k processes can use the channel.

Example 5.4 seem to indicate that increasing the number of registers in real-time adder monitors always leads to a gain in expressive power. For general adder monitors, not required to halt in real-time, that cannot be the case. We show on the contrary that the hierarchy collapses at $k = 6$.

Theorem 5.5. *Adder monitors with* 6 *registers can recognize all computable safety languages.*

Proof. Assume an alphabet $\Sigma = \{a, b\}$, larger alphabets can easily be accommodated. We use 6 registers r, s, t, x, y, z and simulate a deterministic Turing machine whose tape initially contains the input word. This resembles the standard simulation of Turing machines by counter machines [36], however with the additional technical complications that the input must be stored for delayed processing and that counters cannot be decremented. Register t will store the pending input in binary, register s is used as a cursor to write t, registers x and y respectively store the parts of the tape at the left and right of the read/write head, and registers r and z are used as temporary variables to perform divisions.

At every new input event the cursor, initialized to 1, is moved to the right by letting $s \leftarrow 2s$. Register t does not need updating for event a and is updated by $t \leftarrow t + s$ for event b. The symbol currently under the read/write head is stored in finite memory. Reading and writing on the tape is simulated asynchronously (at a slower rate) as follows.

To simulate a move of the head to the left, we first write the symbol under the read/write head as least significant bit in y by $y \leftarrow 2y$ for symbol a and $y \leftarrow 2y + 1$ for symbol b. The encoding of the pending part of the tape is updated by $t \leftarrow 2t$ and $s \leftarrow 2s$. We then read the least significant bit in x. For this, we use register r and z to divide x by 2 by through the updates $z \leftarrow z + 1, r \leftarrow r + 2$ until either $r = x$ (remainder is 0) or $r = x - 1$ (remainder is 1). The remainder is stored in finite memory as the new symbol under the read/write head, and x is updated by $x \leftarrow z$.

To simulate a move of the head to the right, we proceed symmetrically but only after having updated y with the pending part of the tape stored in t. For this, we use $y \leftarrow y + t$, $t \leftarrow 0$ and cancel the update $s \leftarrow 2s$ at the next input event.

While moving the head by one position may cost an arbitrary amount of registers operations, every simulated move will always terminate in finite time. □

5.2 Adders versus Counters in Real Time

In real time, such a completeness result cannot hold due to the existence of problems with time complexity more than exponential. It is likely that every register in adder monitor increases their expressive power. We found no proof of this, but obtained the following separation result: in real time, 2 adders are not sufficient to simulate 3 counters.

Example 5.6. We let $\Sigma = \{a, \#\}$, and consider the safety language

$$N_k = \bigcup_{n_1,\ldots,n_k \in \mathbb{N}} \#(a^{n_1}\# \cup \ldots \cup a^{n_k}\#)^\omega \cup \#(a^{n_1}\# \cup \ldots \cup a^{n_k}\#)^* a^\omega$$

consisting of sequences of strings of a, featuring at most k unique lengths. This language is similar to the language M_k of Example 5.4 but where finite words are over a unary alphabet.

Theorem 5.7. *The language N_2 can be recognized in real time by a counter monitor with* 3 *registers but not by an adder monitor with* 2 *registers.*

For the easy part of the theorem, we can construct a counter monitor with registers $\{x, y, z\}$ recognizing N_2 as follows. We count n_1 in x if it exists, $n_2 \neq n_1$ in y if it exists, and any other sequence of a's in z. If upon occurrence of a separator, $z = x$ or $z = y$ then reject.

For the hard part of the theorem, we let $\mathcal{A}$ be a *real-time* adder monitor over the alphabet $\Sigma = \{a, \#\}$ with 2 registers $\{x, y\}$, rate c, and m control locations. We will use the following lemma. Given a finite word w let us denote by $x(w)$ and $y(w)$ the values of x and y in $\mathcal{A}$ after reading w, respectively.

Lemma 5.8. *Let n be a positive integer and u, u' be words such that $x(u) = y(u) > 2^{cn}$ and $x(u') = y(u') > 2^{cn}$, and $\mathcal{A}$ is in the same location after reading u or u'. For any word w with $|w| \leq n$ we have uw is bad prefix for $L(\mathcal{A})$ iff $u'w$ is also bad prefix for $L(\mathcal{A})$.*

Proof. Assume $x(u) = y(u) = k$ for some $k > 2^{cn}$ and word u. Let us denote by q_i the location of monitor $\mathcal{A}$ after reading u and the first i letters of w. The value of registers x and y after reading i letters of w starting with initial value k can be respectively decomposed into $e_i k + f_i$ and $g_i k + h_i$ for positive integers e_i, g_i and $0 \leq f_i < 2^{ci}$, $0 \leq h_i < 2^{ci}$. This can easily be shown by induction on i. Every update of the form $x \leftarrow x + 1$ adds 1 to f_i and every update of the form $x \leftarrow x + y$ adds g_i to e_i and h_i to f_i, similarly for other forms of updates. In particular $\max\{g_i, h_i\}$ is multiplied at most by 2 per update, and assuming rate c, at most by 2^c per transition.

Assume further $x(u') = y(u') = k'$ for some $k' > 2^{cn}$ and word u' such that $\mathcal{A}$ is in location q_0 after reading u'. We write e'_i, f'_i, g'_i, h'_i the coefficients that occur in the decomposition of x and y after reading i letters of w as previously, starting with alternative register value $x = y = k'$. Let us also write q'_i the locations of $\mathcal{A}$ after reading i letters of w, starting with register values k'. We show by induction on i that $e'_i = e_i$, $f_i = f'_i$, $g_i = g'_i$, $h_i = h'_i$, and $q_i = q'_i$. For $i = 0$ the initial values of each coefficients are the same. For $i \geq 1$, we demonstrated that $|f_i| < k$ and $|f'_i| < k'$, which gives us $e_{i-1}k + f_{i-1} = g_{i-1}k + h_{i-1}$ iff $e_{i-1} = g_{i-1}$ and $f_{i-1} = h_{i-1}$. In turn this holds iff $e_{i-1}k' + f_{i-1} = g_{i-1}k' + h_{i-1}$, and iff $e'_{i-1}k' + f'_{i-1} = g'_{i-1}k' + h'_{i-1}$ by induction hypothesis. Therefore any test $ax + by + d = 0$ with coefficients a, b, d at most 2^c in magnitude passes or fails identically starting from k or from k'. By induction hypothesis we also have $q_{i-1} = q'_{i-1}$ so that the same updates are applied, yielding equal coefficients $e_i = e'_i, \ldots, h_i = h'_i$ and equal target locations $q_i = q'_i$.

We obtain that automaton $\mathcal{A}$ is in the same location after reading uw and $u'w$. Thus $\mathcal{A}$ errs on both words, or on neither. Since $\mathcal{A}$ operates in real time, we have that uw is a bad prefix for $L(\mathcal{A})$ iff $u'w$ is also a bad prefix for $L(\mathcal{A})$. □

Assume towards a contradiction that $\mathcal{A}$ recognizes N_2. In addition to the lemma above, we also make use of the following facts.

Claim 1. *For any $l \geq 0$, there are at most ml^2 words $w = \#u_1\#u_2$ such that $\max\{x(w), y(w)\} < l$.*

Claim 2. *For any word $w = \#u\#v\#u \in L(\mathcal{A})$ with $u, v \in a^*$ such that $|u| > m+1$, there exists u', u'' such that $u'u'' = u$, $|u''| \leq m+1$, and $x(\#u\#v\#u') = y(\#u\#v\#u')$.*

We now have all the ingredients to proceed.

Proof of Theorem 5.7. Let us fix some positive integer n such that $n > 4(m+1)^2$ and $n^2 > 2c(n+m+2)$. We call *unbalanced* any word of the form $w = \#u\#v$ such that $3 \cdot 4^{n^2} < |u| \leq 4^{n^2+1}$ and $n < |v| \leq 2n$, and $\max\{x(w), y(w)\} \geq 2^{n^2}$. There are 4^{n^2} possible words $u \in a^*$ such that $3 \cdot 4^{n^2} < |u| \leq 4^{n^2+1}$, and n possible words $v \in a^*$ with $n < |v| \leq 2n$. Following Claim 1, there are at least $n4^{n^2} - 2^{n^2} > n4^{n^2-1}$ unbalanced words. Thus there exists a special prefix u_0 for which there are at least $\frac{n}{4}$ distinct words v such that $\#u_0\#v$ is unbalanced. Following Claim 2, since $|u_0| > m+1$ for every such v there exist u', u'' such that $u_0 = u'u''$ with $|u''| \leq m+1$, and $x(\#u_0\#v\#u') = y(\#u_0\#v\#u')$. There are at least $\frac{n}{4} > (m+1)^2$ words v such that $\#u_0\#v$ is unbalanced, and thus there exist two words $v_1 \neq v_2$ associated with the same factorization $u'u''$ of u_0, and such that $\mathcal{A}$ is in the same location after reading either $w_1 = \#u_0\#v_1\#u'$ or $w_2 = \#u_0\#v_2\#u'$. But then by Lemma 5.8 since $|\#u''\#v_1| \leq 2n+m+2$ and $x(w_1) = y(w_1)$, $x(w_2) = y(w_2)$ are greater than or equal to $2^{n^2} > 2^{2c(n+m+2)}$ we have that $w_1u''\#v_1\#$ is a bad prefix for $L(\mathcal{A})$ iff $w_2u''\#v_1\#$ is also a bad prefix for $L(\mathcal{A})$. By hypothesis $\mathcal{A}$ halts in real-time and $w_2u''\#v_1\#$ is a bad prefix for N_2 while $w_1u''\#v_1$ is not. Thus $L(\mathcal{A}) \neq N_2$. □

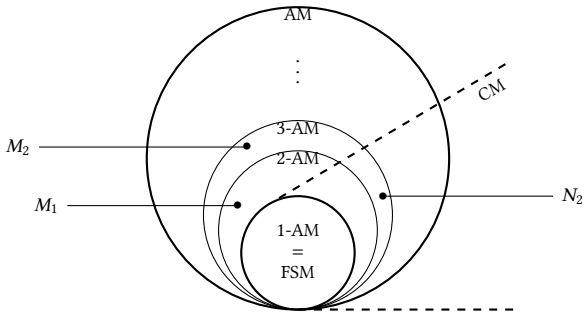

Figure 8: Separation of real-time adders. FSM, CM, and AM stand for finite-state monitor, counter monitor, and real-time adder monitor respectively.

One can easily check that the proof of Theorem 5.7 carries over to the language M_2. Thus the inclusion of real-time 2-adder languages in real-time 3-adder languages is also proper in the subset of languages not counter recognizable. A summary of the results of the last two subsections appears in Figure 8.

5.3 Adder Variants

We observe that providing adder monitors with a subtraction operation and replacing an equality test by a test for zero does not affect their expressive power.

Theorem 5.9. *Register monitors with instruction set $\langle 1, +, = \rangle$ and with instruction set $\langle 1, +, -, =0 \rangle$ are equally expressive.*

Moreover, simulations underlying the above theorem are real-time.

Let us now investigate the effect of copy updates in adders. Take X to be a set of registers. Following [3], an update μ over X is said to be *copyless* if every variable y appears at most once in $\sum_{x \in X} \mu(x)$. An adder monitor is copyless if every one of its updates is.

Surprisingly we show that in copyless monitors, adders are not more expressive than counters. Here we assume that adder monitors are equipped with a *reset* operation, that is, the instruction set we consider is $\langle 0, 1, +, = \rangle$, otherwise the statement is trivial.

Theorem 5.10. *Any copyless reset-adder monitor with k registers can be simulated in real time by a counter monitor with 2^k registers.*

Proof. Let $\mathcal{A}$ be an arbitrary copyless reset-adder monitor, and let X be its set of registers. We construct an equivalent counter monitor $\mathcal{A}'$ as follows. Registers of $\mathcal{A}'$ are taken in the set $R = \{r_Y \mid Y \subseteq X\}$. By definition, the sum of any subset of the right-hand sides of a copyless update does not feature duplicated variables either. Therefore we can maintain each variable r_Y of $\mathcal{A}'$, storing $\sum_{y \in Y} r_y$, by using updates of the form $r_Y \leftarrow r_Z + a$. Tests in $\mathcal{A}'$ are obtained form those in $\mathcal{A}$ by replacing every $x \in X$ by r_x. □

6 BEYOND ADDERS

We now consider general arithmetic register instructions.

Definition 6.1. We call *linear* a register monitor with the instruction set $\langle 0, 1, +, -, \geq \rangle$ and we call *polynomial* a register monitor with the instruction set $\langle 0, 1, +, -, \times, \geq \rangle$.

The expressive power of real-time linear register monitors is related to that of real-time, multi-tape Turing machines. A formal definition of this model is given in [35]. The safety ω-language of a Turing machine is the set of ω-words on which it does not halt. A Turing machine is said to be *real-time* when it errs on the set of bad prefixes for its safety language. It is easy to see that 2 linear registers can emulate a push-down store, using comparisons $x \geq y$ to probe the most significant bit of a register x and $x \leftarrow x + y$, $x \leftarrow x - y$ to write it when y stores 2^n and x has length n. Dividing y by 2 is equivalent to multiplying x by 2. Thus:

Theorem 6.2. *Linear register monitors with $4k$ registers can simulate Turing machines with k work tapes in real time for all $k \geq 0$.*

Since Turing machines with one work tape can simulate any number of counters in real time [37], we have:

Corollary 6.3. *Any counter monitor can be simulated by a linear register monitor with 4 registers in real time.*

In spite of their ability to multiply, polynomial register monitors still have limited expressive power in real time.

Example 6.4. Let $\Sigma = \{a, b\}$, and $\Sigma' = \Sigma \cup \{\#\}$. We consider the following language H, defined as

$$H = \#\Sigma'^{\omega} \setminus \bigcup_{w \in \Sigma^*} (\#\Sigma^*)^* \#w(\#\Sigma^*)^* \#w\#\Sigma'^{\omega}.$$

When finite words over Σ represent numbers in binary notation and # separates words into numbers, language H represents sequences in which no number repeats.

We obtain the following result, analogous to the one of [20] for the model of real-time Turing machines.

Theorem 6.5. *The language H cannot be monitored by real-time polynomial register monitors.*

We will use the following characterization of the number of cells in semialgebraic decompositions due to [32]:

Lemma 6.6. *Given a set of polynomials $P = \{p_1, \ldots, p_s\}$ in variables $x_1, \ldots, x_k$ where each polynomial has degree at most d, the number of cells in the partition of $\mathbb{R}^k$ by P is $O((sd/k)^k)$.*

Proof of Theorem 6.5. Assume towards a contradiction the existence of a real-time polynomial register monitor $\mathcal{A}$ such that $L(\mathcal{A}) = H$. Let m, k, c respectively stand for the number of states, number of registers, and rate of $\mathcal{A}$. We examine configurations of $\mathcal{A}$ after reading a prefix of the form $\#u\#w\#$ where $u \in \Sigma'^*$ and $w \in \Sigma^n$ for fixed length n. Since the prefix $\#u$ is arbitrarily long, it can feature any subset of words of length n. While processing $w\#$, the monitor must discriminate between 2^{2^n} subsets of words of length n. During this subword, $\mathcal{A}$ performs $n + 1$ updates and tests. This is equivalent to testing the values of registers for inequalities with terms of size $c(n + 2)$. These terms are polynomials with k variables of degree cn. Let us denote by P this family of polynomials. We have $|P| < (k + 4)^{c(n+2)}$ by enumeration of their possible syntactic trees. Tests $p \geq 0$ for $p \in P$ form a partition over $\mathbb{Z}^k$ into finitely many cells, in which the sign of every polynomial in P is constant. By application of Lemma 6.6 we obtain that the number of nonempty cells defined by P is $O(((k+4)^{c(n+2)} c(n+2)/k)^k)$. After reading a prefix $\#u\#$, monitor $\mathcal{A}$ is in one of m locations. Throughout the suffix $w\#$, possible register valuations v and v' cannot be distinguished when they lie in the same cell. Hence for large n the numbers of inequivalent configurations of $\mathcal{A}$ is $2^{O(n \log n)}$. This does not suffice to discriminate between 2^{2^n} equivalence classes of prefixes of H. Thus $\mathcal{A}$ does not recognize bad prefixes for H. □

We remark that however if numbers are presented in unary, then the corresponding language can be recognized in real time.

Example 6.7. Let $\Sigma' = \{a, \#\}$. We now consider language

$$K = \#\Sigma'^{\omega} \setminus \bigcup_{n \in \mathbb{N}} (\#a^*)^* \#a^n (\#a^*)^* \#a^n \#\Sigma'^{\omega}$$

of words in which no number, given in unary notation, repeats.

Theorem 6.8. *The language K can be recognized by a linear register monitor in real time.*

Proof. We encode a set of numbers $\{n_1, \ldots, n_k\}$ into the value $2^{n_1} + \ldots + 2^{n_k}$. It is straightforward to maintain this encoding in real-time using a single-tape Turing machine with a special instruction making the read/write head jump back to the first position. A minor adaptation of Theorem 6.2 give us that the language K is also recognizable by a linear register monitor in real time. □

We do not know whether $k + 1$ registers are more powerful than k registers in real-time polynomial monitors. An instruction set in which the hierarchy collapses is $\langle 0, 1, +, \times, e, f, \geq \rangle$ where e and f are binary functions such that $e(i, j) = i^j$, and $f(i, j)$ is the multiplicity of factor j in the prime decomposition of i. With this instruction set, any monitor with k registers can be simulated in real time by a monitor with 1 register.

7 CONCLUSION

We propose *register monitors* as a computational model for the run-time monitoring of reactive systems. The basic monitoring problem asks for recognizing a safety ω-language in real time. While previous approaches put emphasis on the subclass of ω-regular languages, we see no reason to restrict monitors, which are usually implemented in software, to finite-state. Looking beyond finite-state, we uncovered an expressiveness hierarchy for register monitors depending on the number of available registers and arithmetic capabilities. This hierarchy is significantly more nuanced than the computability hierarchy, which does not restrict register machines to a single pass over an input word, nor to a bounded number of steps between consecutive inputs.

There are several directions in which this work needs to be extended. First, while our monitors can only reject an input word, *quantitative monitors* may output values, such as the maximal or average response time seen so far [5]. Quantitative monitors have several advantages: they can be used to over- or underapproximate the desired quantity, and thus, they can be compared according to a

resource-precision trade-off. Similarly to the qualitative monitoring we studied, we expect quantitative and approximate monitoring with arithmetic registers to exhibit a rich theory beyond the finite-state/ω-regular case.

Second, monitors for the same language can be compared as to how "quickly" they reject a violating input word. There is a trade-off between the resources (registers and operations) available to a monitor and its efficiency (delayed rejection), which is closely related to the time and space requirements of on-line computation.

Third, we left several interesting problems open, perhaps most notably the question whether adder monitors exhibit the same strict register hierarchy as counter monitors; we were able to show that 3 adders are more powerful than 2 adders, but the general problem is still open in the real-time case. In particular, the information-theoretic arguments that have been used for real-time Turing machines do not directly apply.

Fourth, a logical next step beyond monitoring is *enforcement*. In enforcement [12] (or "shielding" [23]), the monitor can, in real time, make changes to the observed input sequence in order to repair property violations. Once again, the topic of finite-state enforcement has received much attention, but to the best of our knowledge the power of enforcement with registers has not yet been studied.

ACKNOWLEDGMENTS

We thank Bernhard Kragl for early discussions. This research was supported in part by the Austrian Science Fund (FWF) under grants S11402-N23 (RiSE/SHiNE) and Z211-N23 (Wittgenstein Award).

REFERENCES

[1] Stål O Aanderaa. 1973. *On k-tape versus (k-1)-tape real time computation.* IBM Thomas J. Watson Research Division.

[2] Bowen Alpern and Fred B Schneider. 1985. Defining liveness. *Information processing letters* 21, 4 (1985), 181–185.

[3] Rajeev Alur, Loris D'Antoni, Jyotirmoy Deshmukh, Mukund Raghothaman, and Yifei Yuan. 2013. Regular Functions and Cost Register Automata. In *Symposium on Logic in Computer Science (LICS '13)*. IEEE Computer Society, Washington, DC, USA, 13–22.

[4] Rajeev Alur, Adam Freilich, and Mukund Raghothaman. 2014. Regular Combinators for String Transformations. In *Computer Science Logic and Symposium on Logic in Computer Science (CSL-LICS '14)*. ACM, New York, NY, USA, Article 9, 10 pages. https://doi.org/10.1145/2603088.2603151

[5] Krishnendu Chatterjee, Thomas A. Henzinger, and Jan Otop. 2016. Quantitative Monitor Automata. In *International Symposium on Static Analysis*. 23–38. https://doi.org/10.1007/978-3-662-53413-7_2

[6] Yu-Fang Chen, Ondřej Lengál, Tony Tan, and Zhilin Wu. 2017. Register automata with linear arithmetic. In *Logic in Computer Science (LICS), 2017 32nd Annual ACM/IEEE Symposium on*. IEEE, 1–12.

[7] Marcelo d'Amorim and Grigore Roşu. 2005. Efficient Monitoring of ω-Languages. In *Computer Aided Verification (Lecture Notes in Computer Science)*, Vol. 3576. Springer, 364 – 378.

[8] Ben d'Angelo, Sriram Sankaranarayanan, César Sánchez, Will Robinson, Bernd Finkbeiner, Henny B Sipma, Sandeep Mehrotra, and Zohar Manna. 2005. LOLA: Runtime monitoring of synchronous systems. In *Temporal Representation and Reasoning, 2005. TIME 2005. 12th International Symposium on*. IEEE, 166–174.

[9] Stéphane Demri. 2006. Linear-time temporal logics with Presburger constraints: an overview. *Journal of Applied Non-Classical Logics* 16, 3-4 (2006), 311–347.

[10] Stéphane Demri and Ranko Lazić. 2009. LTL with the freeze quantifier and register automata. *ACM Transactions on Computational Logic* 10, 3 (2009), 16.

[11] Yliès Falcone, Jean-Claude Fernandez, and Laurent Mounier. 2009. Runtime verification of safety-progress properties. In *International Workshop on Runtime Verification*. Springer, 40–59.

[12] Yliès Falcone, Laurent Mounier, Jean-Claude Fernandez, and Jean-Luc Richier. 2011. Runtime enforcement monitors: composition, synthesis, and enforcement abilities. *Formal Methods in System Design* 38, 3 (2011), 223–262. https://doi.org/10.1007/s10703-011-0114-4

[13] Alain Finkel, Stefan Göller, and Christoph Haase. 2013. Reachability in register machines with polynomial updates. In *International Symposium on Mathematical Foundations of Computer Science*. Springer, 409–420.

[14] Patrick C. Fischer, Albert R. Meyer, and Arnold L. Rosenberg. 1968. Counter machines and counter languages. *Mathematical systems theory* 2, 3 (1968), 265–283.

[15] Patrick C. Fischer, Albert R. Meyer, and Arnold L. Rosenberg. 1970. Time-restricted sequence generation. *J. Comput. System Sci.* 4, 1 (1970), 50–73.

[16] MCW Geilen. 2001. On the construction of monitors for temporal logic properties. *Electronic Notes in Theoretical Computer Science* 55, 2 (2001), 181–199.

[17] Dimitra Giannakopoulou and Klaus Havelund. 2001. Automata-based verification of temporal properties on running programs. In *Automated Software Engineering*. IEEE, 412–416.

[18] Radu Grigore, Dino Distefano, Rasmus Lerchedahl Petersen, and Nikos Tzevelekos. 2013. Runtime verification based on register automata. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 260–276.

[19] Orna Grumberg, Orna Kupferman, and Sarai Sheinvald. 2010. Variable automata over infinite alphabets. In *International Conference on Language and Automata Theory and Applications*. Springer, 561–572.

[20] J. Hartmanis and R. E. Stearns. 1965. On the Computational Complexity of Algorithms. *Trans. Amer. Math. Soc.* 117 (1965), 285–306. http://www.jstor.org/stable/1994208

[21] Klaus Havelund, Giles Reger, Daniel Thoma, and Eugen Zălinescu. 2018. Monitoring events that carry data. In *Lectures on Runtime Verification*. Springer, 61–102.

[22] Michael Kaminski and Nissim Francez. 1994. Finite-memory automata. *Theoretical Computer Science* 134, 2 (1994), 329–363.

[23] Bettina Könighofer, Mohammed Alshiekh, Roderick Bloem, Laura Humphrey, Robert Könighofer, Ufuk Topcu, and Chao Wang. 2017. Shield synthesis. *Formal Methods in System Design* 51, 2 (2017), 332–361. https://doi.org/10.1007/s10703-017-0276-9

[24] Orna Kupferman and Moshe Y Vardi. 2001. Model checking of safety properties. *Formal Methods in System Design* 19, 3 (2001), 291–314.

[25] Insup Lee, Sampath Kannan, Moonjoo Kim, Oleg Sokolsky, and Mahesh Viswanathan. 1999. Runtime assurance based on formal specifications. *Departmental Papers (CIS)* (1999), 294.

[26] Martin Leucker and Christian Schallhart. 2009. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming* 78, 5 (2009), 293–303.

[27] Zohar Manna and Amir Pnueli. 2012. *Temporal verification of reactive systems: safety*. Springer Science & Business Media.

[28] Li Ming and Paul MB Vitányi. 1990. Kolmogorov complexity and its applications. In *Algorithms and Complexity*. Elsevier, 187–254.

[29] Marvin L. Minsky. 1961. Recursive Unsolvability of Post's Problem of "Tag" and other Topics in Theory of Turing Machines. *Annals of Mathematics* 74, 3 (1961), 437–455. http://www.jstor.org/stable/1970290

[30] Marvin L Minsky. 1967. *Computation: finite and infinite machines*. Prentice-Hall.

[31] Wolfgang J. Paul, Joel I. Seiferas, and Janos Simon. 1981. An information-theoretic approach to time bounds for on-line computation. *J. Comput. System Sci.* 23, 2 (1981), 108 – 126. https://doi.org/10.1016/0022-0000(81)90009-X

[32] Richard Pollack and M-F Roy. 1993. On the number of cells defined by a set of polynomials. *Comptes rendus de l'Académie des sciences. Série 1, Mathématique* 316, 6 (1993), 573–577.

[33] Michael O. Rabin. 1963. Real time computation. *Israel Journal of Mathematics* 1, 4 (01 Dec 1963), 203–211. https://doi.org/10.1007/BF02759719

[34] Giles Reger, Helena Cuenca Cruz, and David Rydeheard. 2015. MarQ: monitoring at runtime with QEA. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 596–610.

[35] Arnold L. Rosenberg. 1967. Real-Time Definable Languages. *J. ACM* 14, 4 (1967), 645–662.

[36] Rich Schroeppel. 1972. *A two counter machine cannot calculate* 2^N. Technical Report. MIT.

[37] Joel I Seiferas and Paul Vitányi. 1988. Counting is easy. *Journal of the ACM (JACM)* 35, 4 (1988), 985–1000.

[38] J. C. Shepherdson and H. E. Sturgis. 1963. Computability of Recursive Functions. *J. ACM* 10, 2 (April 1963), 217–255. https://doi.org/10.1145/321160.321170

[39] Wolfgang Thomas. 1997. Languages, automata, and logic. In *Handbook of formal languages*. Springer, 389–455.

[40] Moshe Y Vardi and Pierre Wolper. 1986. An automata-theoretic approach to automatic program verification. In *Proceedings of the First Symposium on Logic in Computer Science*. IEEE Computer Society, 322–331.

[41] H. Yamada. 1962. Real-Time Computation and Recursive Functions Not Real-Time Computable. *IRE Transactions on Electronic Computers* EC-11, 6 (Dec 1962), 753–760. https://doi.org/10.1109/TEC.1962.5219459

[42] Yifei Yuan, Dong Lin, Ankit Mishra, Sajal Marwaha, Rajeev Alur, and Boon Thau Loo. 2017. Quantitative Network Monitoring with NetQRE. In *Conference of the ACM Special Interest Group on Data Communication*. ACM, 99–112.

A APPENDIX: OMITTED PROOFS

In this appendix, we produce all proofs that were omitted in the body of the text. They appear here in the order with which the results were stated.

In Section 2

Proof of Theorem 2.2. Let $\mathcal{A}_1 = (\Sigma, X_1, Q_1, s_1, \Delta_1)$ and $\mathcal{A}_2 = (\Sigma, X_2, Q_2, s_2, \Delta_2)$ with disjoint sets of locations and of registers. We construct the product $\mathcal{A}_\cap = (\Sigma, X, Q_\cap, s, \Delta_\cap)$ of $\mathcal{A}_1$ by $\mathcal{A}_2$ as follows. Let $X = X_1 \cup X_2$, $Q_\cap = Q_1 \times Q_2$, and $s = (s_1, s_2)$. The set $\Delta_\cap$ consists of edges $((q_1, q_2), \sigma, \phi_1 \wedge \phi_2, \mu_1 \| \mu_2, (q_1', q_2'))$ for every pair of edges $(q_i, \sigma, \phi_i, \mu_i, q_i')$ in Δ_i, $i = 1, 2$. We construct the product $\mathcal{A}_\cup = (\Sigma, X, Q_\cup, s, \Delta_\cup)$ of $\mathcal{A}_1$ by $\mathcal{A}_2$ as follows. Let $X = X_1 \cup X_2$, $Q_\cup = Q_1 \cup Q_2 \cup Q_1 \times Q_2$, and $s = (s_1, s_2)$. The set $\Delta_\cup$ contains edges in Δ_1, Δ_2 and $\Delta_\cap$ as previously and those of the form $((q_1, q_2), \sigma, \phi_1 \wedge \neg\phi_2, \mu_1, q_1')$ and $((q_1, q_2), \sigma, \neg\phi_1 \wedge \phi_2, \mu_2, q_2')$. □

Proof of Theorem 2.5. Let $\mathcal{A}$ be an ϵ-monitor with delay c for some $c \in \mathbb{N}$. Any pair of edges in $\mathcal{A}$ of the form $(q, \epsilon, \phi, \gamma, r)$ and $(r, \sigma, \phi', \gamma', s)$ can be replaced by a single edge $(q, \sigma, \phi \wedge \phi'[\gamma], \gamma'[\gamma], s)$ where $\phi'[\gamma]$ and $\gamma'[\gamma]$ denote the test and update obtained from ϕ' and γ' by replacing variables in X with their image by γ. We apply this rule in order to construct all edges associated to some path $\epsilon^k\sigma$ for $k \leq c$. Any run containing transitions labeled ϵ can be replaced by a path containing transitions with labels in Σ. After removing edges with labels not in Σ, we obtain an equivalent monitor with rate $c + 1$.

In the other direction, we can split edges featuring terms or atomic formulas of size greater than 1 into pair of edges $(q, \epsilon, \phi, \gamma, r)$ and $(r, \sigma, \phi', \gamma', r)$ using a fresh location r. Any update featuring terms of size $n \geq 1$ can be decomposed into $\mu[\mu']$ where μ has terms of size at most 1, and μ' has terms of size at most $n - 1$. This is done by introducing auxiliary registers $x_1', \ldots, x_k'$. For any x_i such that $\mu(x_i)$ is of the form $f(t)$ for function f and term t, we let $\mu(x_i) = f(x_i')$ and $\mu'(x_i') = t$. We proceed similarly for tests. This process is iterated until no new edges are found, and will terminate since at every step the size of the terms is reduced by one. Removing edges with terms or atomic formulas of size greater than 1, we obtain the desired ϵ-monitor. □

In Section 4

Proof of Theorem 4.5. Assume without loss of generality that all register monitors have rate $c = 1$ thanks to Theorem 4.4. We show that one test can simulate the other by either using an expression that is logically equivalent, or using finite state memory to remember how registers relate to each other. For the easy part it suffices to observe that $x{=}y$ iff $x \geq y \wedge y \geq x$.

For the other direction we observe the behaviors of registers and store the information which is not provided by the test = using additional control locations. More specifically, for every distinct pair of registers we use a 3-valued label associated with locations, denoting how the corresponding pair of registers is related. Let $S = \{-1, 0, 1\}$ denote the relations where -1 corresponds to $x < y$, 0 to $x{=}y$, and 1 to $x > y$. If the current location has the label 0 for the pair (x, y), we check the updates to decide on the label of the next location, e.g., incrementing x results in a transition to a location with label 1 for the corresponding pair. If the location is labeled with -1 or 1 for (x, y), we test for $x{=}y$ at every input symbol, and stay at the locations implying the same relation as long as the test is false. If it becomes true, we take the transition to the location labeled with 0 for the pair, and repeat the process by enabling the inequality test by 3-valued labels. □

Proof of Theorem 4.6. Given $a, b \in \mathbb{N}$ and some term t, we use $t + a$ as a shorthand for $t + 1 + \ldots + 1$ (a times), and symmetrically for $t - b$. Let us informally refer to registers with instruction set $\langle -1, +1, =0\rangle$ as *signed* counters, and to registers with instruction set $\langle +1, =\rangle$ as *unsigned* counters.

A signed counter x can be replaced by two unsigned counters x^+, x^- respectively storing the number of increments and decrements. An update $x \leftarrow y + a - b$ is implemented by $x^+ \leftarrow y^+ + a, x^- \leftarrow y^- + b$. An update $x \leftarrow a - b$ is implemented by $x^+ \leftarrow x^- + a, x^- \leftarrow x^+ + b$. A test $x + a - b = 0$ is achieved by testing for $x^+ + b = x^- + a$. Tests and updates can always be put in this form.

In the other direction, every pair of two unsigned counters x, y is replaced by a signed counter r_{x-y} storing their difference. A pair of updates $x \leftarrow x' + a, y \leftarrow y' + b$ is emulated with update $r_{x-y} \leftarrow r_{x'-y'} + a - b$. Then $x + a = y + b$ if and only if $r_{x-y} + (a - b) = 0$.

The transformed monitor halts iff the original one does. □

Proof of Theorem 4.8. Let us denote by S the set of bad prefixes for L_1'. Let $\mathcal{A}$ satisfy $(q_0, v_0) \xrightarrow{a^n} (q_1, v_1)$ and $(q_0, v_0) \xrightarrow{a^m} (q_2, v_2)$. Consider the strings $u_1 = a^n b^n$ and $u_2 = a^m b^n$ such that $(q_0, v_0) \xrightarrow{u_1} (q_1', v_1')$ and $(q_0, v_0) \xrightarrow{u_2} (q_2', v_2')$. Clearly, if $m < n$ then $u_1 \notin S$ whereas $u_2 \in S$. Assume that $(q_1, v_1) = (q_2, v_2)$, that is, runs over a^n and a^m result in the same configuration. Then, since $(q_1, v_1) \xrightarrow{b^n} (q_1', v_1')$ and $(q_2, v_2) \xrightarrow{b^n} (q_2', v_2')$ and $\mathcal{A}$ is deterministic, $(q_1', v_1') = (q_2', v_2')$. It implies that $u_1 \in S$ if and only if $u_2 \in S$. However, if $m < n$ then $u_2 \in S$, hence we cannot have $v_1 = v_2$ for such m and n.

In other words, two strings of a's of different lengths cannot produce the same configuration of register values. Hence, we can always find a string of consecutive a's of length at most 1 more than the boundedly many number of configurations, $q(2cd + 1)^k + 1$, such that at least one of the k registers goes out of the range $[-cd, cd]$. □

Proof of Theorem 4.11. The idea is to apply a translation which is almost identical to that of Theorem 4.6 in two steps, in order to reach from monitors with instructions in $\langle +1, =\rangle$, copyless monitors with instructions in $\langle 0, +1, =\rangle$. We use copyless monitors with instructions in $\langle 0, -1, +1, =0\rangle$ as an intermediary stage in this translation. By Theorem 4.4, we assume without loss of generality that all register monitors have rate 1.

Assume a register monitor with instruction set $\langle +1, =\rangle$. We construct a copyless register monitor with instructions in $\langle 0, -1, +1, =0\rangle$ by storing and maintaining the difference between variables in pairs, as described in the proof of Theorem 4.6. Notice that whenever a copy occurs, it is replaced by a reset. Formally, a pair of updates $x \leftarrow z + a, y \leftarrow z + b$ is simulated by $r_{x-y} \leftarrow a - b$. The rest of the construction is as previously. For the next step we go from copyless monitors with instructions $\langle 0, -1, +1, =0\rangle$ to copyless monitors with instructions $\langle 0, +1 =\rangle$. We use the other direction of the translation

in Theorem 4.6, in which we additionally simulate each reset $x \leftarrow a$ by $x^+ \leftarrow a, x^- \leftarrow 0$ if $a \geq 0$, $x^+ \leftarrow 0, x^- \leftarrow -a$ otherwise. □

In Section 5

Proof of Theorem 5.3. We show that the language M_1 is not recognizable by any counter monitor. Assume towards a contradiction the existence of a counter monitor $\mathcal{A}$ with k registers, m locations, and rate c recognizing language M_1. The set S_w of inequivalent prefixes of length n and the set S_c of possible configurations after n steps are given as follows:

$$S_w = \{w \mid w \in \Sigma^n\} \text{ and } S_c = \{(q, v) \mid w \in S_w \wedge (q_0, v_0) \xrightarrow{w} (q, v)\}.$$

Note that $|S_w| = 2^n$ and $|S_c| = m(cn)^k$. Since $2^n > m(cn)^k$ for some sufficiently large n, it is clear that there exist some strings $w_1 \neq w_2$ in S_w resulting in the same configuration. In this case, the monitor $\mathcal{A}$ behaves the same on $(w_1\#)^\omega$ and on $w_2\#(w_1\#)^\omega$, which contradicts the definition of M_1. Hence, such a counter monitor cannot exist. □

Proof of Claim 1. Since registers only have non-negative integer values, if there were more than ml^2 such words then by pigeonhole principle there would exist two words $w \neq w'$ satisfying the above condition and such that $x(w) = x(w')$, $y(w) = y(w')$, and $\mathcal{A}$ in the same location after reading either w or w'. But then $w\#w$ and $w'\#w$ are indistinguishable by $\mathcal{A}$, which contradicts the fact that $w'\#w$ is bad for N_2, while $w\#w$ isn't. □

Proof of Claim 2. Assume $x(\#u\#v\#u') \neq y(\#u\#v\#u')$ holds for all factorizations $u'u'' = u \in a^*$ with $|u''| \leq m + 1$, in search of a contradiction. Then, we have in particular $x(\#u\#v\#u) \neq y(\#u\#v\#u)$, and $x(\#u\#v\#u_2) \neq y(\#u\#v\#u_2)$ for at least one $u_2 \in \Sigma^*$ with $u_2 \neq u$ such that $\mathcal{A}$ is in the same location reading $\#u\#v\#u$ as after reading $\#u\#v\#u_2$. But then $\mathcal{A}$ either errs on both $\#u\#v\#u\#$ and $\#u\#v\#u_2$, or on neither. This contradicts the fact that $\mathcal{A}$ recognizes N_2 in real-time. □

Proof of Theorem 5.9. Let us call signed adder monitor a register monitor with instruction set $\langle 0, 1, +, -, =0\rangle$, and unsigned adder monitor as previously.

A signed adder x can be simulated by two unsigned adders x^-, x^+ similarly as previously. Updates $x \leftarrow \lambda$ where λ is a linear combination of variables are simulated with $x^+ \leftarrow \lambda^+$ and $x^- \leftarrow \lambda^-$, where $\lambda \equiv \lambda^+ - \lambda^-$ and λ^+, λ^- only have positive coefficients. Tests of the form $\lambda = 0$ are equivalent to $\lambda^+ = \lambda^-$.

Updates in an unsigned adder are a special case of those in a signed one. Tests $\lambda_1 = \lambda_2$ are naturally emulated as $\lambda_1 - \lambda_2 = 0$. □

In Section 6

Proof of Theorem 6.2. Every tape-head unit t can be encoded using six adder registers as follows. First, the tape is divided between the parts lying at the right and left of its head. Each of these parts is essentially a push-down store, that we maintain as follows. We use some register x storing the content of the store in binary notation, padded with m zeros. Auxiliary registers y and z are used to read and write into the store, and hold 2^m and $2^m - 1$ respectively. To push a symbol a on the stack we update y, z with $y \leftarrow 2y, z \leftarrow z + y$. To push a symbol b on the stack we update x, y, z with $x \leftarrow x+2y, y \leftarrow 2y, z \leftarrow z + y$. To pop a symbol form the stack we must first read it, and then erase it. Reading is done by using inequality $x \leq z$, which holds if and only if the symbol a is at the top of the stack. Erasing is done by $x \leftarrow x - y$ if b is at the top of the stack, nothing if a is at the top stack, and padding x with one more zero, via $x \leftarrow 2x$. □