On the Succinctness of Idioms for Concurrent Programming

David Harel[1], Guy Katz[1], Robby Lampert[2], Assaf Marron[1], and Gera Weiss[3]

1 **Weizmann Institute of Science, Rehovot, Israel**
2 **Mobileye Vision Technologies Ltd., Jerusalem, Israel**
3 **Ben Gurion University, Beer-Sheva, Israel**

Abstract

The ability to create succinct programs is a central criterion for comparing programming and specification methods. Specifically, approaches to concurrent programming can often be thought of as idioms for the composition of automata, and as such they can then be compared using the standard and natural measure for the complexity of automata, descriptive succinctness. This measure captures the size of the automata that the evaluated approach needs for expressing the languages under discussion. The significance of this metric lies, among other things, in its impact on software reliability, maintainability, reusability and simplicity, and on software analysis and verification. Here, we focus on the succinctness afforded by three basic concurrent programming idioms: requesting events, blocking events and waiting for events. We show that a programming model containing all three idioms is exponentially more succinct than non-parallel automata, and that its succinctness is additive to that of classical nondeterministic and "and" automata. We also show that our model is strictly contained in the model of cooperating automata *a la* statecharts, but that it may provide similar exponential succinctness over non-parallel automata as the more general model — while affording increased encapsulation. We then investigate the contribution of each of the three idioms to the descriptive succinctness of the model as a whole, and show that they each have their unique succinctness advantages that are not subsumed by their counterparts. Our results contribute to a rigorous basis for assessing the complexity of specifying, developing and maintaining complex concurrent software.

1998 ACM Subject Classification D.1.3 Concurrent Programming

Keywords and phrases Descriptive Succinctness, Module Size, Automata, Bounded Concurrency

1 Introduction

As is well known, many measures of computational complexity are used to compare solutions to algorithmic and software development problems. However, when it comes to comparing the methods, languages and tools that are used to construct those solutions, one needs quite different criteria for comparison. One of the main approaches to this, which has been used ever since the Rabin-Scott work on nondeterministic automata [22], is the size of the description. Size comparisons are usually carried out on the finite automata level of detail, and the most common metric, often called *descriptive succinctness* or *state complexity*, is the total number of states needed by the automata to express certain languages.

A large amount of work has been dedicated to descriptive succinctness in recent decades. A few notable models whose succinctness has been studied in detail are nondeterministic and universal automata, alternating automata, reverse automata, unary automata, and also various kinds of grammars and language formalisms (see, e.g., [15] for a survey). These studies have been motivated by the strong connection between succinctness and *software*

reliability [20], indicating that succinct software is easier to develop, maintain and reuse. Further, the descriptive succinctness of a model is often connected to the complexity of various decision problems in it [15], and hence can be relevant also to verification problems.

In this paper, we set out to analyze the descriptive succinctness of various idioms used in concurrent programming, seeking, as in most previous studies, exponential gaps in descriptive power. In particular, we study whether the addition of certain idioms to a programming model exponentially improves that model's succinctness, and in what cases. In addition to the considerations mentioned above and to our desire to better understand the fundamental nature of these concurrency idioms, our motivation has another aspect: a careful selection of concurrency idioms may make resulting programs more amenable to formal analysis. Thus, a better characterization of concurrency idioms and of the types of problems which they are suitable for solving could allow programmers to more carefully tailor the programming model used to the problem at hand — on the one hand retaining "just enough" concurrency to efficiently solve the problem, while on the other hand keeping the model simple and amenable to analysis. However, these topics are beyond the scope of this paper; for a broader discussion of the usefulness of keeping concurrency idioms simple in tasks like program repair and compositional verification we refer the interested reader to [9, 11, 12].

Here, we focus on three fundamental concurrency idioms: requesting, blocking and waiting for events (defined formally in Section 2). The requesting and waiting-for idioms are fairly common in discrete-event programming languages, with versions thereof appearing as first-class citizens in, e.g., *publish-subscribe* architectures [6]; whereas the blocking idiom is somewhat less common, appearing, e.g., in the *live sequence charts* (LSCs) formalism [4]. All three idioms can, of course, be implemented in any high level language. Combined, they also form the *behavioral programming* (*BP*) model [13]. Research suggests that using these idioms may lead to simple code modules that are aligned with the specification [13].

Following the required definitions presented in Section 2, the paper's contributions appear in Sections 3 and 4. In Section 3 we study a model containing the requesting, waiting-for and blocking idioms (which we call the $\mathcal{RWB}$ model), and position it in comparison to other well known models. Specifically, we show that $\mathcal{RWB}$ is polynomially expressible as automata with cooperative concurrency *a la* statecharts [5], but that cooperative concurrency can be exponentially more succinct than $\mathcal{RWB}$. We then show that despite this gap, the $\mathcal{RWB}$ model, which affords greater encapsulation, shares some of the cooperative model's strength and offers considerable advantages when compared to non-parallel automata. Next, we show that the succinctness of $\mathcal{RWB}$ is additive to that of classical nondeterminism and universal ("and") nondeterminism, and that a combination of all three features yields a triple-exponential improvement in succinctness. This last result establishes a hierarchy of succinctness relations indicating, e.g., that the (more practical) nondeterministic or universal $\mathcal{RWB}$ models are double-exponentially more succinct than non-parallel automata.

Next, in Section 4, we study the separate contribution of each of $\mathcal{RWB}$'s idioms to the model's descriptive succinctness. We define variants of $\mathcal{RWB}$ in which each of these idioms is omitted, and show that the full $\mathcal{RWB}$ model has exponential succinctness advantages over each of the variants. We also show that each of these downgraded versions has succinctness advantages over one or both of the other downgraded versions and over non-parallel models. This establishes the fact that each of the idioms makes its own unique contribution to succinctness, and is not subsumed by its counterparts. Notable among these results is the fact that event blocking, which is less common as a first-class concurrency idiom, provides exponential savings in succinctness. Further, we show that the succinctness afforded by each of these three idioms is not of equal power: for instance, the waiting-for idiom is weaker than

the requesting one. Related work appears in Section 5, and we conclude with Section 6.

2 Definitions

2.1 The Request-Wait-Block Model

In this work we focus on the *Request-Wait-Block* ($\mathcal{RWB}$) model for concurrent programs. As we mentioned before, the requesting, waiting-for and blocking idioms are common and appear in various models such as *publish-subscribe* architectures [6], *live sequence charts* [4] and *behavioral programming* [13]. Further, research has shown that these idioms often enable programmers to specify and develop systems naturally and incrementally, with components that are aligned with how humans often describe behavior [7, 13]. Still, the $\mathcal{RWB}$ model is not intended to be programmed in directly — rather, it is intended as a formal representation of programs written in higher level languages, for the sake of rigorous analysis.

The formal definitions of the $\mathcal{RWB}$ model are as follows. An *$\mathcal{RWB}$-automaton* consists of orthogonal components called $\mathcal{RWB}$-threads:

▶ **Definition 1.** A *Request-Wait-Block-thread* (*$\mathcal{RWB}$-thread*) is a tuple $\langle Q, \Sigma, \delta, q_0, R, B\rangle$, where Q is a finite set of states, Σ is a finite set of events, $\delta \subseteq Q \times \Sigma \times Q$ is a transition relation and q_0 is an initial state. We require that δ be deterministic, i.e. $\langle q, e, q_1\rangle \in \delta \wedge \langle q, e, q_2\rangle \in \delta \implies q_1 = q_2$. For simplicity of notation, we use $\bar{\delta}$ to indicate the effect event e has in state q (or its absence):

$$\bar{\delta}(q,e) = \begin{cases} q' & ;\text{if exists } q' \in Q \text{ such that } \langle q, e, q'\rangle \in \delta \\ q & ;\text{otherwise}\,. \end{cases}$$

The mapping functions $R, B \colon Q \to 2^{\Sigma}$ associate a state with the set of events *requested* and *blocked*, respectively, by the $\mathcal{RWB}$-thread in that state.

Observe that there is no labeling function for waited-for events: the notion of waiting is expressed via the transitions between states. If state q has a transition labeled with event e that was not requested at q, the thread is considered to be waiting for event e in state q.

A composition of $\mathcal{RWB}$-threads yields an $\mathcal{RWB}$-automaton, defined as follows:

▶ **Definition 2.** An *$\mathcal{RWB}$-automaton* (*RWBA*) A over a finite event set Σ is a finite tuple of $\mathcal{RWB}$-threads $\langle T_1, \ldots, T_n\rangle$, denoted $T_i = \langle Q^i, \Sigma^i, \delta^i, q_0^i, R^i, B^i\rangle$, such that $\Sigma^i \subseteq \Sigma$ for all i, and the Q_i state sets are pairwise disjoint.

A *configuration* of an RWBA is the state of its threads, i.e. an element of $Q^1 \times \ldots \times Q^n$. A configuration $\hat{c} = \langle \hat{q}^1, \ldots, \hat{q}^n\rangle$ is a *successor* of configuration $c = \langle q^1, \ldots, q^n\rangle$ with respect to an event $e \in \Sigma$, denoted $c \xrightarrow{e} \hat{c}$, whenever

$$\underbrace{e \in \bigcup_{i=1}^{n} R^i(q^i)}_{e \text{ is requested}} \bigwedge \underbrace{e \notin \bigcup_{i=1}^{n} B^i(q^i)}_{e \text{ is not blocked}} \bigwedge_{i=1}^{n} \Big(\underbrace{(e \in \Sigma^i \implies \hat{q}^i = \bar{\delta}^i(q^i, e))}_{\substack{\text{affected threads read the event} \\ \text{and change state if needed}}} \wedge \underbrace{(e \notin \Sigma^i \implies \hat{q}^i = q^i)}_{\substack{\text{unaffected threads} \\ \text{stay in the same state}}} \Big).$$

Observe that, since the threads have deterministic transition functions, each configuration can have at most one successor with respect to a specific event. It may, however, have multiple successors, each with respect to a different event.

A *run* of A is a sequence of configurations $c_0 c_1 c_2 \ldots$ such that, for all i, c_{i+1} is a successor (with respect to some event) of c_i and $c_0 = \langle q_0^1, \ldots, q_0^n\rangle$ is the initial configuration. A run may be an *infinite* sequence of successive configurations, or a *finite* sequence that ends in

a *terminal configuration*, i.e., a configuration with no successors. Every run $r = c_0 c_1 c_2 \ldots$ of an RWBA induces a set $words(r) = \{\sigma \in \Sigma^* \cup \Sigma^\omega : \forall_{0 \le i < |r|}, c_i \xrightarrow{\sigma[i]} c_{i+1}\}$. Note that $words(r) \subseteq \Sigma^*$ or $words(r) \subseteq \Sigma^\omega$, depending on r being finite or infinite, respectively. We say that a word $\sigma \in \Sigma^* \cup \Sigma^\omega$ is *accepted* by an RWBA A if there is a run r of A such that $\sigma \in words(r)$. The *language* of A, denoted $\mathcal{L}(A)$, is the set of all words accepted by A.

The acceptance condition in this definition is simple — all valid runs are accepted. Of course, the formalism can be modified to cater for more elaborate acceptance conditions, such as conventional accepting states or the various acceptance conditions for ω-automata. The motivation for the present choice is that we regard $\mathcal{RWB}$ as representing the underlying models of programming approaches. As such, languages are seen as generated by, rather than accepted by, a program; indeed, we use these two terms interchangeably.

Next, we define our notion of size, to be used in the analysis of the descriptive succinctness of various variants of RWBAs and other models.

▶ **Definition 3.** The size of an $\mathcal{RWB}$-automaton A with threads $\{\langle Q^i, \Sigma^i, \delta^i, q_0^i, R^i, B^i\rangle\}_{i=1}^n$ is $|A| = \sum_{i=1}^n |Q^i| + |\{\langle q, e, \hat{q}\rangle \in \delta^i\}|$, namely the total number of states and transitions in the threads. For simplicity, the requested and blocked events in every state are omitted from the calculation. They contribute no more than $|\Sigma| \cdot |Q^i|$ to the size of each thread, and have no effect on the size's order of magnitude as $|\Sigma|$ is considered constant.

2.2 Finite Parallel Automata

In order to measure the advantages of $\mathcal{RWB}$ and of other parallel models, we define the following non-parallel model to serve as a reference point:

▶ **Definition 4.** A *deterministic looping automaton* (*DLA*) A is a tuple $\langle Q, \Sigma, \delta, q_0\rangle$, where Q is a set of states, Σ is an alphabet, $\delta \subseteq Q \times \Sigma \times Q$ is a deterministic transition relation and $q_0 \in Q$ is an initial state. As it reads an input word, A traverses its states according to δ, in the usual manner. A accepts infinite words, as well as finite words that end in terminal states (states with no successors). A word is rejected if it contains a letter for which there is no matching transition, or if it ends in a non-terminal state. The *language* $\mathcal{L}(A)$ is the set of words accepted by A, and the *size* of A is $|A| = |Q| + |\{\langle q, e, \hat{q}\rangle \in \delta\}|$, namely the number of states plus the number of transitions in A.

We now discuss other parallel models, focusing on the three fundamental notions: *non-determinism* [22] ($\mathcal{E}$-automata) and its dual, *pure parallelism* ($\mathcal{A}$-automata), which when combined yield *alternating automata* [3], and *cooperative* concurrency ($\mathcal{C}$-automata) [5]. The first two notions take the form of $\exists$- and $\forall$-states in alternating automata, whereas cooperative automata play a role in formalisms and languages such as statecharts [8].

All three features — $\mathcal{E}$, $\mathcal{A}$ and $\mathcal{C}$ — may co-exist. Further, it is shown in [5] that each feature contributes exponentially to the succinctness of the model, independently and additively, so that, e.g., $(\mathcal{E}, \mathcal{A}, \mathcal{C})$-automata allow for triple-exponentially more succinct representations than is possible without these features. Below we give the definition of $(\mathcal{E}, \mathcal{A}, \mathcal{C})$-automata; the other models are regarded as restrictions thereof.

▶ **Definition 5.** An *alternating cooperative automaton* (an $(\mathcal{E}, \mathcal{A}, \mathcal{C})$-*automaton*) over a finite alphabet Σ is a tuple $M = \langle M^1, M^2, \ldots, M^n, \Phi\rangle$ where each M^i is a triple $\langle Q^i, \delta^i, q_0^i\rangle$. Q^i are pairwise-disjoint state sets and q_0^i are the initial states. $\delta^i \subseteq Q^i \times \Sigma \times \Gamma \times Q^i$ are transition relations, where Γ is the set of propositional formulas over the states of all components, $\bigcup_{i=1}^n Q^i$. Elements from Γ serve as *guards*: a transition can be applied only if its guard

evaluates to true. For example, for $q_1 \in Q^1$ and $q_2 \in Q^2$, the guard $q_1 \wedge \neg q_2$ evaluates to true precisely when component M^1 is in state q_1 and component M^2 is not in state q_2. Finally, $\Phi \in \Gamma$ is the $\mathcal{E}$-condition — a condition that, when true, implies that the configuration is existential (an *$\mathcal{E}$-configuration*); otherwise, the configuration is universal (an *$\mathcal{A}$-configuration*). In [5], these automata include a termination condition as well, but as we deal with the simple variant of looping automata, we may omit it.

A *configuration* of M is an element of $Q^1 \times Q^2 \times \ldots \times Q^n \times (\Sigma^* \cup \Sigma^\omega) \times \mathbb{N}$, indicating the state of each component, the (finite or infinite) input word, and the position of M in that word. A configuration c satisfies a guard condition $\gamma \in \Gamma$ if γ evaluates to true when assigned the states of c. Let $\sigma = \sigma_0\sigma_1 \ldots \in \Sigma^* \cup \Sigma^\omega$ and let $t = \langle q, a, \gamma, p\rangle$ be a transition in δ^i. We say that t is applicable to a configuration $c = \langle q^1, \ldots, q^n, \sigma, j\rangle$ if $\sigma_j = a, q^i = q$ and c satisfies γ. A configuration $\langle p^1, \ldots, p^n, \sigma, m\rangle$ is a successor of c if for each i there is a transition $\langle q^i, \sigma_j, \gamma^i, p^i\rangle \in \delta^i$ that is applicable to c, and $m = j + 1$.

A computation of M on input word σ can be described as a tree. It starts at the initial configuration $\langle q_0^1, q_0^2, \ldots, q_0^n, \sigma, 1\rangle$, and reads a letter. If the state has multiple successors, the computation "splits", and progresses in parallel for all possible successor states. The process then continues. Any infinite path in this tree is said to be *accepting*. A finite path is accepting iff it ends in a terminal configuration (a configuration with no successors). An $\mathcal{E}$-configuration is accepting iff there *exists* an accepting path starting at that state, whereas an $\mathcal{A}$-configuration is said to be accepting iff *every* path starting at that state is accepting. Word σ is accepted by M iff the root of its computation tree is accepting.

If each configuration of M has a single successor (i.e., all transitions are deterministic), we have a $\mathcal{C}$-automaton, which we might call a cooperative automaton. When $n = 1$ it is in fact an $(\mathcal{E}, \mathcal{A})$-automaton: an alternating looping automaton. When $n = 1$ and $\Phi = true$, M is a nondeterministic looping automaton; and when $n = 1$ and $\Phi = false$ it is a universal looping automaton. Finally, when both $n = 1$ and every configuration has a single successor, M is simply a deterministic looping automaton — a DLA.

▶ **Definition 6.** The *size* of an $(\mathcal{E}, \mathcal{A}, \mathcal{C})$-automaton M is defined to be the sum of the sizes of its condition and components; i.e. $|M| = |\Phi| + \sum_{i=1}^{n} |M^i|$, where $|M^i| = |Q^i| + \sum_{\langle q,a,\gamma,p\rangle \in \delta^i} |\gamma|$. A condition's size is defined as the length of the formula that represents it.

2.3 Succinctness Gaps

We next lay out the method of comparing the succinctness of two models. Informally, we say that a computational model $\mathcal{M}_1$ is more succinct than model $\mathcal{M}_2$ if there are programs that have descriptions in $\mathcal{M}_1$ that are significantly smaller than the smallest possible descriptions for those programs in $\mathcal{M}_2$. In this paper we consider a gap to be significant if it is at least exponential. Following [5], we define upper and lower bounds on gaps in succinctness:

▶ **Definition 7.** Let $\mathcal{M}_1, \mathcal{M}_2$ denote two computational models. We write $\mathcal{M}_1 \xrightarrow{p} \mathcal{M}_2$ (resp., $\mathcal{M}_1 \xrightarrow{\cdot} \mathcal{M}_2$) if there is a polynomial p (resp., a polynomial p and a constant $k > 1$) such that for any automaton $M_1 \in \mathcal{M}_1$ of size m there is an automaton $M_2 \in \mathcal{M}_2$ such that $\mathcal{L}(M_1) = \mathcal{L}(M_2)$, and M_2 is of size no more than $p(m)$ (resp., $k^{p(m)}$). In this case, we say that $\mathcal{M}_1$ is *at most polynomially* (resp., *exponentially*) *more succinct than* $\mathcal{M}_2$.

We write $\mathcal{M}_1 \xrightarrow{\cdot\cdot} \mathcal{M}_2$ if there is a family of ω-regular languages L_n, a polynomial p and a constant $k > 1$, such that L_n is accepted by an automaton $M_1 \in \mathcal{M}_1$ of size $p(f(n))$ for some monotonically-increasing function f, but the smallest $M_2 \in \mathcal{M}_2$ accepting it is at least of size $k^{f(n)}$. In this case, we say that $\mathcal{M}_1$ is *at least exponentially more succinct than* $\mathcal{M}_2$.

3 $\mathcal{RWB}$ and Parallel Automata

In this section, we investigate how $\mathcal{RWB}$-automata fare when considered in the context of $\mathcal{E}$-, $\mathcal{A}$- and $\mathcal{C}$-automata; that is, how the special $\mathcal{RWB}$ idioms relate to the conventional idioms of and- and or-nondeterminism and bounded concurrency. We observe that, of the three models, $\mathcal{RWB}$ seems most closely related to $\mathcal{C}$ — as the threads of an RWBA constitute cooperating components running in parallel — although this cooperation is more limited than in the $\mathcal{C}$ model. The first part of this section validates this observation, by proving that $\mathcal{RWB} \xrightarrow{p} \mathcal{C}$, but that $\mathcal{C} \not\xrightarrow{} \mathcal{RWB}$. This establishes a firm succinctness relationship between $\mathcal{C}$ and $\mathcal{RWB}$: the former is strictly stronger.

The proof that $\mathcal{C} \not\xrightarrow{} \mathcal{RWB}$ revolves around *counting* — a task for which the $\mathcal{C}$ model is particularly suited, as it allows one to count to n using automata of size only $O(\log^2 n)$ [5]. As we prove, in the general case of counting, $\mathcal{RWB}$-automata must be of size n, which is exponentially worse. This result gives rise to the question: does $\mathcal{RWB}$ retain any of $\mathcal{C}$'s power, i.e. is it succinctness-wise better than non-parallel automata?

We answer the question in the affirmative, in two parts. First, we show that $\mathcal{RWB}$ shares some of the power of $\mathcal{C}$ automata; e.g., in certain cases it is possible to count to n with $\mathcal{RWB}$-automata of size $O(\log^2 n \cdot \log\log n)$, and so $\mathcal{RWB} \not\xrightarrow{}$ DLA. Second, we study the relationship between $\mathcal{RWB}$ and the $\mathcal{E}$ and $\mathcal{A}$ models, and show that $\mathcal{RWB}$ can sometimes replace $\mathcal{C}$ in $(\mathcal{E}, \mathcal{A}, \mathcal{C})$-automata, while preserving that model's descriptive succinctness.

The relationship between $\mathcal{E}, \mathcal{A}$ and $\mathcal{C}$ has been extensively studied in [5], where it is shown that they are *orthogonal*, i.e. that their descriptive succinctness is independent and additive. In particular, [5] shows that the $(\mathcal{E}, \mathcal{A}, \mathcal{C})$ model offers a tight triple-exponential gap in succinctness compared to non-parallel automata. Our proof that the $(\mathcal{E}, \mathcal{A}, \mathcal{RWB})$ model affords the same triple-exponential gap thus strengthens the original result of [5], as it shows that a model in which components cannot freely observe other components' states, and is thus more encapsulated than $\mathcal{C}$, suffices for obtaining the triple exponential gap.

3.1 $\mathcal{RWB}$-Automata and $\mathcal{C}$-Automata

Of the three models $\mathcal{E}, \mathcal{A}$ and $\mathcal{C}$, it is natural to define $\mathcal{RWB}$ programs in terms of $\mathcal{C}$-automata, as the underlying parallel components of both make transitions that depend on other components. $\mathcal{C}$-automata take the most general form, allowing components to query the internal states of other components. This is established in the following proposition, proven in Appendix A of the supplementary material [10]:

▶ Proposition 1. $\mathcal{RWB} \xrightarrow{p} \mathcal{C}$

We next show that the converse does not hold; i.e., that there exists a family of languages that can be expressed succinctly using $\mathcal{C}$-automata, but that the smallest RWBAs that can express them are exponentially larger.

▶ Proposition 2. $\mathcal{C} \not\xrightarrow{} \mathcal{RWB}$

Proof. For $n \in \mathbb{N}$, consider the language $L_n = (0+1)^n 0^\omega$. For every n, there exists a $\mathcal{C}$-automaton of size $O(\log^2 n)$ that accepts L_n, as follows. The automaton consists of $\log n$ components, each representing a single bit of a $(\log n)$-bit counter that counts to n. Carries are performed using the guards: bit number $i+1$ moves from state 0 to 1 if and only if all previous bits $1 \ldots i$ are in state 1. A final transition occurs when the counter reaches n, into a state that only allows 0s. As the $\log n$ components can have size $\log n$ because of the transition guards, the automaton is of size $O(\log^2 n)$. See [5] for details.

Now, let us consider the same language in the $\mathcal{RWB}$ model. Suppose that an $\mathcal{RWB}$-automaton A with threads $T_1, \ldots, T_k$ accepts L_n. We show that at least one of these threads has to have $\Omega(n)$ states, thus proving the claim. Intuitively, the proof relies on the fact that while A reads the n-bit prefix of the word the threads cannot use events to communicate between themselves, and so a single thread has to handle the counting up to n.

Suppose, contrary-wise, that all threads have fewer than n states, and consider the word $\sigma = 0^{n-1} \cdot 1 \cdot 0^\omega \in L_n$. Examine an arbitrary thread T_i as it reads the $\rho = 0^{n-1}$ prefix of σ. By our assumption, thread T_i has fewer than n states. Consequently, by the pigeonhole principle, it has a state s_1 that it will visit at least twice as it reads ρ. The portion of the path of states that it traverses between these two visits, denoted $s_1 \xrightarrow{0} s_2 \xrightarrow{0} \ldots \xrightarrow{0} s_{\alpha_i} \xrightarrow{0} s_1$, constitutes a *cycle* of length α_i in the thread's state graph. This holds for every thread T_i, and so all the threads must traverse cycles of lengths $\alpha_1, \ldots, \alpha_n$ as they read ρ.

We now use a pumping argument to show that A accepts a word that is not in L_n. Let $\beta = \prod_{i=1}^{n} \alpha_i$. Consider the word $\sigma' = 0^{n-1} \cdot 0^\beta \cdot 1 \cdot 0^\omega$, and its prefix $\rho' = 0^{n-1} \cdot 0^\beta$. The word 0^ω is in L_n, and ρ' is a prefix of this word; hence, the automaton cannot reject the input word after reading ρ'. However, as the threads are traversing cycles of lengths that divide β, they will each be in the same state after reading ρ' as they would be after reading ρ. Thus, as they read the $1 \cdot 0^\omega$ suffix of σ', they would accept the word — just as they would accept σ. Since $\sigma' \notin L_n$, this is a contradiction. ◀

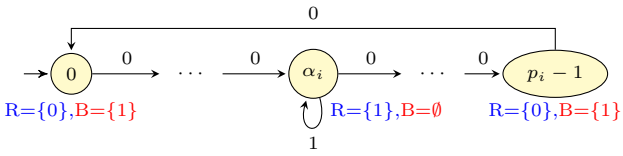
We note that the gap shown by Proposition 2 is tight, in the sense that $\mathcal{C}$-automata are at most (single) exponentially more succinct than $\mathcal{RWB}$-automata. See Appendix B of the supplementary material [10] for the proof.

3.2 Counting with Succinct $\mathcal{RWB}$-Automata

Proposition 2 implies that perhaps the $\mathcal{RWB}$ model is not much stronger than non-parallel automata; indeed, for the task of counting, an RWBA requires as many states as a DLA — exponentially many more than a $\mathcal{C}$-automaton requires. However, the main difference in power between $\mathcal{C}$ and $\mathcal{RWB}$ is in the ability of one component in a $\mathcal{C}$-automaton to observe the state of another without any restrictions, whereas in $\mathcal{RWB}$ a marker event (a *sentinel*) must be triggered for such an observation to be made. Thus, when a sentinel is present, the difference in succinctness between $\mathcal{C}$-automata and $\mathcal{RWB}$-automata diminishes greatly:

▶ Proposition 3. For every $n \in \mathbb{N}$, there exists an $\mathcal{RWB}$-automaton A_n that accepts the language $L_n = 0^n 1^\omega$, such that A_n is of size $O(\log^2 n \cdot \log\log n)$.

Proof. We use the first appearance of 1 to mark the end of the counting phase. Let $k \in \mathbb{N}$ be the smallest number such that the first k prime numbers $p_1, \ldots, p_k$ satisfy $\prod_{i=1}^{k} p_i > n$, and let $\langle \alpha_1, \ldots, \alpha_k \rangle$ be defined by $\alpha_i = n \bmod p_i$ for all $1 \le i \le k$. By the Chinese Remainder Theorem, n is the only integer in the range $[1..\prod_{i=1}^{k} p_i]$ that has these remainders. Consider the $\mathcal{RWB}$-automaton A_n that has k threads $T_1, \ldots, T_k$, where thread T_i is given by:



The sets of events requested (R) and blocked (B) in each state are listed by that state. In state α_i the thread requests 1 and blocks nothing, and in the other states it requests 0

and blocks 1. To see that this automaton accepts L_n, note that if even one of the threads is not in its respective α_i state, the next event in any accepted word has to be 0, because that thread requests 0 and blocks 1 and no thread ever blocks 0. On the other hand, once all threads are in their α_i states the only requested event is 1, resulting in a 1^ω suffix. Finally, The Chinese Remainder Theorem guarantees us that the first time the threads are all in their α_i states is precisely at the nth step, as required.

Since we chose the smallest k for which $\prod_{i=1}^{k} p_i > n$, it follows that $k = O(\log n)$. By the Prime Number Theorem we have $p_i = O(i \log i)$. Combining the two, we get that the total size of A_n is indeed $O(\log^2 n \cdot \log\log n)$. ◀

From Proposition 3 it follows that $\mathcal{RWB} \not\xrightarrow{\cdot}$ DLA. Further, because $\mathcal{RWB} \xrightarrow{p} \mathcal{C}$ and $\mathcal{C} \xrightarrow{\cdot}$ DLA [5], we get that $\mathcal{RWB} \xrightarrow{\cdot}$ DLA, i.e. that the bound is tight.

3.3 Combining $\mathcal{RWB}$ with $\mathcal{E}$- and $\mathcal{A}$-Automata

One of the main results of [5] establishes a tight triple-exponential gap in succinctness between $(\mathcal{E}, \mathcal{A}, \mathcal{C})$-automata and DLA. Specifically, there exists a family of languages L_n expressible by $(\mathcal{E}, \mathcal{A}, \mathcal{C})$-automata of size $O(\log^2 n)$, but that require at least 2^{2^n} states when expressed by a DLA. In this section we quantify the succinctness gap between the $(\mathcal{E}, \mathcal{A}, \mathcal{RWB})$ model — where $\mathcal{C}$ is replaced by $\mathcal{RWB}$ — and the DLA model.

The semantics of an $(\mathcal{E}, \mathcal{A}, \mathcal{RWB})$-automaton is as follows: as before, the threads run in parallel, and a transition may occur if the event is requested by at least one thread and is blocked by none. In this model, unlike in $\mathcal{RWB}$, we allow nondeterministic transitions in threads, and so a state may have multiple outgoing transitions labeled with the same event. We also adapt the $\mathcal{E}$-condition to operate in an $\mathcal{RWB}$-like fashion, by allowing threads to request/block that a configuration be universal. Thus, a configuration is existential by default, but becomes universal if this was requested by at least one thread and blocked by none (this $\mathcal{E}$-condition is somewhat arbitrary — other definitions could be used as well). Observe that this form of $\mathcal{E}$-condition is a restriction (i.e., a special case) of the $\mathcal{E}$-condition of [5]. The acceptance criteria is the same as for $(\mathcal{E}, \mathcal{A}, \mathcal{C})$ (see Definition 5).

Having shown in Proposition 1 that $\mathcal{RWB} \xrightarrow{p} \mathcal{C}$, it follows that the upper bound of [5] holds; that is, a program in the $(\mathcal{E}, \mathcal{A}, \mathcal{RWB})$ model will incur at most a triple-exponential blowup when transformed appropriately into a DLA. Next, we show that this bound is tight, by establishing a corresponding lower bound. The family of languages that we use is an adaptation of a similar family from [5]:

$$L_n = \{(0+1+\#)^*\#w\#(0+1+\#)^*\#\$w\bot 0^\omega \mid w \in \{0,1\}^n\} \cup \{(0+1+\#)^\omega\}$$

over the alphabet of $\{0, 1, \#, \$, \bot\}$. Intuitively, an automaton that accepts L_n encounters a sequence of words, separated by #s. Then, it encounters a \$, followed by a word w, terminated by $\bot$. The automaton must then decide if this w is of size n, and whether it was encountered before, in the initial sequence of words. If the answer is *yes*, the automaton accepts the word if it ends in an infinite sequence of 0s; otherwise, it rejects the word. The automaton also accepts all words in which the \$ and $\bot$ signs never appear.

Pigeonhole and pumping arguments show that a non-parallel automaton that recognizes the language has to remember, by the time it reaches the \$ sign, all the words of length n that it has encountered previously. Thus, it must have at least 2^{2^n} states [3, 19]. However, an $(\mathcal{E}, \mathcal{A}, \mathcal{RWB})$-automaton for L_n may be triple-exponentially smaller, as we now show:

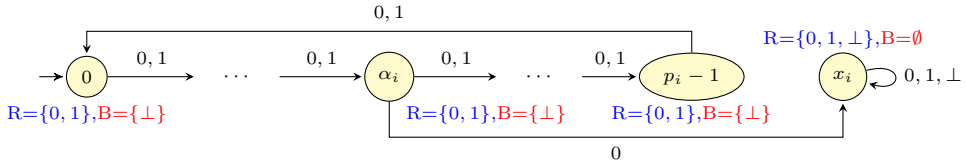
▶ Proposition 4. L_n is recognizable by an $(\mathcal{E}, \mathcal{A}, \mathcal{RWB})$-automaton of size $O(\log^2 n \cdot \log\log n)$.

Proof. Due to space considerations, we provide here only the core of the proof. Our strategy, inspired by [5], is as follows: in words where the \$ and $\perp$ signs appear, the automaton's nondeterminism is used to "guess" when the first instance of w is encountered. Then, universality is used to compare all n bits of the two occurrences of w simultaneously. Finally, ensuring that both copies of w are of length n, and performing the necessary counting to compare each pair of bits, is performed efficiently by the automaton's $\mathcal{RWB}$-threads.

More explicitly, the $\mathcal{RWB}$ idioms are used for 3 tasks: (1) verifying that the first occurrence of w is of size n; (2) verifying that the second occurrence of w is of size n; and (3) comparing a single pair of bits in the two occurrences of w. Because task (3) is performed universally for all n bits, it ensures that the two occurences of w are equal. For task (3) the automaton counts to n, but is suspended on # and resumed on \$. Thus, when the counting is finished, the next symbol should match the symbol on which the counting was started.

Tasks (1) and (2) can be performed succinctly by an RWBA, as both occurrences of w in L_n are terminated by a sentinel — # or $\perp$. Thus, the construction from Section 3.2 suffices. The automaton size these tasks require is $O(\log^2 n \cdot \log\log n)$. Task (3), however, requires counting *without* a sentinel, which — according to the proof of Proposition 2 — requires an $\mathcal{RWB}$-automaton of size $\Omega(n)$. However, we now show that in an $(\mathcal{E}, \mathcal{A}, \mathcal{RWB})$-automaton such counting can actually be performed succinctly, by leveraging the $\mathcal{E}$ and $\mathcal{A}$ idioms.

Let $k \in \mathbb{N}$ be the smallest number such that the first k prime numbers, $p_1, \ldots, p_k$, satisfy $\prod_{i=1}^{k} p_i > n$. Let $\langle \alpha_1, \ldots, \alpha_k \rangle$ be the tuple of remainders, i.e., $\alpha_i = n \bmod p_i$ for all $1 \leq i \leq k$. By the Chinese Remainder Theorem, these remainders uniquely determine n in the range $[1..\prod_{i=1}^{k} p_i]$. Suppose, without loss of generality, that the symbol in the first occurrence of w was 0. Then, our goal is to count to n and verify that we reach another 0. Consider an $\mathcal{RWB}$-automaton with k threads $T_1, \ldots, T_k$, where T_i is given by:



All states $0, \ldots, p_i - 1$ request both 0 and 1 and block $\perp$; state x_i requests $0, 1$ and $\perp$. Finally, thread T_i requests that the global configuration be universal if and only if it is at state x_i. The details of suspending the count on # and resuming it on \$ are omitted from the figure, to reduce clutter; this can be performed by associating each state $s \in \{1, \ldots, p_i\}$ with an auxiliary state s', and having the appearance of # send the thread to s', where it loops, until a later appearance of \$ sends it back to s. If the \$ and $\perp$ signs do not appear, then the word is accepted, as it has the form $(0 + 1 + \#)^\omega$, which we included in L_n.

Intuitively, the automaton works as follows. All threads traverse their loops, counting to n. While in these loops, a $\perp$ symbol causes the word to be rejected. Hence, the only way a word that has a $\perp$ sign can be accepted is if all threads escape their loops before reaching $\perp$. The only way to escape the counting loops is through the α states. If thread T_i reaches state α_i and reads a 0 symbol, it may escape its loop, assuming the transition is existential; if it is universal, one branch of the thread will remain in the loop, and will reject the word.

The escape transition remains existential until some thread has used it to escape. Afterwards, that thread will remain in its x_i state, requesting that all successive configurations be universal. Hence, all threads must traverse the transition from α_i to x_i simultaneously in order for the word to be accepted. This can only happen if all threads are in their respective α states — which, by the Chinese Remainder Theorem, only occurs at index n — and if

the next symbol is the required 0. Hence, since this testing is performed universally for all symbols in w, the word is rejected if even one pair of matching symbols differs.

We stress that this solution is in line with our previous observations that $\mathcal{RWB}$ is weaker than $\mathcal{C}$, in that $\mathcal{RWB}$ cannot succinctly count without a sentinel. In this construction, the behavior threads use the ability of the $\mathcal{E}$-condition semantics to peek into the states of other threads, thus achieving some of the power of the $\mathcal{C}$-automaton guards, and enabling it to count succinctly, even without a sentinel.

As in Proposition 3, analysis shows that the automaton is of size $O(\log^2 n \cdot \log\log n)$. ◀

We have thus established the triple-exponential succinctness gap between $(\mathcal{E}, \mathcal{A}, \mathcal{RWB})$ and DLA. While $(\mathcal{E}, \mathcal{A}, \mathcal{RWB})$ is not a practical programming model, we observe that, combined with the results of [5] and Proposition 1, this result immediately establishes a succinctness hierarchy concerning other, more practical models, such as $(\mathcal{E}, \mathcal{RWB})$ and $(\mathcal{A}, \mathcal{RWB})$. These corollaries are depicted and explained in Figure 1. In particular, these results indicate that the $\mathcal{RWB}$ idioms of requesting, blocking and waiting for events provide a succinctness advantage that is additive and independent of the succinctness provided by the $\mathcal{E}$ and $\mathcal{A}$ idioms — and that the $\mathcal{RWB}$ idioms are not just those of $\mathcal{E}$- or $\mathcal{A}$-automata in disguise. While similar results were previously shown for the $\mathcal{C}$ model [5], our results are stronger as they show that a limited version of $\mathcal{C}$ already suffices to uphold the hierarchy.

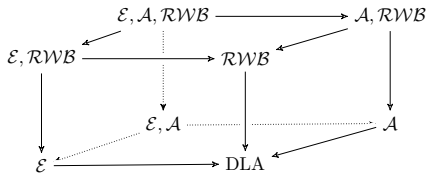


Figure 1 The succinctness hierarchy involving the $\mathcal{E}$, $\mathcal{A}$ and $\mathcal{RWB}$ models, and their combinations. Arrows indicate tight exponential gaps in succinctness. By Proposition 4, the $(\mathcal{E}, \mathcal{A}, \mathcal{RWB})$ model is at least triple exponentially more succinct than the DLA model; and, applying Proposition 1, it is also at most as succinct as the $(\mathcal{E}, \mathcal{A}, \mathcal{C})$ model. Combining this with the fact that $(\mathcal{E}, \mathcal{A}, \mathcal{C})$ is triple exponentially more succinct than DLA [5], we get that the same holds for $(\mathcal{E}, \mathcal{A}, \mathcal{RWB})$. Thus, any path along the edges of the depicted cube, starting at $(\mathcal{E}, \mathcal{A}, \mathcal{RWB})$ and ending at DLA, must include precisely 3 exponential gaps. The tight exponential gaps depicted in the figure then follow from known results regarding alternating automata and $\mathcal{C}$-automata [5], combined with Proposition 1.

From a software-engineering point of view, $\mathcal{C}$-automata afford their succinctness by allowing each component to be aware of the internal state of each of the other components; this liberal awareness is not provided in the $\mathcal{RWB}$ model, resulting in increased module encapsulation, which is usually considered desirable (see, e.g., [21]).

4 Contributions of the Request, Wait, and Block Idioms

Whereas Section 3 was dedicated to comparing $\mathcal{RWB}$ to other parallel models succinctness-wise, in this section we focus on its internal structure. We study each of its main idioms of requesting, waiting-for, and blocking events, and quantify their contribution to the succinctness afforded by $\mathcal{RWB}$ as a whole. Towards this end, we define the following sub-models:

1. The $\mathcal{WB}$ model: Requesting is omitted. Any event that is not blocked can be triggered. Waiting-for and blocking are allowed. This model can be viewed as having all threads

request all events in each state, which, in the notation of Definitions 1 and 2, corresponds to $R^i(q^i) = \Sigma^i$ for every state $q^i \in Q^i$ of thread T_i, for every i.

2. The $\mathcal{RB}$ model: Waiting is omitted; requesting and blocking are allowed. Threads are not informed of events they did not request, and cannot change states when such events are triggered. Formally, for every T_i, if $e \notin R^i(q)$ then $\delta^i(q, e) = q$.
3. The $\mathcal{RW}$ model: Blocking is omitted. Requesting and waiting-for are allowed, and any requested event may be triggered. Formally, $B^i(q) = \emptyset$ for every state q and T_i.

We begin by establishing a simple upper bound, proven in Appendix B of the supplementary material [10]:

▶ Proposition 5. For any $\mathcal{M}_1, \mathcal{M}_2 \in \{\mathcal{RWB}, \mathcal{WB}, \mathcal{RB}, \mathcal{RW}\}$, $\mathcal{M}_1 \dashrightarrow \mathcal{M}_2$

Next, we establish tight bounds on the difference in succinctness of every pair of these models, as depicted in Figure 2.

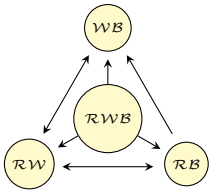


Figure 2 The descriptive succinctness of the $\mathcal{RWB}$, $\mathcal{WB}$, $\mathcal{RW}$ and $\mathcal{RB}$ models, compared to each other. A bi-directional arrow indicates a tight exponential gap in succinctness in both directions — either model may be more succinct than the other. A directed arrow indicates a tight exponential succinctness advantage of the source over the destination, but no such advantage in the reverse direction, i.e., a reverse translation is always possible with only a polynomial blowup.

We begin by proving that the $\mathcal{RWB}$ model is exponentially more succinct than the $\mathcal{WB}$ variant, i.e., that the event requesting idiom exponentially improves the succinctness of the model.

▶ Proposition 6. $\mathcal{RWB} \not\dashrightarrow \mathcal{WB}$

Proof. Let $n \in \mathbb{N}$, and let $k \in \mathbb{N}$ be the smallest number such that the first k prime numbers, $p_1, \ldots, p_k$, satisfy $\prod_{i=1}^{k} p_i > n$. Define the family of languages $L_n = \{\ell_1 \ell_2 \ldots\}$ by:

$$\ell_j = \begin{cases} 0 \text{ or } 1 & ; \quad \exists i \text{ such that } p_i \mid \#_0(\ell_1 \ldots \ell_{j-1}) \\ 0 & ; \quad \text{otherwise} \end{cases}$$

Here $\#_0(\ell_1, \ldots, \ell_{j-1})$ is the number of 0s that have appeared in the word so far. The jth event can be either 0 or 1 if there is a p_i which divides this number. In the $\mathcal{RWB}$ model, this language can be accepted by an automaton of size $O(\log^2 n \cdot \log\log n)$, whereas the smallest $\mathcal{WB}$-automaton that accepts it is of size at least n (for more details, see Appendix C of the supplementary material [10]). ◀

This proof affords some insight into the power of the requesting idiom. Particularly, requesting allows us to succinctly express *or* conditions — i.e., that an event only be triggered if a disjunction of conditions holds.

We now prove that the waiting idiom also affords exponential succinctness:

▶ Proposition 7. $\mathcal{RWB} \not\dashrightarrow \mathcal{RB}$

Proof. Let $n \in \mathbb{N}$, and consider the family of singleton languages $L_n = 0^n 1^\omega$. Section 3.2 shows an $\mathcal{RWB}$-automaton of size $O(\log^2 n \cdot \log\log n)$ that accepts L_n. However, the smallest $\mathcal{RB}$-automaton that accepts L_n must have a thread of size n; see Appendix D of the supplementary material [10] for details. ◀

The language used for this proof illustrates the power of the waiting idiom. In the basic construction of Section 3.2, each thread would count modulo some prime number, and would, upon the correct remainder, request 1. However, that thread would also wait for a 0 event, thus letting other threads supersede it; if one of them determined that it was not yet time to trigger a 1, they would block 1 and request 0. Without the wait-for idiom, however, a thread cannot observe events it did not request, preventing this sort of inter-thread cooperation.

We now show that $\mathcal{RWB}$ is exponentially more succinct than the $\mathcal{RW}$ variant, i.e. event blocking also yields exponential succinctness. We consider this result to be particularly interesting, as blocking is perhaps the least common, or most special, idiom of $\mathcal{RWB}$.

▶ Proposition 8. $\mathcal{RWB} \not\to \mathcal{RW}$

Proof. Let $n \in \mathbb{N}$, and let $k \in \mathbb{N}$ be the smallest number such that the first k prime numbers, $p_1, \ldots, p_k$, satisfy $\prod_{i=1}^{k} p_i > n$. Observe the languages $L_n = (0^{N-1}(0+1))^\omega$, for $N = \prod_{i=1}^{k} p_i$. In $\mathcal{RWB}$, this language is accepted by an automaton of size $O(\log^2 n \cdot \log\log n)$. In the $\mathcal{RW}$ model, however, an automaton accepting this language must be of size at least n (see Appendix E of the supplementary material [10] for precise details). ◀

The language used for the proof gives some intuition as to the power of the blocking idiom. Particularly, it shows that blocking can succinctly enforce *and* conditions — e.g., that an event is not blocked iff it gives the correct remainder for *all* the primes.

We conclude by examining how the $\mathcal{RWB}$ idioms fare with respect to each other. For example, can requesting be replaced by blocking without having to pay with an exponential decrease in succinctness? Our results, illustrated in Figure 2, show that the $\mathcal{WB}$ and $\mathcal{RW}$ models, and also the $\mathcal{RB}$ and $\mathcal{RW}$ models, are incomparable — i.e., there can be exponential gains in both directions. Also, we prove that the $\mathcal{WB}$ model is weaker than the $\mathcal{RB}$ model, and so, in a way, requesting outpowers waiting. We also show that each of the $\mathcal{WB}$, $\mathcal{RB}$ and $\mathcal{RW}$ models is exponentially more succinct than DLA. For more details, see Appendix F of the supplementary material [10].

5 Related Work

In this paper we focused on studying the $\mathcal{RWB}$ concurrency idioms from a succinctness point of view. For a software-engineering oriented comparison between these $\mathcal{RWB}$ idioms (in the context of Behavioral Programming) and other programming models, see [13] and references therein. Below we discuss some notable related work on descriptive succinctness.

Starting with [22], extensive comparative analysis of expressiveness and succinctness in various models of computations has been carried out. Examples include Büchi, Streett, and Emerson and Lei automata [23], two-way finite automata [24, 2], sweeping automata [16], and — most relevant to the present paper — cooperative automata [5, 14]. Expressiveness and succinctness in timed automata are studied in [1].

The issue of counting to n using unary automata, which played a central role in Section 3, was raised in [19] and has been studied extensively. It is well known that counting requires $\Theta(n)$ states in deterministic and nondeterministic finite automata. As any deterministic unary automaton with n states has an equivalent alternating automaton with $O(\log n)$

states [18], it follows that alternating automata can count with size $O(\log n)$. [17] shows a $\Theta(\sqrt{n})$ bound for counting with universal automata, whereas cooperating automata can count to n with size $O(\log^2 n)$ [5]. Counting in other automata types has also been studied: one-switch alternating automata, for instance, count to n with $O(\log^2 n \cdot \log\log n)$ states [2].

6 Conclusion and Future Work

In this work we set out to analyze the descriptive succinctness afforded by various concurrent programming idioms. Our motivation was the strong connections between the succinctness of the software's description and its simplicity, maintainability, reliability, analysis and verification. We focused on three basic and common idioms — requesting, blocking and waiting for events. We began by analyzing the succinctness of the three idioms taken together, showing that the $\mathcal{RWB}$ model can be translated into cooperating automata with only a polynomial increase in size, but that the converse translation might incur an exponential blowup. Hence, the $\mathcal{RWB}$ model, in which components cannot directly query the state of other components, is strictly less succinct than the $\mathcal{C}$ model. We continued by showing that $\mathcal{RWB}$ can nevertheless succinctly perform non-trivial tasks, that its succinctness is independent and additive to that of the $\mathcal{E}$- and $\mathcal{A}$-automata, and that $(\mathcal{E}, \mathcal{A}, \mathcal{RWB})$-automata are triple-exponentially more succinct than DLA — making them in some cases as strong as the more general $(\mathcal{E}, \mathcal{A}, \mathcal{C})$-automata of [5]. This result established a succinctness hierarchy, indicating the succinctness advantages of models like $(\mathcal{E}, \mathcal{RWB})$ and $(\mathcal{A}, \mathcal{RWB})$. These findings show that the $\mathcal{RWB}$ model, which offers stronger encapsulation and has additional software-engineering advantages over $\mathcal{C}$-automata [13], can sometimes retain the succinctness of the more general model.

We then quantified the contribution of the requesting, waiting-for and blocking idioms to the succinctness of $\mathcal{RWB}$ as a whole. We proved that they are each vital to the succinctness of $\mathcal{RWB}$, as the removal of either may cause an exponential blowup in size, and hence that they do not subsume one another.

The contribution of the present work is thus in substantiating formally the advantages for software engineering that the $\mathcal{RWB}$ idioms, in a variety of programming languages, appear to have; and also in gaining insights into the particular tasks for which each of the idioms is particularly useful. One natural future research direction is to study the succinctness afforded by additional idioms for concurrent programming, such as the lock-step progression idiom, by which all components process a triggered event simultaneously. Another direction is to further study the gap in succinctness between $\mathcal{RWB}$ and $\mathcal{C}$-automata; e.g., to characterize additional tasks, besides counting, in which $\mathcal{C}$'s superiority is manifested.

Acknowledgements The research of Harel, Katz, Lampert and Marron was partly supported by an Advanced Research Grant from the ERC under the European Community's 7th Framework Programme (FP7/2007-2013), by an ISF grant, and by the Philip M. Klutznick Fund for Research, the Joachimowicz Fund and the Benoziyo Fund for the Advancement of Science at the Weizmann Institute of Science. The research of Weiss was supported by the Lynn and William Frankel Center for CS at Ben-Gurion University, by a reintegration (IRG) grant under the European Community's FP7 Programme, and by an ISF grant.

References

1 R. Alur and D. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

2 J. Birget. Two-Way Automata and Length-Preserving Homomorphisms. *Mathematical Systems Theory*, 29(3):191–226, 1996.

3 A. Chandra, D. Kozen, and L. Stockmeyer. Alternation. *J. Assoc. Comput. Mach.*, 28(1):114–133, 1981.

4 W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. *J. on Formal Methods in System Design*, 19(1):45–80, 2001.

5 D. Drusinsky and D. Harel. On the Power of Bounded Concurrency I: Finite Automata. *J. Assoc. Comput. Mach.*, 41:517–539, 1994.

6 P. Eugster, P. Felber, R. Guerraoui, and A. Kermarrec. The Many Faces of Publish/Subscribe. *ACM Computing Surveys*, 35(2):114–131, 2003.

7 M. Gordon, A. Marron, and O. Meerbaum-Salant. Spaghetti for the Main Course? Observations on the Naturalness of Scenario-Based Programming. In *Proc. 17th Conf. on Innovation and Technology in Computer Science Education (ITICSE)*, pages 198–203, 2012.

8 D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, 1987.

9 D. Harel, A. Kantor, G. Katz, A. Marron, L. Mizrahi, and G. Weiss. On Composing and Proving the Correctness of Reactive Behavior. In *Proc. 13th Int. Conf. on Embedded Software (EMSOFT)*, pages 1–10, 2013.

10 D. Harel, G. Katz, R. Lampert, A. Marron, and G. Weiss. On the Succinctness of Idioms for Concurrent Programming: Supplementary Material. `http://www.wisdom.weizmann.ac.il/~bprogram/doc/Concur15Sup.pdf`.

11 D. Harel, G. Katz, A. Marron, and G. Weiss. Non-Intrusive Repair of Safety and Liveness Violations in Reactive Programs. *Trans. on Computational Collective Intelligence*, 16:1–33, 2014.

12 D. Harel, G. Katz, A. Marron, and G. Weiss. The Effect of Concurrent Programming Idioms on Verification. In *Proc. 3rd Int. Conf. on Model-Driven Engineering and Software Development (MODELSWARD)*, 2015.

13 D. Harel, A. Marron, and G. Weiss. Behavioral programming. *Comm. Assoc. Comput. Mach.*, 55(7):90–100, 2012.

14 T. Hirst and D. Harel. On the Power of Bounded Concurrency II: Pushdown Automata. *J. Assoc. Comput. Mach.*, 41:540–559, 1994.

15 M. Holzer and M. Kutrib. Descriptional and Computational Complexity of Finite Automata - a Survey. *Information and Computation*, 209(3):456–470, 2011.

16 J. Hromkovič and G. Schnitger. Lower Bounds on the Size of Sweeping Automata. *Journal of Automata, Languages and Combinatorics*, 14(1):23–13, 2009.

17 O. Kupferman, A. Ta-Shma, and M. Vardi. *Counting With Automata*, 1999. Tech. Report.

18 E. Leiss. Succinct Representation of Regular Languages by Boolean Automata. *Theoretical Computer Science*, 13:323–330, 1981.

19 A. Meyer and M. Fischer. Economy of Description by Automata, Grammars, and Formal Systems. In *Proc. 12th Sym. on Switching and Automata Theory (SWAT)*, pages 188–191, 1971.

20 J. Musa. *Software Reliability Engineered Testing*. McGraw-Hill, 1998.

21 D. Parnas. On the Criteria to be used in Decomposing Systems into Modules. *Comm. Assoc. Comput. Mach.*, 15(12):1053–1058, 1972.

22 M. Rabin and D. Scott. Finite Automata and Their Decision Problems. *IBM Journal of Research and Development*, 3(2):114–125, 1959.

23 S. Safra and M. Vardi. On ω-Automata and Temporal Logic. In *Proc. 21st Sym. on Theory of Computing (STOC)*, pages 127–137, 1989.

24 W. Sakoda and M. Sipser. Nondeterminism and the Size of Two Way Finite Automata. In *Proc. 10th Sym. on Theory of Computing (STOC)*, pages 275–286, 1978.