Bounded Context Switching for Valence Systems

Roland Meyer[1], Sebastian Muskalla[1], and Georg Zetzsche[2]

1 **TU Braunschweig,** {roland.meyer, s.muskalla}@tu-bs.de
2 **IRIF (Université Paris-Diderot, CNRS),** zetzsche@irif.fr

Abstract

We study valence systems, finite-control programs over infinite-state memories modeled in terms of graph monoids. Our contribution is a notion of bounded context switching (BCS). Valence systems generalize pushdowns, concurrent pushdowns, and Petri nets. In these settings, our definition conservatively generalizes existing notions. The main finding is that reachability within a bounded number of context switches is in NP, independent of the memory (the graph monoid). Our proof is genuinely algebraic, and therefore contributes a new way to think about BCS.

Digital Object Identifier 10.4230/LIPIcs...

1 Introduction

Bounded context switching (BCS) is an under-approximate verification technique typically applied to safety properties. It was introduced for concurrent and recursive programs [41]. There, a context switch happens if one thread leaves the processor for another thread to be scheduled. The analysis explores the subset of the computations where the number of context switches is bounded by a given constant. Empirically, it was found that safety violations occur within few context switches [39, 37]. Algorithmically, the complexity of the analysis drops from undecidable to NP [41, 22]. The idea received considerable interest from both practice and theory, a detailed discussion of related work can be found below.

Our contribution is a generalization of bounded context switching to programs operating over arbitrary memories. To be precise, we consider valence systems, finite-control programs equipped with a potentially infinite-state memory modeled as a monoid [19, 46, 47]. In valence systems, both the data domain and the operations are represented by monoid elements, and an operation o will change the current memory value m to the product $m \cdot o$. Of course, the monoid has to be given in some representation.

We consider so-called graph monoids that capture the memories commonly found in programs, like stacks, counters, and tapes, but also combinations thereof. A graph monoid is represented by a graph. Each vertex is interpreted as a symbol (say c) on which the operations push (c^+) and pop (c^-) are defined. A computation is a sequence of such operations. The edges of the graph define an independence relation among the symbols that is used to commute the corresponding operations in a computation. To give an example, if c and d are independent, the computation $d^+.c^+.d^-$ acts on two counters c and d and yields the values 1 and 0, respectively. Pushdowns are represented by valence systems over graphs without edges and concurrent pushdowns by complete m-partite graphs (for m stacks). Petri nets yield complete graphs, blind counter systems complete graphs with self-loops on all vertices.

Our definition of context switches concentrates on the memory and does not reference the control flow. This frees us from having to assume a notion of thread, and makes the analysis applicable to sequential programs as well. We define a context switch as two consecutive operations in a computation that act on different and independent (in the above sense) symbols. This conservatively generalizes existing notions and yields intuitive behavior where a notion of context switch is not defined. When modeling concurrent pushdowns, a context

Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

switch indeed corresponds to switching the stack. For Petri nets and blind counter systems, it means switching the counter. Note, however, that the restriction applies to all memories expressible in terms of graph monoids.

Our main result shows that reachability within a bounded number of context switches is in NP, *for all graph monoids*. The result requires a uniform representation for the computations over very different memories. We prove that a computation can always be split into quadratically-many blocks (in the number of context switches) – independent of the monoid. These blocks behave like single operations in that they commute or form inverses (in the given monoid). With this decomposition result, we develop an automata-theoretic approach to checking reachability. A more elaborate explanation of the proof approach can be found in Section 3, where we have the required terminology at hand.

Taking a step back, our approach provides the first algebraic view to context-bounded computation, and hence enriches the tool box so far containing graph-theoretic interpretations and logical encodings of computations. We elaborate on the related work.

Related Work. There are two lines of work on BCS that are closely related to ours in that they apply to various memory structures. Aiswarya [5] and Madhusudan and Parlato [38] define a graph-theoretic interpretation of computations that manipulate a potentially infinite memory. They restrict the analysis to computations where graph-based measures like the split-width or the tree-width are bounded, and obtain general decidability results by reductions to problems on tree automata. The graph interpretation has been applied to multi pushdowns [6], timed systems [8, 9], and has been generalized to controller synthesis [7]. It also gives a clean formulation of existing restrictions and uniformizes the corresponding analysis algorithms, in particular for [41, 29, 30, 33, 27]. Different from under-approximations based on split- or tree-width, we are able to handle counters, even nested within stacks. We cannot handle, however, the queues to which those technique apply. Indeed, our main result is NP-completeness whereas graph-based analyzes may have a higher complexity. Our approach thus applies to an incomparable class of models. Moreover, it contributes an algebraic view to bounded computations that complements the graph-theoretic interpretation.

The second line of related work are reductions of reachability under BCS to satisfiability in existential Presburger arithmetic [22, 26]. Hague and Lin propose an expressive model, concurrent pushdowns communicating via reversal-bounded counters. Their main result is NP-completeness, like in our setting. The model does not admit the free combination of stacks and counters that we support. The way it is presented, we in turn do not handle reversal boundedness, where the counters may change as long as the mode (increasing/decreasing) does not switch too often. Our approach should be generalizable to reversal boundedness by replacing the emptiness test in the free automata reduction of Section 5 by a satisfiability check, using [44]. The details remain to be worked out. Besides providing an incomparable class of models, our approach complements the logical view to computations.

Reductions to existential Presburger arithmetic often restrict the set of computations by an intersection with a bounded language [25], like in [22, 4]. The importance of bounded languages for under-approximation has been observed by Ganty et al. [24, 21].

Besides the above unifying approaches, there has been a body of work on generalizations of BCS, towards exploring a larger set of computations [29, 34, 20, 10, 43, 1] and handling more expressive programming models [30, 12, 27, 14]. An unconventional instantance of the former direction are restrictions to the network topology [13]. As particularly relevant instantiations of the latter, the BCS under-approximation has been applied to programs operating on relaxed memories [11, 3] and programs manipulating data bases [2]

The practical work on BCS concentrated on implementing fast context-bounded analyzes. Sequentialization techniques [42] were successful in bridging the gap between the parallel program at hand and the available tooling, which is often limited to sequential programs. The idea is to translate the BCS instance into a sequential safety verification problem. The first sequentialization for BCS has been proposed in [35], [31] gave a lazy formulation, and [15] a systematic study of when sequentialization can be achieved. The approach now applies to full C-programs [28] and has won the concurrency track in the softare verification competition. Current work is on parallelizing the analysis by further restricting the interleavings and in this way obtaining instances that are easier to solve [40].

Also with the goal of parallelization, recent works study the multi-variate complexity of context-bounded analyzes. While [22, 23] focus on P and NP, [17] studies fixed-parameter tractability, and [18] the fine-grained complexity. The goal of the latter work is to achieve an analysis of comlexity $2^k poly(n)$, with k a parameter and n the input size. Ideally, this analysis could be performed by 2^k independent threads, each solving a poly-time problem.

Our results contribute to related work on valence automata over graph monoids [47]. They have previously been studied with respect to elimination of silent transitions [45], semilinearity of Parikh images [16], decidability of unrestricted reachability [48], and decidability of first-order logic with reachability [19]. See [46] for a general overview.

2 Valence Systems over Graph Monoids

We introduce the basics on graph monoids and valence systems following [47].

Graph Monoids. Let $G = (V, I)$ be an undirected graph, without parallel edges, but possibly with self-loops. This means $I \subseteq V \times V$, which we refer to as the *independence relation*, is symmetric but neither necessarily reflexive nor necessarily anti-reflexive. We use infix notation and write $o_1 \; I \; o_2$ for $(o_1, o_2) \in I$.

To understand how the graph induces a monoid (a memory), think of the nodes $o \in V$ as stack symbols or counters. To each symbol o, we associate two operations, a positive operation o^+ that can be understood as *push* o or *increment* o and a negative operation o^-, *pop* o or *decrement* o. We call $+$ and $-$ the polarity of the operation. By $o^\pm$ we denote an arbitrary element from $\{o^+, o^-\}$. Let $\mathcal{O} = \{o^+, o^- \mid o \in V\}$ denote the set of all operations. We refer to sequences of operations from $\mathcal{O}^*$ as computations. We lift the independence relation to operations by setting ${o_1}^\pm \; I \; {o_2}^\pm$ if $o_1 \; I \; o_2$. We also write $v_1 \; I \; v_2$ for $v_1, v_2 \in \mathcal{O}^*$ if the operations in the computations are pairwise independent, and similar for subsets of operations $\mathcal{O}_1 \; I \; \mathcal{O}_2$ with $\mathcal{O}_1, \mathcal{O}_2 \subseteq \mathcal{O}$.

We obtain the monoid by factorizing the set of all computations. The congruence will identify computations that order independent operations differently. Moreover, it will implement that o^+ followed by o^- should have no effect, like a push followed by a pop. Formally, we define $\cong$ as the smallest congruence (with respect to concatenation) on $\mathcal{O}^*$ containing ${o_1}^\pm.{o_2}^\pm \cong {o_2}^\pm.{o_1}^\pm$ for all $o_1 \; I \; o_2$ and $o^+.o^- \cong \varepsilon$ for all o.

The *graph monoid for graph* G is $\mathbb{M}_G = \mathcal{O}^*/_{\cong}$. For a word $w \in \mathcal{O}^*$, we use $[w]_\mathbb{M} \in \mathbb{M}_G$ to denote its equivalence class. Note that the multiplication is $[u]_\mathbb{M} \cdot [v]_\mathbb{M} = [u.v]_\mathbb{M}$, which is well-defined as $\cong$ is a congruence. The neutral element of $\mathbb{M}_G$ is the equivalence class of the empty word, $1_\mathbb{M} = [\varepsilon]_\mathbb{M}$.

Recall that an element x of a monoid M is called *right-invertible* if there is $y \in M$ such that $x \cdot y = 1_M$. We lift this notation to $\mathcal{O}^*$ by saying that $w \in \mathcal{O}^*$ is *right-invertible* if its equivalence class $[w]_\mathbb{M} \in \mathbb{M}_G$ is.

Valence Systems. Given a graph G, a *valence system* over the graph monoid $\mathbb{M}_G$ is a pair $A = (Q, \rightarrow)$, where Q is a finite set of control states and $\rightarrow \subseteq Q \times (\mathcal{O} \cup \{\varepsilon\}) \times Q$ is a set of transitions. A transition $q_1 \xrightarrow{x} q_2$ is labeled by an operation on the memory. We write $q_1 \rightarrow q_2$ if the label is ε, indicating that no operation is executed. The size of A is $|A| = |{\rightarrow}|$. We use $\mathcal{O}(A)$ to access the set of operations that label transitions in A.

A *configuration* of A is a tuple $(q, w) \in Q \times \mathcal{O}^*$ consisting of a control state and the sequence of storage operations that has been executed so far. We will restrict ourselves to configurations where w is right-invertible. More precisely, in (q, w) a transition $q_1 \xrightarrow{x} q_2$ is *enabled* if $q = q_1$ and $w.x$ is right-invertible. In this case, the transition leads to the new configuration $(q_2, w.x)$, and we write $(q, w) \rightarrow (q_2, w.x)$. A sequence of consecutive transitions is called a *run*.

This restriction to right-invertible configurations is justified by the definition of the *reachability problem* for valence systems. It asks, given a valence system with two states q_{init}, q_{fin}, whether we can reach q_{fin} with neutral memory from q_{init} with neutral memory, i.e. whether there is a run from (q_{init}, ε) to (q_{fin}, w) with $[w]_{\mathbb{M}} = 1_{\mathbb{M}}$. To be able to reach such a configuration (q_{fin}, w) from some configuration (q, w'), w' has to be right-invertible.

Examples. Figure 1 depicts various graphs. The graph monoid of each of these graph models a commonly used storage mechanism, i.e. it represents the behavior of the storage.

(1) Valence systems for this graph are pushdown systems over the stack alphabet $\{a, b, c\}$.
(2) Valence systems for this graph can be seen as concurrent pushdown systems with two stacks, each over a binary alphabet.
(3) Petri nets resp. vector addition systems with four counters/places p_1, p_2, p_3, p_4 can be modeled as valence systems for this graph. Since the valence system labels transitions by single increments or decrements, the transition multiplicities are encoded in unary.
(4) Integer vector addition systems resp. blind counter automata with counters c_1, c_2, c_3 (that may assume negative values) can be seen as valence systems for this graph.

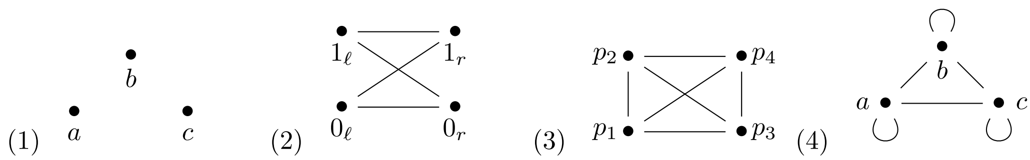

Figure 1 Various examples of graphs representing commonly used storage mechanism.

3 Bounded Context Switching

We introduce a notion of bounded context switching that applies to all valence systems, over arbitrary graph monoids. The idea is to let a new context start with an operation that is independent of the current computation, and hence intuitively belongs to a different thread. We elaborate on the notion of dependence.

We call a set of symbols $V' \subseteq V$ *dependent*, if it does not contain $o_1, o_2 \in V$, $o_1 \neq o_2$ with $o_1 \ I \ o_2$. A set of operations $\mathcal{O}' \subseteq \mathcal{O}$ is dependent if its underlying set of symbols $\{o \mid o^+ \in \mathcal{O}' \text{ or } o^- \in \mathcal{O}'\}$ is. A computation is dependent, if it is over a dependent set of operations. A valence system is said to be dependent, if the operations labeling the transitions form a dependent set.

▶ **Definition 1.** Given $w \in \mathcal{O}^*$, its context decomposition is defined inductively: If w is dependent, w is a single context and does not decompose. Else, the first context w_1 of w is the (non-empty) maximal dependent prefix of w. Then, the context decomposition of w is $w = w_1, \ldots, w_k$, where $w_2, \ldots, w_k$ is the context decomposition of the rest of the word. The number of context switches in w, $cs(w)$, is the number of contexts minus one.

We study reachability under a restricted number of context switches.

Reachability under bounded context switching (BCSREACH)
Given: Valence system A, initial state q_{init}, final state q_{fin}, bound k in unary.
Decide: Is there a run from (q_{init}, ε) to (q_{fin}, w) so that $[w]_{\mathbb{M}} = 1_{\mathbb{M}}$ and $cs(w) \leqslant k$?

In all previously-mentioned graph monoids, the restriction has an intuitive meaning that generalizes existing results. Note, however, that the definition applies to all storage structures expressible in terms of graph monoids, including combinations like stacks of counters.

▶ **Lemma 2.** *(BCSREACH) yields the following restriction:*

(1) On pushdowns, the notion does not incur a restriction.
(2) On concurrent pushdowns, the notion corresponds to changing the stack k-times and hence yields the original definition [41].
(3) On Petri nets and blind counters, the notion corresponds to changing the counter k-times.

Our main result is this.

▶ **Theorem 3.** *(BCSREACH) is in* NP*, independent of the storage graph.*

Note that the NP upper bound matches the lower bound in the case of concurrent pushdowns [32]. We consider the proof technique the main contribution of the paper. Different from existing approaches, which are based on graph interpretations of computations or encodings into Presburger, ours is of algebraic nature. With an algebraic analysis, given in Section 4, we simplify the problem to check whether a given computation reduces to one, $[w]_{\mathbb{M}} = 1_{\mathbb{M}}$. We show that such a reduction exists if and only if the computation admits a decomposition into so-called blocks that reduce to one in a strong sense. There are two surprising aspects about the block decomposition. First, the strong reduction is defined by either commuting two blocks or canceling them if they are inverses. This means the blocks behave like operations, despite being full subcomputations. Second, the decomposition yields only quadratically-many blocks in the number of context switches (important for NP-membership). The block decomposition is the main technical result of the paper.

The second step, presented in Section 5, is a symbolic check for whether a computation exists whose block decomposition admits a strong reduction. We rely on automata-theoretic techniques to implement the operations of a strong reduction. Key is a saturation based on which we give a complete check of whether two automata accept blocks that are inverses.

4 Block Decomposition

In this section, we show how to decompose a computation that reduces to the neutral element into polynomially-many blocks such that the decomposition admits a syntactic reduction to ε. The size of the decomposition will only depend on the number of contexts of the computation and not on its length. This result will later provide the basis for our algorithm.

To be precise, we restrict ourselves to computations with so-called irreducible contexts. In the next section, we will prove that the restriction to this setting is sufficient.

▶ **Definition 4.** We call a computation $w \in \mathcal{O}^*$ *irreducible* if it cannot be written as $w = w'.a.w_I.b.w''$ such that $a = o^+$, $b = o^-$ and o commutes with every symbol in w_I, or $a = o^-, b = o^+, o\ I\ o$ and o commutes with every symbol in w_I.

In other words, a computation is irreducible if we cannot eliminate a pair $o^+.o^-$ after using commutativity. This is in fact the standard definition of irreducibility in the so-called trace monoid, which we do not introduce here.

Our goal is to decompose irreducible contexts such that the decomposition of all contexts in the computation admits a syntactic reduction defined as follows.

▶ **Definition 5** ([36])**.** Let $w_1, w_2, \ldots, w_n$ be a sequence of computations in $\mathcal{O}^*$. A *free reduction* is a finite sequence of applications of the following rewriting rules that transforms $w_1, \ldots, w_n$ into the empty sequence.

(FR1) $\vec{w}', w_i, w_j, \vec{w}'' \mapsto_{free} \vec{w}', \vec{w}''$, applicable if $[w_i.w_j]_{\mathbb{M}} = 1_{\mathbb{M}}$.
(FR2) $\vec{w}', w_i, w_j, \vec{w}'' \mapsto_{free} \vec{w}', w_j, w_i, \vec{w}''$, applicable if $w_i\ I\ w_j$

We call $w_1, w_2, \ldots, w_n$ *freely reducible* if it admits a free reduction.

Being freely reducible is a strictly stronger property than $[w_1.w_2.\ldots.w_n]_{\mathbb{M}} = 1_{\mathbb{M}}$: It means that the sequence can be reduced to $1_{\mathbb{M}}$ by block-wise canceling, Rule (FR1), and swapping whole blocks, Rule (FR2). Indeed, consider ${o_1}^+.{o_2}^+, {o_2}^-, {o_1}^-$ where no two symbols commute. We have $[{o_1}^+.{o_2}^+.{o_2}^-.{o_1}^-]_{\mathbb{M}} = 1_{\mathbb{M}}$, but the sequence is not freely reducible.

The decomposition of a computation w with $[w]_{\mathbb{M}} = 1_{\mathbb{M}}$ into its single operations is always freely reducible. The main result of this section is that for a computation with irreducible contexts, we can always find a freely-reducible decomposition whose length is independent of the length of the computation.

▶ **Theorem 6.** *Let w be a computation with $[w]_{\mathbb{M}} = 1_{\mathbb{M}}$ and let $w = w_1 \ldots w_k$ be its decomposition into irreducible contexts. There is a decomposition of each $w_i = w_{i,1}.w_{i,2} \ldots w_{i,m_i}$ such that $m_i \leqslant k - 1$ and the sequence*

$$w_{1,1}, w_{1,2}, \ldots, w_{1,m_1}, w_{2,1}, w_{2,2}, \ldots, w_{2,m_2}, \ldots, w_{k,1}, w_{k,2}, \ldots, w_{k,m_k}$$

is freely reducible.

Note that the number of words occurring in the decomposition is bounded by k^2. Theorem 6 can be seen as a strengthened version of Lemma 23 from [36]: We use the bound on the number of contexts to obtain a polynomial-size decomposition instead of an exponential one. The proofs of the two results are vastly different.

Constructing a Freely-Reducible Decomposition. The rest of this section will be dedicated to the proof of Theorem 6. Let $w \in \mathcal{O}^*$ be the computation of interest with $[w]_{\mathbb{M}} = 1_{\mathbb{M}}$. We assume that it has length n and $w = w_1 \ldots w_k$ is its decomposition into contexts. For the first part of the proof, we do not require that each w_i is irreducible. As $[w]_{\mathbb{M}} = 1_{\mathbb{M}}$, w can be transformed into ε by finitely often swapping letters and canceling out operations. We formalize this by defining transition rules, similar to the definition of a free reduction.

For the technical development, it will be important to keep track of the original position of each operation in the computation. To this end, we see w as a word over $\mathcal{O} \times \{1, \ldots, n\}$, i.e. we identify the x^{th} operation a of w with the tuple (a, x). For ease of notation, we write $w[x]$ for the x^{th} operation of w. The annotation of letters by their original position will be preserved under the transition rules.

▶ **Definition 7.** A *reduction* of w is a finite sequence of applications of the following rewriting rules that transforms w into into ε.

(R1) $w'.w[x].w[y].w'' \mapsto_{red} w'.w''$, applicable if $w[x] = o^+$, $w[y] = o^-$ for some o.
(R2) $w'.w[x].w[y].w'' \mapsto_{red} w'.w''$, applicable if $w[x] = o^-$, $w[y] = o^+$ for $o \; I \; o$.
(R3) $w'.w[x].w[y].w'' \mapsto_{red} w'.w[y].w[x].w''$, applicable if $w[x] \in {o_1}^{\pm}, w[y] \in {o_1}^{\pm}$ for $o_1 \; I \; o_2$, $o_1 \neq o_2$.

If a word u can be transformed into v using these rules, we write $u \mapsto^*_{red} v$. Note that a reduction of w to ε can be seen as a free reduction of the sequence we obtain by decomposing w into single operations.

▶ **Lemma 8.** *For a word w, we have $[w]_{\mathbb{M}} = 1_{\mathbb{M}}$ iff w admits a reduction.*

Consequently, we may fix a reduction $\pi = w \mapsto^*_{red} \varepsilon$ that transforms w into ε. The following definitions will depend on this fixed π.

▶ **Definition 9.** We define a relation R_π that relates positions of w that cancel in π, i.e.

$$w[x] \; R_\pi \; w[y] \quad \text{if} \quad w'.w[x].w[y].w'' \mapsto_{red} w'.w'' \text{ or } w'.w[y].w[x].w'' \mapsto_{red} w'.w'' \text{ is used in } \pi \, .$$

We lift it to infixes of w by defining inductively

$$t_1 s_1 \; R_\pi \; s_2 t_2 \quad \text{if} \quad \begin{array}{l} \text{there are contexts } w_i = w_{i1}.t_1.s_1.w_{i2} \text{ and } w_j = w_{j1}.s_2.t_2.w_{j2} \\ \text{of } w \text{ such that } s_1 \; R_\pi \; s_2 \text{ and } t_1 \; R_\pi \; t_2 \; . \end{array}$$

An infix u of a context w_i is called a *cluster* if there is an infix u' of a context w_j such that $u \; R_\pi \; u'$. Moreover, if u is a maximal cluster in w_i, then it is called a *block*.

Note that R_π is symmetric by definition. In the following, when we write $s_1 \; R_\pi \; s_2$, we will assume that s_1 appears before s_2 in w, i.e. $w = w'.s_1.w''.s_2.w'''$. We now show that each context has a unique decomposition into blocks. Afterwards, we will see that the resulting block decomposition is the decomposition required by Theorem 6.

▶ **Lemma 10.** *Every context has a unique factorization into blocks.*

Proof. It suffices to show that each position in a context belongs to exactly one block. Since π reduces w to ε, for each $x \in \{1, \ldots, n\}$, there is exactly one $y \in \{1, \ldots, n\}$, $x \neq y$ such that Rule (R1) or Rule (R2) is applied either to $w'.w[x].w[y].w''$ or to $w'.w[y].w[x].w''$. Consequently, each $w[x]$ belongs to at least one block.

We also need to show that no position in w belongs to more than one block. Towards a contradiction, assume there are blocks u, v that overlap, i.e. $u = r.s$, $v = s.t$. Then there is some context w_i of w that we may write as $w_i = w'_i.r.s.t.w''_i$. As u is a block, there is another context w_j such that u cancels with an infix of w_j, i.e. $w_j = w'_j.u'.w''_j$ with $u \; R_\pi \; u'$. By the definition of R_π, we have $u' = s'.r'$ such that $s \; R_\pi \; s'$ and $r \; R_\pi \; r'$.

Similarly, there is a context $w_{\bar{j}}$ containing infix v' which cancels v. As s' is the unique infix of w such that the operations in s cancel out with s', we need to have $\bar{j} = j$, and we can write $w_j = w'_j.t'.s'.r'.w''_j$ where s', r' are as before and $t \; R_\pi \; t'$. Consequently, we have $r.s.t \; R_\pi \; t'.s'.r'$ which contradicts the maximality of the blocks u and v. ◀

We call the unique factorization of a context w_i into blocks the *block decomposition* of w_i (induced by π) and denote it by

$$w_i = w_{i1}, \ldots, w_{im_i} .$$

The *block decomposition* of w (induced by π) is the concatenation of the block decompositions of its contexts,

$$w = w_{1,1}, \ldots, w_{1,m_1}, \ldots, w_{k,1}, \ldots, w_{k,m_k} \ .$$

Note that if u is a block and $u \ R_\pi \ v$, then v is a block as well. Therefore, R_π is a one-to-one correspondence of blocks.

It remains to prove that the block decomposition of w admits a free reduction. We will show that we can inductively cancel out blocks pairwise, starting with an *innermost* pair. Being innermost is formalized by the following relation.

▶ **Definition 11.** We define relation $\leqslant_w$ on R_π-related pairs of blocks by $(s_1 \ R_\pi \ s_2) \leqslant_w (t_1 \ R_\pi \ t_2)$ if $w = w^{(1)}.t_1.w^{(2)}.s_1.w^{(3)}.s_2.w^{(4)}.t_2.w^{(5)}$ for appropriately chosen $w^{(1)}, \ldots, w^{(5)}$. A pair $s_1 \ R_\pi \ s_2$ minimal wrt. this order is called *minimal nesting* in w.

Note that we still assume that all letters are annotated by their position. This means if $w^{(1)}, \ldots, w^{(5)}$ exist, they are uniquely determined.

▶ **Lemma 12.** *$\leqslant_w$ has a minimal nesting.*

The next lemmas state that $s_1 \ R_\pi \ s_2$ implies that s_2 is (a representative of) a right inverse of s_1. Intuitively, π might swap s_1 and s_2, and use s_2 as left inverse.

▶ **Lemma 13.** *If $s_1 \ R_\pi \ s_2$, then $[s_1.s_2]_{\mathbb{M}} = 1_{\mathbb{M}}$.*

▶ **Proposition 14.** *Let $\pi\colon w \rightarrow^*_{red} \varepsilon$ be a reduction of w. The block decomposition of w induced by π is freely reducible.*

Proof. If $w = \varepsilon$, then there is nothing to do. Else, w decomposes into at least two blocks.

We proceed by induction on the number of blocks. In the base case, let us assume that $w = u, v$ is the block decomposition, where $u \ R_\pi \ v$ has to hold. Using Lemma 13, $u, v \xmapsto{\text{(FR1)}}_{\mathit{free}} \varepsilon$ is the desired free reduction.

In the inductive step, we pick a minimal nesting $s_1 \ R_\pi \ s_2$ in w. As argued in Lemma 12, this is always possible. We may write

$$w = w_1 \ldots \underbrace{w_{i_1} s_1 w_{i_2}}_{\text{context } w_i} \ldots \underbrace{w_{j_1} s_2 w_{j_2}}_{\text{context } w_j} \ldots w_k \ .$$

As we have $s_1 \ R_\pi \ s_2$, they cancel each other out in π. Consequently, π has to move each letter from s_1 next the corresponding letter of s_2 or vice versa.

Let us consider the effect of π on the infix $w_{i_2} \ldots w_{j_1}$. Without further arguments, the reduction π could cancel some letters inside this infix, and it can swap the remaining letters with the letters in s_1 or s_2. In fact, there can be no canceling within $w_{i_2} \ldots w_{j_1}$, as $s_1 \ R_\pi \ s_2$ was chosen to be a minimal nesting: Assume that $w_{i_2} \ldots w_{j_1}$ contains some letters a, b with $a \ R_\pi \ b$. Pick the unique blocks u, v to which they belong, and note that we have $(u \ R_\pi \ v) <_w (s_1 \ R_\pi \ s_2)$, i.e. $(u \ R_\pi \ v) \leqslant_w (s_1 \ R_\pi \ s_2)$ and $(u, v) \neq (s_1, s_2)$, a contradiction to the minimality of $s_1 \ R_\pi \ s_2$.

Hence, we have $u = w_{i_2} \ldots w_{j_1}$, and consequently $s_1 \ I \ w_{i_2} \ldots w_{j_1} \ I \ s_2$ We construct a free reduction as follows:

$$\begin{aligned}
& w_1 \ldots w_{i_1} s_1 w_{i_2} w_{i+1} \ldots w_{j-1} w_{j_1} s_2 w_{j_2} \ldots w_k \\
\xmapsto{\text{(FR2)}}{}^{*}_{\mathit{free}} \quad & w_1 \ldots w_{i_1} w_{i_2} w_{i+1} \ldots w_{j-1} w_{j_1} s_1 s_2 w_{j_2} \ldots w_k \\
\xmapsto{\text{(FR1)}}_{\mathit{free}} \quad & w_1 \ldots w_{i_1} w_{i+1} \ldots w_{j-1} w_{j_2} \ldots w_k =: w' \ .
\end{aligned}$$

The applications of Rule (FR2) are valid as $s_1 \ I \ w_{i_2} \dots w_{j_1} \ I \ s_2$ holds. The application of Rule (FR1) to s_1, s_2 is valid by Lemma 13.

Let us denote by w' the result of these reduction steps. We consider the reduction π' that is obtained by restricting π to transitions that work on letters still present in w'. Indeed, π' reduces w' to ε. In particular, for each operation in w', the operation it cancels with is the same in π and π'. Consequently, the relation $R_{\pi'}$ is the restriction of R_π to the operation still occurring in w', and the block decomposition of w' induced by π' is the block decomposition of π minus the blocks s_1, s_2 that have been removed.

We may apply induction to obtain that w' admits a free reduction. We prepend the above reduction steps to this free reduction to obtain the desired reduction for w.

We emphasize the fact that we have not used in the proof that the w_i are contexts. This is important, as the context decompositions of w and w' can differ substantially. Potentially, we have that w consists of four contexts, $w = w_1, s_1, w_2, s_2$, but after canceling s_1 with s_2, w_1 and w_2 merge to a single context, $w' = w_1.w_2$. As we have preserved R_π and its induced block decomposition, this does not hurt the validity of the proof. ◀

A Bound on the Number of Blocks. It remains to prove the desired bound on the number of blocks. To this end, we will exploit that each context w_i is irreducible.

▶ **Proposition 15.** *Let w be a computation with irreducible contexts and $\pi \colon w \to^*_{red} \varepsilon$ a reduction. In the block decomposition of w induced by π, $m_i \leqslant k-1$ holds for all i.*

We prove the proposition in the form of two lemmas.

▶ **Lemma 16.** *The relation R_π never relates blocks from the same context.*

The following lemma allows us to bound the number of blocks in a context by the total number k of contexts.

▶ **Lemma 17.** *For any two contexts w_i and w_j, there is at most one block in w_i that is R_π-related to a block in w_j.*

Proof. Towards a contradiction, assume that some context contains two blocks that are R_π-related to a block from the same context. Let us consider the minimal i such that w_i contains such blocks. Let w_j be the context to which the two blocks are related. By the choice of i, w_i occurs in w before w_j does.

We pick s_1, t_1 as a pair of blocks in w_i canceling with blocks from w_j with minimal distance, i.e. $w_i = w_{i_1} s_1 w_{i_2} t_1 w_{i_3}$ where w_{i_2} contains no block that is canceled by some block in w_j. Let s_2, t_2 be the blocks in w_j such that $s_1 \ R_\pi \ s_2$, $t_1 \ R_\pi \ t_2$. We have to distinguish two cases, depending on the order of occurrence of s_2 and t_2 in w_j. In the first case, we have $w_j = w_{j_1} t_2 w_{j_2} s_2 w_{j_3}$ and thus

$$w = w_1 \dots w_{i-1} \ \underbrace{w_{i_1} s_1 w_{i_2} t_1 w_{i_3}}_{\text{context } w_i} \ w_{i+1} \dots w_{j-1} \ \underbrace{w_{j_1} t_2 w_{j_2} s_2 w_{i_3}}_{\text{context } w_j} \ w_{j+1} \dots w_k \ .$$

Our goal is to show that w_{i_2} and w_{j_2} have to be empty. We then obtain $s_1 t_1 \ R_\pi \ t_2 s_2$, a contradiction to the choice of the blocks as maximal R_π-related infixes in each context.

We start by assuming that w_{i_2} contains some operation b. As π reduces w to ε, w contains some operation c that b cancels with. We first note that c cannot be contained in w_j, as we have chosen s_1, t_1 such that w_{i_2} contains no block that cancels with a block of w_j. Assume that c is contained in the prefix $w_1 \dots w_{i-1} w_{i_1}$. Reduction π either tttttg5 needs to swap b or c with s_1, or it needs to swap s_2 with b (to cancel s_1). In any case, by definition of $\mapsto_{red}$, this means s_1 contains an operation that commutes with b and is distinct from b. However, this is impossible, as s_1 and b are contained in the same context w_i, and contexts do not contain distinct independent symbols. For the same reason, c cannot be contained in the suffix $w_{j_3}w_{j+1}\dots w_k$.

If c is contained in the infix $w_{i+1}\dots w_{j-1}$, π needs to swap b with t_1, or c with t_1, or t_2 with c. In any case, this means t_1 contains an operation that commutes with b and is distinct from b. However, this is impossible, as t_1 and b are contained in the same context w_i, and contexts do not contain distinct independent symbols.

Consequently w_{i_2} needs to be empty. Let us assume that w_{j_2} contains an operation b, and let c denote the operation it cancels with. As for w_{i_2}, we can show that c can neither be contained in the prefix $w_1\dots w_{i-1}w_{i_1}$, nor in the suffix $w_{j_3}w_{j+1}\dots w_k$, nor in the infix $w_{i+1}\dots w_{j-1}$. We conclude that w_{j_2} is also empty and obtain a contradiction to the maximality of the blocks as explained above.

It remains to consider the second case, i.e. $w_j = w_{j_1}s_2w_{j_2}t_2w_{j_3}$ and

$$w = w_1\dots w_{i-1}\ \underbrace{w_{i_1}s_1w_{i_2}t_1w_{i_3}}_{\text{context } w_i}\ w_{i+1}\dots w_{j-1}\ \underbrace{w_{j_1}s_2w_{j_2}t_2w_{i_3}}_{\text{context } w_j}\ w_{j+1}\dots w_k\ .$$

Reduction π either needs to swap s_1 with t_1 or equivalently s_2 with t_1. Again by definition of $\mapsto_{red}$, this means there is an operation a in s_1 and an operation b in t_1 such that $a\ I\ b$ and a, b have distinct symbols. Since s_1, t_1 and s_2, t_2 belong to the same context, this is impossible. ◀

Lemma 16 and Lemma 17 together prove Proposition 15, finishing the proof of Theorem 6.

5 Decision Procedure

Given a valence system A with states q_{init} and q_{fin}, and a bound k, we give an algorithm that checks whether there is a run from (q_{init}, ε) to (q_{fin}, w) such that $[w]_{\mathbb{M}} = 1_{\mathbb{M}}$ and $cs(w) \leqslant k$.

Implementing Irreducibility. The theory we have developed above applies to irreducible contexts. To determine the irreducible versions of contexts in A, we define a saturation operation on valence systems. The algebraic idea behind the saturation is the following.

▶ **Lemma 18.** *Let w be a dependent computation. Then w can be turned into an irreducible computation by applying the following rules: $o^+.o^- \mapsto \varepsilon$ and, provided $o\ I\ o$, $o^-.o^+ \mapsto \varepsilon$.*

To see the lemma, note that in a dependent computation, reducible operations o^+ and o^- cannot be separated by an operation on a different symbol. Hence, o^+ and o^- are placed side by side (potentially after further reductions). If $o\ I\ o$ does not hold, the first rule is sufficient for the reduction. If $o\ I\ o$ does holds, we may find $o^-.o^+$ and need both rules.

The saturation operation implements these two rules. Since Lemma 18 assumes a dependent computation, we consider a dependent valence system $B = (P, \rightsquigarrow)$. The *saturation* is the valence system $sat(B) = (P, \rightsquigarrow_{sat})$ with the same set of control states. The transitions are defined by requiring $\rightsquigarrow \subseteq \rightsquigarrow_{sat}$ and exhaustively applying the following rules:

(1) If $p_1 \overset{o^+}{\rightsquigarrow}_{sat} p \rightsquigarrow^*_{sat} p' \overset{o^-}{\rightsquigarrow}_{sat} p_2$, add an ε-transition $p_1 \rightsquigarrow_{sat} p_2$.

(2) If $p_1 \overset{o^-}{\rightsquigarrow}_{sat} p \rightsquigarrow^*_{sat} p' \overset{o^+}{\rightsquigarrow}_{sat} p_2$ and $o\ I\ o$, add an ε-transition $p_1 \rightsquigarrow_{sat} p_2$.

Here, $p \rightsquigarrow^*_{sat} p'$ denotes that p' is reachable from p by a sequence of ε-transitions.

▶ Remark. In the worst case, we add $|Q|^2$ many transitions.

▶ **Lemma 19.** *There is a computation $(q_1, \varepsilon) \to (q_2, u)$ in B if and only if there is a computation $(q_1, \varepsilon) \to (q_2, v)$ with v irreducible and $u \cong v$ in $sat(B)$.*

The valence system $A = (Q, \to)$ of interest may not be dependent. We will determine dependent versions of it (one for each context) by restricting to a dependent set of operations $\mathcal{O}' \subseteq \mathcal{O}$. The *restriction* is defined by $A[\mathcal{O}'] = (Q, \to \cap (Q \times (\mathcal{O}' \cup \{\varepsilon\}) \times Q))$.

Representing Block Decompositions. Theorem 6 considers a computation decomposed into irreducible contexts w_1 to w_k. It shows that each context w_i can be further decomposed into at most k blocks such that the overall sequence of blocks $w_{1,1}, \dots, w_{k,m_k}$ freely reduces to $1_{\mathbb{M}}$. Our goal is to represent the block decompositions of all candidate computations in a finite way. To this end, we analyze the result more closely.

The decomposition into contexts means there are dependent sets $\mathcal{O}_1, \dots, \mathcal{O}_k \subseteq \mathcal{O}$ such that each context w_i only uses operations from the set $\mathcal{O}_i$. The decomposition into blocks means there are $n = k^2$ computations v_1 to v_n and states q_1 to q_{n-1} such that v_i leads from q_{i-1} to q_i with $q_0 = q_{init}$ and $q_n = q_{fin}$. The last thing to note is that a block itself does not have to be right-invertible. This means we should represent block decompositions by (non-deterministic finite) automata rather than valence systems.

We define, for each pair of states $q_i, q_f \in Q$, each dependent set of operations $\mathcal{O}_{con} \subseteq \mathcal{O}$, and each subset $\mathcal{O}_{bl} \subseteq \mathcal{O}_{con}$ the automaton

$$N(q_i, q_f, \mathcal{O}_{con}, \mathcal{O}_{bl}) = \mathit{2nfa}(q_i, q_f, sat(A[\mathcal{O}_{con}])[\mathcal{O}_{bl}]) \ .$$

Function *2nfa* understands the given valence system $sat(A[\mathcal{O}_{con}])[\mathcal{O}_{bl}]$ as an automaton, with the first parameter as the initial and the second as the final state. The set $\mathcal{O}_{con}$ will be the operations used in the context of interest. As these operations are dependent, $sat(A[\mathcal{O}_{con}])$ will include the irreducible versions of all computations in $A[\mathcal{O}_{con}]$, Lemma 19. The second restriction to $\mathcal{O}_{bl}$ identifies the operations of one block in the context.

With this construction at hand, we define our representation of block decompositions.

▶ **Definition 20.** A *test* for the given (BCSREACH)-instance is a sequence $N_1 \dots N_n$ of $n = k^2$ automata $N_i = N(q_{i-1}, q_i, \mathcal{O}_j, \mathcal{O}_{j,i})$ with $j = \lceil \frac{i}{k} \rceil$, $q_0 = q_{init}$, and $q_n = q_{fin}$.

The following lemma links Theorem 6 and the notion of tests. With Theorem 6, we have to check whether there is a computation w from q_{init} to q_{fin} with $cs(w) \leqslant k$ whose block decomposition admits a free reduction. With the analysis above, such a computation exists iff there is a test N_1 to N_n whose automata accept the blocks in the decomposition.

▶ **Lemma 21.** *We have $(q_{init}, \varepsilon) \to (q_{fin}, w)$ with $cs(w) \leqslant k$ and $[w]_{\mathbb{M}} = 1$ in A iff there is a test N_1 to N_n and computations $v_1 \in L(N_1)$ to $v_n \in L(N_n)$ that freely reduce to $1_{\mathbb{M}}$.*

Determining Free Reducibility. Given a test N_1 to N_n, we have to check whether the automata accept computations that freely reduce to $1_{\mathbb{M}}$. To get rid of the reference to single computations, we now define a notion of free reduction directly on sequences of automata. This means we have to lift the following operations from computations to automata. On computations u and v, a free reduction may check commutativity, $u \ I \ v$, and whether the computations are inverses, $[u]_{\mathbb{M}} \cdot [v]_{\mathbb{M}} = 1_{\mathbb{M}}$. Consider N_u and N_v from N_1 to N_n.

Rather than checking whether N_u and N_v accept computations that commute, the free reduction on automata will check whether the alphabets are independent, $\mathcal{O}(N_u) \ I \ \mathcal{O}(N_v)$.

To see that this yields a complete procedure, note that Lemma 21 existentially quantifies over all tests, and hence all sets of operations to construct N_u and N_v. If there are computations u and v that commute in the free reduction, we can construct the automata N_u and N_v by restricting to the letters in these words. This will still guarantee $u \in L(N_u)$ and $v \in L(N_v)$.

To check whether N_u and N_v accept computations that multiply up to $1_{\mathbb{M}}$, we rely on the syntactic inverse. Consider a computation u that contains negative operations o^- only for symbols with $o\ I\ o$. In this case, the *syntactic inverse* $sinv(u)$ is defined by reversing the letters and inverting the polarity of operations. The operation is not defined otherwise. The following lemma is immediate.

▶ **Lemma 22.** *If $u, v \in \mathcal{O}^*$ are irreducible, dependent with $[u]_{\mathbb{M}} \cdot [v]_{\mathbb{M}} = 1_{\mathbb{M}}$, then $v = sinv(u)$.*

For N_u and N_v, the idea is to admit v as the inverse of u if $v = sinv(u)$ holds. The equality will of course entail that v is the inverse of u, for any pair of computations. Lemma 22 moreover shows that for irreducible, dependent computations the check is complete. Since N_u and N_v are dependent and saturated, it will be complete (Lemma 19) to use the syntactic inverse also on the level of automata.

The definition swaps initial and final state, turns around the transitions, removes the negative operations on non-commutative symbols, and inverts the polarity of the others. Formally, the *syntactic inverse* yields $sinv(N_u) = (Q, q_{u,fin}, remswap(\rightarrow_u^{-1}), q_{u,init})$. The reverse relation contains $(q_2, o^{\pm}, q_1) \in \rightarrow_u^{-1}$ iff $(q_1, o^{\pm} q_2) \in \rightarrow_u$. Function *remswap* removes transitions with operations o^- for which $o\ I\ o$ does not hold and inverts the remaining polarities. The construction guarantees that $sinv(L(N_u)) = L(sinv(N_u))$. With this, the check of whether N_u and N_v contain computations u and v with $v = sinv(u)$ amounts to checking whether N_v and $sinv(N_u)$ have a computation in common.

▶ **Lemma 23.** *There are $u \in L(N_u), v \in L(N_v)$ with $v = sinv(u)$ iff $L(N_v) \cap L(sinv(N_u)) \neq \emptyset$.*

The analogue of the free reduction defined on automata is the following definition.

▶ **Definition 24.** A *free automata reduction* on a test N_1 to N_n is a sequnce of operations

(FRA1) $N_i, N_{i+1} \mapsto_{free} \varepsilon$, if $L(N_{i+1}) \cap L(sinv(N_i)) \neq \emptyset$.
(FRA2) $N_i, N_{i+1} \mapsto_{free} N_{i+1}, N_i$, if $\mathcal{O}(N_i)\ I\ \mathcal{O}(N_{i+1})$.

Since we quantify over all tests, free automata reductions are complete as follows.

▶ **Lemma 25.** *There is a test N_1 to N_n and computations $u_1 \in L(N_1)$ to $u_n \in L(N_n)$ that freely reduce to $1_{\mathbb{M}}$ iff there is a test N_1 to N_n that admits a free automata reduction to ε.*

Together, the Lemma 21 and Lemma 25 yield a decision procedure for (BCSREACH). We guess a suitable test and for this test a suitable free automata reduction. The restrictions, the saturation, the automata conversion, and the independence and disjointness tests require time polynomial in $|A| + k$. Moreover, the free automata reduction contains polynomially-many (in k) steps. Together, this yields membership in NP and proves Theorem 3.

References

1 P. A. Abdulla, C. Aiswarya, and M. F. Atig. Data multi-pushdown automata. In *CONCUR*, volume 85 of *LIPIcs*, pages 38:1–38:17. Dagstuhl, 2017.

2 P. A. Abdulla, C. Aiswarya, M. F. Atig, M. Montali, and O. Rezine. Recency-bounded verification of dynamic database-driven systems. In *PODS*, pages 195–210. ACM, 2016.

3 P. A. Abdulla, M. F. Atig, A. Bouajjani, and T. P. Ngo. Context-bounded analysis for POWER. In *TACAS*, volume 10206 of *LNCS*, pages 56–74. Springer, 2017.

4 P. A. Abdulla, M. F. Atig, R. Meyer, and M. S. Salehi. What's decidable about availability languages? In *FSTTCS*, volume 45 of *LIPIcs*, pages 192–205. Dagstuhl, 2015.

5 C. Aiswarya. *Verification of communicating recursive programs via split-width*. PhD thesis, École normale supérieure de Cachan, France, 2014.

6 C. Aiswarya, P. Gastin, and K. N. Kumar. MSO decidability of multi-pushdown systems via split-width. In *CONCUR*, volume 7454 of *LNCS*, pages 547–561. Springer, 2012.

7 C. Aiswarya, P Gastin, and K. N. Kumar. Controllers for the verification of communicating multi-pushdown systems. In *CONCUR*, volume 8704 of *LNCS*, pages 297–311. Springer, 2014.

8 S. Akshay, P. Gastin, and S. N. Krishna. Analyzing timed systems using tree automata. In *CONCUR*, volume 59 of *LIPIcs*, pages 27:1–27:14. Dagstuhl, 2016.

9 S. Akshay, P. Gastin, S. N. Krishna, and I. Sarkar. Towards an efficient tree automata based technique for timed systems. In *CONCUR*, volume 85 of *LIPIcs*, pages 39:1–39:15. Dagstuhl, 2017.

10 M. F. Atig, A. Bouajjani, K. N. Kumar, and P. Saivasan. On bounded reachability analysis of shared memory systems. In *FSTTCS*, volume 29 of *LIPIcs*, pages 611–623. Dagstuhl, 2014.

11 M. F. Atig, A. Bouajjani, and G. Parlato. Getting rid of store-buffers in TSO analysis. In *CAV*, volume 6806 of *LNCS*, pages 99–115. Springer, 2011.

12 M. F. Atig, A. Bouajjani, and S. Qadeer. Context-bounded analysis for concurrent programs with dynamic creation of threads. In *TACAS*, volume 5505 of *LNCS*, pages 107–123. Springer, 2009.

13 M. F. Atig, A. Bouajjani, and T. Touili. On the reachability analysis of acyclic networks of pushdown systems. In *CONCUR*, volume 5201 of *LNCS*, pages 356–371. Springer, 2008.

14 A. Bouajjani and M. Emmi. Bounded phase analysis of message-passing programs. *STTT*, 16(2):127–146, 2014.

15 A. Bouajjani, M. Emmi, and G. Parlato. On sequentializing concurrent programs. In *SAS*, volume 6887 of *LNCS*, pages 129–145. Springer, 2011.

16 P. Buckheister and Georg Zetzsche. Semilinearity and context-freeness of languages accepted by valence automata. In *MFCS*, volume 8087 of *LNCS*, pages 231–242. Springer, 2013.

17 P. Chini, J. Kolberg, A. Krebs, R. Meyer, and P. Saivasan. On the complexity of bounded context switching. In *ESA*, volume 87 of *LIPIcs*, pages 27:1–27:15. Schloss Dagstuhl, 2017.

18 P. Chini, R. Meyer, and P. Saivasan. Fine-grained complexity of safety verification. In *TACAS*, volume 87 of *LNCS*. Springer, 2018.

19 E. D'Osualdo, R. Meyer, and G. Zetzsche. First-order logic with reachability for infinite-state systems. In *LICS*, pages 457–466. ACM, 2016.

20 M. Emmi, S. Qadeer, and Z. Rakamaric. Delay-bounded scheduling. In *POPL*, pages 411–422. ACM, 2011.

21 J. Esparza, P. Ganty, and R. Majumdar. A perfect model for bounded verification. In *LICS*, pages 285–294. IEEE, 2012.

22 J. Esparza, P. Ganty, and T. Poch. Pattern-based verification for multithreaded programs. *ACM ToPLaS*, 36(3):9:1–9:29, 2014.

23 F. Furbach, R. Meyer, K. Schneider, and M. Senftleben. Memory-model-aware testing: A unified complexity analysis. *ACM Trans. Embedded Comput. Syst.*, 14(4):63:1–63:25, 2015.

24 P. Ganty, R. Majumdar, and B. Monmege. Bounded underapproximations. In *CAV*, volume 6174 of *LNCS*, pages 600–614. Springer, 2010.

25 S. Ginsburg and E. Spanier. Bounded ALGOL-like languages. *Trans. Amer. Math. Soc.*, 113:333–368, 1964.

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

27 A. Heussner, J. Leroux, A. Muscholl, and G. Sutre. Reachability analysis of communicating pushdown systems. *LMCS*, 8(3), 2012.

28 O. Inverso, T. L. Nguyen, B. Fischer, S. La Torre, and G. Parlato. Lazy-CSeq: A context-bounded model checking tool for multi-threaded C-programs. In *ASE*, pages 807–812. IEEE, 2015.

29 S. La Torre, P. Madhusudan, and G. Parlato. A robust class of context-sensitive languages. In *LICS*, pages 161–170. IEEE, 2007.

30 S. La Torre, P. Madhusudan, and G. Parlato. Context-bounded analysis of concurrent queue systems. In *TACAS*, volume 4963 of *LNCS*, pages 299–314. Springer, 2008.

31 S. La Torre, P. Madhusudan, and G. Parlato. Reducing context-bounded concurrent reachability to sequential reachability. In *CAV*, volume 5643 of *LNCS*, pages 477–492. Springer, 2009.

32 S. La Torre, P. Madhusudan, and G. Parlato. The language theory of bounded context-switching. In *LATIN*, pages 96–107. Springer, 2010.

33 S. La Torre, P. Madhusudan, and G. Parlato. Model-checking parameterized concurrent programs using linear interfaces. In *CAV*, volume 6174 of *LNCS*, pages 629–644. Springer, 2010.

34 S. La Torre and M. Napoli. Reachability of multistack pushdown systems with scope-bounded matching relations. In *CONCUR*, volume 6901 of *LNCS*, pages 203–218. Springer, 2011.

35 A. Lal and T. W. Reps. Reducing concurrent analysis under a context bound to sequential analysis. In *CAV*, volume 5123 of *LNCS*, pages 37–51. Springer, 2008.

36 M. Lohrey and G. Zetzsche. Knapsack in graph groups, HNN-extensions and amalgamated products. In *STACS*, pages 50:1–50:14, 2016.

37 S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *ASPLOS*, pages 329–339, 2008.

38 P. Madhusudan and G. Parlato. The tree width of auxiliary storage. In *POPL*, pages 283–294. ACM, 2011.

39 M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI*, pages 446–455. ACM, 2007.

40 T. L. Nguyen, P. Schrammel, B. Fischer, S. La Torre, and G. Parlato. Parallel bug-finding in concurrent programs via reduced interleaving instances. In *ASE*, pages 753–764. IEEE, 2017.

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

42 S. Qadeer and D: Wu. KISS: Keep it simple and sequential. In *PLDI*, pages 14–24. ACM, 2004.

43 E. Tomasco, O. Inverso, B. Fischer, S. La Torre, and G. Parlato. Verifying concurrent programs by memory unwinding. In *TACAS*, volume 9035 of *LNCS*, pages 551–565. Springer, 2015.

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

45 G. Zetzsche. Silent transitions in automata with storage. In *ICALP*, volume 7966 of *LNCS*, pages 434–445. Springer, 2013.
46 G. Zetzsche. Monoids as storage mechanisms. *Bulletin of the EATCS*, 120:237–249, 2016.
47 G. Zetzsche. *Monoids as Storage Mechanisms*. PhD thesis, Technische Universität Kaiserslautern, 2016.
48 G. Zetzsche. The emptiness problem for valence automata over graph monoids, 2018. To appear in Information and Computation.

A Proofs for Section 4

Proof of Lemma 8. Clearly, $w \mapsto^*_{red} \varepsilon$ implies $[w]_{\mathbb{M}} = 1_{\mathbb{M}}$. We prove the converse using another rewriting relation that has been studied before [47, 45]. Let $u \vdash v$ if either (i) $u = s.o^+.o^-.t$ and $v = s.t$ for some $s, t \in \mathcal{O}^*$ and $o \in \mathcal{O}$ or (ii) $u = s.a.b.t$ and $v = s.b.a.t$ for some $s, t \in \mathcal{O}^*$ and $a \in {o_1}^{\pm}$, $b \in {o_2}^{\pm}$ for some $o_1 \; I \; o_2$. Let $\vdash^*$ be the reflexive transitive closure of $\vdash$. It was shown in [47, 45] that $[w]_{\mathbb{M}} = 1_{\mathbb{M}}$ if and only if $w \vdash^* \varepsilon$.

Let us show by induction on $|w|$ that $w \vdash^* \varepsilon$ implies $w \mapsto^*_{red} \varepsilon$. Consider a sequence of steps witnessing $w \vdash^* \varepsilon$. Let us call a step *undesirable* if we cannot (directly) match it with $\mapsto_{red}$. This means, a step $u \vdash v$ where $u = s.a.b.t$ and $v = s.b.a.t$ with $\{a, b\} = \{o^+, o^-\}$.

If the sequence does not apply an undesirable step, it is already a reduction for w. If such a step does occur, suppose w' is the first word where we apply one: We have $w \mapsto^*_{red} w' = s.a.b.t \vdash s.b.a.t \vdash^* \varepsilon$. Observe that then $[a.b]_{\mathbb{M}} = 1_{\mathbb{M}}$ and hence $[s.t]_{\mathbb{M}} = [s.a.b.t]_{\mathbb{M}} = 1_{\mathbb{M}}$. Since $|s.t| < |w'| \leqslant |w|$, induction yields $s.t \mapsto^*_{red} \varepsilon$ and thus $w \mapsto^*_{red} s.a.b.t \mapsto_{red} s.t \mapsto^*_{red} \varepsilon$. ◀

Proof of Lemma 12. We argue that $\leqslant_w$ is transitive and antisymmetric. As the domain of $\leqslant_w$ is finite, this is sufficient to guarantee that a minimal nesting exists: We may start with an arbitrary pair $s_1 \; R_\pi \; s_2$ and iteratively pick smaller pairs as long as possible. For transitivity, note that $(s_1 \; R_\pi \; s_2) \leqslant_w (t_1 \; R_\pi \; t_2)$ and $(t_1 \; R_\pi \; t_2) \leqslant_w (r_1 \; R_\pi \; r_2)$ implies that we can write $w = w^{(1)}.r_1.w^{(2)}.t_1.w^{(3)}.s_1.w^{(4)}.s_2.w^{(5)}.t_2.w^{(6)}.r_2.w^{(7)}$, proving $(s_1 \; R_\pi \; s_2) \leqslant_w (r_1 \; R_\pi \; r_2)$. For antisymmetry, assume $(s_1 \; R_\pi \; s_2) \leqslant_w (t_1 \; R_\pi \; t_2)$ and $(s_1 \; R_\pi \; s_2) \leqslant_w (t_1 \; R_\pi \; t_2)$. This implies that we can write $w = w^{(1)}.s_1.w^{(2)}.t_1.w^{(3)}.s_1.w^{(4)}.s_2.w^{(5)}.t_2.w^{(6)}.s_2.w^{(7)}$, a contradiction to the fact that s_1 has a unique occurrence in w. ◀

Proof of Lemma 13. We proceed by induction on $|s_1| = |s_2|$. In the base case, $s_1 = a$ and $s_2 = b$ are single operations. If $a = o^+$, $b = o^-$ for some o, the statement obviously holds. Otherwise, we have $a = o^-$, $b = o^+$. By definition of $\mapsto_{red}$, this implies that $o \; I \; o$ holds, and $[o^-.o^+]_{\mathbb{M}} = [o^+.o^-]_{\mathbb{M}} = 1_{\mathbb{M}}$ follows as desired.

Assume that $s_1 = u.t$, $s_2 = r.v$ such that $u \; R_\pi \; v$, $t \; R_\pi \; r$. We may apply induction and use that $\cong$ is a congruence, obtaining $[s_1.s_2]_{\mathbb{M}} = [u.t.r.v]_{\mathbb{M}} = [u.v]_{\mathbb{M}} = 1_{\mathbb{M}}$. ◀

Proof of Lemma 16. We show that a context cannot contain two operations that cancel out. As two blocks that cancel out would contain such operations, this is sufficient.

Towards a contradiction, assume $a \; R_\pi \; b$ where a, b are contained in the same context w_i, i.e. we have $w_i = w_{i_1}.a.w_{i_2}.b.w_{i_3}$. We have $a = o^+, b = o^-$, or $a = o^-, b = o^+$ and $o \; I \; o$. If $w_{i_2} = \varepsilon$, we obtain a contradiction to the assumption that w_i is irreducible in both cases. If w_{i_2} is a sequence of operations from $o^{\pm}$, we also obtain a contradiction to irreducibility. Otherwise w_i contains some operation $c \in {o_2}^{\pm}$ for $o_2 \neq o$. Since w_i is a context, o_2 and o are not independent. Since a should cancel with b, π needs to swap one of them over c, or it needs to swap the inverse of c (which is also in ${o_2}^{\pm}$) over one of them. As o_2 and o are not independent, this is not possible. We obtain that a cannot cancel with b, a contradiction. ◀

B Proofs for Section 5

Proof of Lemma 22. We show that for dependent and irreducible u, v, $[u]_{\mathbb{M}} \cdot [v]_{\mathbb{M}} = 1_{\mathbb{M}}$ implies $v = \mathit{sinv}(u)$.

We proceed by induction on the length of u. In the base case, we have $u = \varepsilon$, which implies $[v]_{\mathbb{M}} = [\varepsilon]_{\mathbb{M}}$. As v is irreducible and dependent, we have $v = \varepsilon$ as required. Here, we have used that any word that reduces to $1_{\mathbb{M}}$ needs to have a first reduction step, which can only exist if the word is non-irreducible.

Assume that $u = u'.a$. We claim that we can write $v = b.v'$, where $b = \mathit{sinv}(a)$ and $v' = \mathit{sinv}(u')$, which implies $v = \mathit{sinv}(u)$. As we have $[u]_{\mathbb{M}} \cdot [v]_{\mathbb{M}} = 1_{\mathbb{M}}$, v contains an operation canceling a. If this operation is not the very first letter in v, we obtain a contradiction.

Let $a \in {o_1}^{\pm}$ and assume that the first operation in v is in ${o_2}^{\pm}$ for $o_1 \neq o_2$. Since v is dependent, the first operation cannot commute with the inverse of a, a contradiction to $[u]_{\mathbb{M}} \cdot [v]_{\mathbb{M}} = 1_{\mathbb{M}}$. Hence, v starts with a prefix using operations in ${o_2}^{\pm}$ and containing the operation that cancels a.

If $o_1 \; I \; o_1$ does not hold, then we need to have $a = {o_1}^{+}$. If $a = {o_1}^{-}$, we would have that v is not right-invertible (since it is irreducible), a contradiction to the assumption $[u]_{\mathbb{M}} \cdot [v]_{\mathbb{M}} = 1_{\mathbb{M}}$. Having $a = {o_1}^{+}$ implies that the first operation b in v is ${o_1}^{-}$, which is indeed the syntactic inverse of a.

If $o_1 \; I \; o_1$ holds, then we have to consider both cases $a = {o_1}^{-}$ and $a = {o_1}^{+}$. In the first case, we claim that $b = {o_1}^{+}$ has to hold. If v starts with a sequence of ${o_1}^{-}$, and then has an occurrence of ${o_1}^{+}$, we get a contradiction to the irreducibility of v. Similarly, in the second case $a = {o_1}^{+}$, $b = {o_1}^{-}$ has to hold.

Altogether, we have $v = b.v'$ with $b = \mathit{sinv}(a)$. We have

$$1_{\mathbb{M}} = [u.v]_{\mathbb{M}} = [u'.a.b.v']_{\mathbb{M}} = [u']_{\mathbb{M}} \cdot [a.b]_{\mathbb{M}} \cdot [v']_{\mathbb{M}} = [u']_{\mathbb{M}} \cdot 1_{\mathbb{M}} \cdot [v']_{\mathbb{M}} = [u'.v']_{\mathbb{M}} \; .$$

Since u' and v' are still dependent and irreducible, we obtain $v' = \mathit{sinv}(u')$ by induction. We conclude $\mathit{sinv}(u) = \mathit{sinv}(u'.a) = \mathit{sinv}(a).\mathit{sinv}(u') = b.v' = v$ as desired. ◀