


# Bounded Context Switching for Valence Systems

## Roland Meyer

TU Braunschweig, Germany


roland.meyer@tu-braunschweig.de

 <https://orcid.org/0000-0001-8495-671X>

## Sebastian Muskalla

TU Braunschweig, Germany


s.muskalla@tu-braunschweig.de

 <https://orcid.org/0000-0001-9195-7323>

## Georg Zetsche<sup>1</sup>

IRIF (Université Paris-Diderot, CNRS), France

zetsche@irif.fr

 <https://orcid.org/0000-0002-6421-4388>

---

### 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. In addition, we exhibit a class of storage mechanisms for which BCS reachability belongs to P.

### 2012 ACM Subject Classification:

Theory of computation → Parallel computing models,

Theory of computation → Formal languages and automata theory,

Theory of computation → Logic and verification

**Keywords and Phrases:** valence systems, graph monoids, bounded context switching

---

<sup>1</sup> Supported by a fellowship of the Fondation Sciences Mathématiques de Paris and partially funded by the DeLTA project (ANR-16-CE40-0007).

## 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 [49]. There, a context switch happens if one thread leaves the processor for another thread to be scheduled. The analysis explores the subset of 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 [47, 45]. Algorithmically, the complexity of the analysis drops from undecidable to NP [49, 26]. 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 [23, 55, 56]. 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  $d^+ \cdot c^+ \cdot d^-$  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 switch indeed corresponds to switching the stack. For Petri nets and blind counter systems, it means switching the counter. Note, however, that the restriction can be applied 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.

In addition, we investigate the precise complexity of the problem for individual graph monoids. While there are graph monoids for which our problem is NP-complete (such as those corresponding to the setting of concurrent pushdowns), we show that for an important subclass, those induced by transitive forests, the problem can be solved in polynomial time. Moreover, we describe those graph monoids for which the problem is NL-complete.

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 [6] and Madhusudan and Parlato [46] 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 [7], timed systems [9, 10], and has been generalized to controller synthesis [8]. It also gives a clean formulation of existing restrictions and uniformizes the corresponding analysis algorithms, in particular for [49, 36, 37, 40, 31]. 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 analyses 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 [26, 30]. 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 [52]. 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 [29], like in [26, 5]. The importance of bounded languages for under-approximation has been observed by Ganty et al. [28, 25].

Besides the above unifying approaches, there has been a body of work on generalizations of BCS, towards exploring a larger set of computations [36, 41, 24, 12, 51, 2] and handling more expressive programming models [37, 14, 31, 16]. An unconventional instance of the former direction are restrictions to the network topology [15]. As particularly relevant instantiations of the latter, the BCS under-approximation has been applied to programs operating on relaxed memories [13, 4] and programs manipulating data bases [3].

The practical work on BCS concentrated on implementing fast context-bounded analyses. Sequentialization techniques [50] 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 [42], [38] gave a lazy formulation, and [17] a systematic study of when sequentialization can be achieved. The approach now applies to full C-programs [33] and has won the concurrency track in the software 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 [48].

Also with the goal of parallelization, recent works study the multi-variate complexity of context-bounded analyses. While [26, 27] focus on P and NP, [20] studies fixed-parameter

tractability, and [21] the fine-grained complexity. The goal of the latter work is to achieve an analysis of complexity  $2^k \text{poly}(n)$ , 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 a line of work on valence systems over graph monoids [56]. They have previously been studied with respect to elimination of silent transitions [54], semi-linearity of Parikh images [19], decidability of unrestricted reachability [57], and decidability of first-order logic with reachability [23]. See [55] for a general overview.

## 2 Valence Systems over Graph Monoids

We introduce the basics on graph monoids and valence systems following [56].

**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. 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  $\varepsilon$ ,  $1_{\mathbb{M}} = [\varepsilon]_{\mathbb{M}}$ .

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  $\varepsilon$ , 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  $A$ .

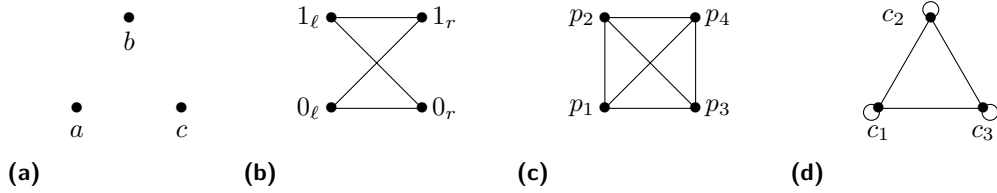
A *configuration* of  $A$  is a tuple  $(q, w) \in Q \times \mathcal{O}^*$  consisting of a control state and the sequence of storage operations that has been executed. We will restrict ourselves to configurations where  $w$  is right-invertible. More precisely, in  $(q, w)$  a transition  $q_1 \xrightarrow{x}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

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  $(q_{init}, \varepsilon)$  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.

- (a) Valence systems for this graph are pushdown systems over the stack alphabet  $\{a, b, c\}$ .
- (b) Valence systems for this graph can be seen as concurrent pushdown systems with two stacks, each over a binary alphabet.
- (c) 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.
- (d) 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.

**What about Queues?** Let us quickly comment on why it is hard to fit queues into this framework. An appealing aspect of valence automata over graph monoids is that by using the monoid identity as the target for reachability problems (resp. as an acceptance condition [19, 54, 56, 57]), we can realize a range of storage mechanisms by only varying the underlying monoid. This is because in the mechanisms that we can realize, the actions (or compositions of actions) that transform the empty storage into the empty storage are precisely those that equal the identity transformation.

In order to keep this aspect, we would need to construct a monoid whose generators can be interpreted as queue actions so that a sequence of generators transforms the empty queue into the empty queue if and only if this sequence evaluates to the identity of the monoid. This, however, is not possible: Suppose that  $a$  and  $b$  represent enqueue operations and that  $\bar{a}$  and  $\bar{b}$  are the corresponding dequeue operations. Each of the action sequences  $a.\bar{a}$  and  $b.\bar{b}$  transforms the empty queue into the empty queue, but  $a.b.\bar{b}.\bar{a}$  does not (it is undefined on the empty queue). Hence, in the monoid, we would want to have  $a\bar{a} = 1$ ,  $b\bar{b} = 1$ , but  $a\bar{b}\bar{a} \neq 1$ , which violates associativity. Hence, although it is possible to model queue behavior in a monoid [32, 34, 35], one would need a different target element (or set).

### 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  $o_1 \perp o_2$ . 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 3.1.** 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  $w = w_1, \dots, w_k$ , where  $w_2, \dots, w_k$  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. For technical reasons, it will be convenient to define  $cs(\varepsilon) = -1$ .

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  $(q_{init}, \varepsilon)$  to  $(q_{fin}, w)$  so that  $[w]_{\mathbb{M}} = 1_{\mathbb{M}}$  and  $cs(w) \leq k$ ?

In all abovementioned graph monoids, the restriction has an intuitive meaning that generalizes existing results. Using the finite states, our notion of BCS also permits a finite shared memory among the threads. In addition, our definition applies to all storage structures expressible in terms of graph monoids, including combinations like stacks of counters.

► **Lemma 3.2.** (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 [49].*
- (3) *On Petri nets and blind counters, the notion corresponds to changing the counter  $k$ -times.*

Our main result is this.

► **Theorem 3.3.** (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 [39]. 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 of checking 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  $\varepsilon$ . 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 4.1.** 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 4.2** ([44]). Let  $w_1, w_2, \dots, w_n$  be a sequence of computations in  $\mathcal{O}^*$ . A *free reduction* is a finite sequence of applications of the following rewriting rules to consecutive entries of the sequence that transforms  $w_1, \dots, w_n$  into the empty sequence.

(FR1)  $w_i, w_j \mapsto_{free} \varepsilon$ , applicable if  $[w_i.w_j]_{\mathbb{M}} = 1_{\mathbb{M}}$ .

(FR2)  $w_i, w_j \mapsto_{free} w_j, w_i$ , applicable if  $w_i I w_j$

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

Being freely reducible is a strictly stronger property than  $[w_1.w_2.\dots.w_n]_{\mathbb{M}} = 1_{\mathbb{M}}$ : 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 4.3.** *Let  $w$  be a computation with  $[w]_{\mathbb{M}} = 1_{\mathbb{M}}$  and let  $w = w_1 \dots w_k$  be its decomposition into irreducible contexts. There is a decomposition of each  $w_i = w_{i,1}.w_{i,2} \dots w_{i,m_i}$  such that  $m_i \leq k - 1$  and the sequence*

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

*is freely reducible.*

Note that the number of words occurring in the decomposition is bounded by  $k^2$ . Theorem 4.3 can be seen as a strengthened version of Lemma 3.10 from [44]: We use the bound on the number of contexts to obtain a polynomial-size decomposition instead of an exponential one. However, 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 4.3. 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  $w = w_1 \dots w_k$  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  $\varepsilon$  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  $\mathcal{O} \times \{1, \dots, n\}$ , i.e. we identify the  $x^{\text{th}}$  operation  $a$  of  $w$  with the tuple  $(a, x)$ . For ease of notation, we write  $w[x]$  for the  $x^{\text{th}}$  operation of  $w$ . The annotation of letters by their original position will be preserved under the transition rules.

► **Definition 4.4.** A *reduction* of  $w$  is a finite sequence of applications of the following rewriting rules that transforms  $w$  into  $\varepsilon$ .

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

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

► **Lemma 4.5.** For a word  $w$ , we have  $[w]_{\mathbb{M}} = 1_{\mathbb{M}}$  iff  $w$  admits a reduction.

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

► **Definition 4.6.** We define a relation  $R_\pi$  that relates positions of  $w$  that cancel in  $\pi$ , i.e.

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

We lift it to infixes of  $w$  by defining inductively

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

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  $R_\pi$  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 4.3.

► **Lemma 4.7.** Every context has a unique factorization into blocks.

To prove the lemma, we show that each position belongs to at least one block and to at most one block. We call the unique factorization of a context  $w_i$  into blocks the *block decomposition* of  $w_i$  (induced by  $\pi$ ) and denote it by

$$w_i = w_{i,1}, \dots, w_{i,m_i}.$$



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

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

Note that if  $u$  is a block and  $u R_\pi v$ , then  $v$  is a block as well. Therefore,  $R_\pi$  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 4.8.** We define relation  $\leq_w$  on  $R_\pi$ -related pairs of blocks by  $(s_1 R_\pi s_2) \leq_w (t_1 R_\pi t_2)$  if  $w = w^{(1)}.t_1.w^{(2)}.s_1.w^{(3)}.s_2.w^{(4)}.t_2.w^{(5)}$  for appropriately chosen  $w^{(1)}, \dots, w^{(5)}$ . 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  $w^{(1)}, \dots, w^{(5)}$  exist, they are uniquely determined.

► **Lemma 4.9.**  $\leq_w$  has a minimal nesting.

The next lemma states that  $s_1 R_\pi s_2$  implies that  $s_2$  is (a representative of) a right inverse of  $s_1$ . While we already know that the operations in  $s_1$  cancel with those in  $s_2$ , it could ostensibly be the case that  $[s_2]_{\mathbb{M}}$  is a left-inverse to  $[s_1]_{\mathbb{M}}$ .

► **Lemma 4.10.** If  $s_1 R_\pi s_2$ , then  $[s_1.s_2]_{\mathbb{M}} = 1_{\mathbb{M}}$ .

► **Proposition 4.11.** Let  $\pi: w \xrightarrow{*}_{red} \varepsilon$  be a reduction of  $w$ . The block decomposition of  $w$  induced by  $\pi$  is freely reducible.

**Proof.** If  $w = \varepsilon$ , then there is nothing to do. Otherwise,  $w$  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 4.10,  $u, v \xrightarrow{(FR1)}_{free} \varepsilon$  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 4.9, this is always possible. We may write

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

Since  $s_1 R_\pi s_2$ , we know that by definition of  $R_\pi$ ,  $\pi$  has to move each letter from  $s_1$  next to the corresponding letter of  $s_2$  or vice versa.

Let us consider the effect of  $\pi$  on the infix  $w_{i_2} \dots w_{j_1}$ . Without further arguments, the reduction  $\pi$  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  $w_{i_2} \dots w_{j_1}$ , as  $s_1 R_\pi s_2$  was chosen to be a minimal nesting: Assume that  $w_{i_2} \dots w_{j_1}$  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.  $(u R_\pi v) \leq_w (s_1 R_\pi s_2)$  and  $(u, v) \neq (s_1, s_2)$ , a contradiction to the minimality of  $s_1 R_\pi s_2$ .

Hence, the reductions needs to swap all letters in  $w_{i_2} \dots w_{j_1}$  with  $s_1$  or  $s_2$  and we have  $s_1 I w_{i_2} \dots w_{j_1} I s_2$ . We construct a free reduction as follows:

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

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 4.10.

Let us denote by  $w'$  the result of these reduction steps. We consider the reduction  $\pi'$  that is obtained by restricting  $\pi$  to transitions that work on letters still present in  $w'$ . Indeed,  $\pi'$  reduces  $w'$  to  $\varepsilon$ . In particular, for each operation in  $w'$ , the operation it cancels with is the same in  $\pi$  and  $\pi'$ . Consequently, the relation  $R_{\pi'}$  is the restriction of  $R_\pi$  to the operation still occurring in  $w'$ , and the block decomposition of  $w'$  induced by  $\pi'$  is the block decomposition of  $\pi$  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  $R_\pi$  and its induced block decomposition, this does not hurt the validity of the proof.  $\blacktriangleleft$

**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 4.12.** *Let  $w$  be a computation with irreducible contexts and  $\pi: w \xrightarrow{*}_{red} \varepsilon$  a reduction. In the block decomposition of  $w$  induced by  $\pi$ ,  $m_i \leq k - 1$  holds for all  $i$ .*

We prove the proposition in the form of two lemmas.

► **Lemma 4.13.** *The relation  $R_\pi$  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 4.14.** *For any two contexts  $w_i$  and  $w_j$ , there is at most one block in  $w_i$  that is  $R_\pi$ -related to a block in  $w_j$ .*

**Proof.** Towards a contradiction, assume that some context contains two blocks that are  $R_\pi$ -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

$$w = w_1 \dots w_{i-1} \underbrace{w_{i_1} s_1 w_{i_2} t_1 w_{i_3}}_{\text{context } w_i} w_{i+1} \dots w_{j-1} \underbrace{w_{j_1} t_2 w_{j_2} s_2 w_{j_3}}_{\text{context } w_j} w_{j+1} \dots w_k .$$

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 definition of blocks as maximal  $R_\pi$ -related infixes in each context.

We start by assuming that  $w_{i_2}$  contains some operation  $b$ . As  $\pi$  reduces  $w$  to  $\varepsilon$ ,  $w$  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  $\pi$  either 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  $w_{i+1}\dots w_{j-1}$ ,  $\pi$  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

$$w = w_1\dots w_{i-1} \underbrace{w_{i_1}s_1w_{i_2}t_1w_{i_3}}_{\text{context } w_i} w_{i+1}\dots w_{j-1} \underbrace{w_{j_1}s_2w_{j_2}t_2w_{j_3}}_{\text{context } w_j} w_{j+1}\dots w_k .$$

Reduction  $\pi$  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.  $\blacktriangleleft$

Lemma 4.13 and Lemma 4.14 together prove Proposition 4.12, finishing the proof of Theorem 4.3.

## 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  $(q_{init}, \varepsilon)$  to  $(q_{fin}, w)$  such that  $[w]_{\mathbb{M}} = 1_{\mathbb{M}}$  and  $cs(w) \leq k$ .

**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 5.1.** *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 hold, we may find  $o^-.o^+$  and need both rules.

The saturation operation implements these two rules. Since Lemma 5.1 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  $p_1 \xrightarrow{o^+}_{sat} p \rightsquigarrow^*_{sat} p' \xrightarrow{o^-}_{sat} p_2$ , add an  $\varepsilon$ -transition  $p_1 \rightsquigarrow_{sat} p_2$ .
- (2) If  $p_1 \xrightarrow{o^-}_{sat} p \rightsquigarrow^*_{sat} p' \xrightarrow{o^+}_{sat} p_2$  and  $o I o$ , add an  $\varepsilon$ -transition  $p_1 \rightsquigarrow_{sat} p_2$ .

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

► Remark. In the worst case, we add  $|P|^2$  many transitions.

► **Lemma 5.2.** *There is a computation  $(q_1, \varepsilon) \rightarrow (q_2, u)$  in  $B$  if and only if there is a computation  $(q_1, \varepsilon) \rightarrow (q_2, v)$  with  $v$  irreducible and  $u \cong v$  in  $\text{sat}(B)$ .*

The valence system  $A = (Q, \rightarrow)$  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  $A[\mathcal{O}'] = (Q, \rightarrow \cap (Q \times (\mathcal{O}' \cup \{\varepsilon\}) \times Q))$ .

**Representing Block Decompositions.** Theorem 4.3 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  $q_0 = q_{\text{init}}$  and  $q_n = q_{\text{fin}}$ . 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  $\mathcal{O}_{\text{con}} \subseteq \mathcal{O}$ , and each subset  $\mathcal{O}_{\text{bl}} \subseteq \mathcal{O}_{\text{con}}$  the automaton

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

Function  $\mathcal{Nfa}$  understands the given valence system  $\text{sat}(A[\mathcal{O}_{\text{con}}])[\mathcal{O}_{\text{bl}}]$  as an automaton, with the first parameter as the initial and the second as the final state. The set  $\mathcal{O}_{\text{con}}$  will be the operations used in the context of interest. As these operations are dependent,  $\text{sat}(A[\mathcal{O}_{\text{con}}])$  will include the irreducible versions of all computations in  $A[\mathcal{O}_{\text{con}}]$ , Lemma 5.2. The second restriction to  $\mathcal{O}_{\text{bl}}$  identifies the operations of one block in the context.

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

► **Definition 5.3.** 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  $j = \lceil \frac{i}{k} \rceil$ ,  $q_0 = q_{\text{init}}$ , and  $q_n = q_{\text{fin}}$ .

The following lemma links Theorem 4.3 and the notion of tests. With Theorem 4.3, we have to check whether there is a computation  $w$  from  $q_{\text{init}}$  to  $q_{\text{fin}}$  with  $\text{cs}(w) \leq k$  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 5.4.** *We have  $(q_{\text{init}}, \varepsilon) \rightarrow (q_{\text{fin}}, w)$  with  $\text{cs}(w) \leq k$  and  $[w]_{\mathbb{M}} = 1$  in  $A$  iff there is a test  $N_1, \dots, N_n$  and computations  $v_1 \in \mathcal{L}(N_1)$  to  $v_n \in \mathcal{L}(N_n)$  that freely reduce to  $1_{\mathbb{M}}$ .*

**Determining Free Reducibility.** Given a test  $N_1, \dots, N_n$ , 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  $N_1, \dots, N_n$ .

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 5.4 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  $u \in \mathcal{L}(N_u)$  and  $v \in \mathcal{L}(N_v)$ .

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*  $\text{sinv}(u)$  is defined by reversing the letters and inverting the polarity of operations. The operation is not defined otherwise. The following lemma is immediate.

► **Lemma 5.5.** *If  $u, v \in \mathcal{O}^*$  are irreducible, dependent with  $[u]_{\mathbb{M}} \cdot [v]_{\mathbb{M}} = 1_{\mathbb{M}}$ , then  $v = \text{sinv}(u)$ .*

The idea is to admit  $v$  as the inverse of  $u$  if  $v = \text{sinv}(u)$  holds. The equality will of course entail that  $v$  is the inverse of  $u$ , for any pair of computations. Lemma 5.5 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 5.2) 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  $\text{sinv}(N_u) = (Q, q_{u,\text{fin}}, \text{remswap}(\rightarrow_u^{-1}), q_{u,\text{init}})$ . 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  $\text{sinv}(\mathcal{L}(N_u)) = \mathcal{L}(\text{sinv}(N_u))$ . With this, the check of whether  $N_u$  and  $N_v$  contain computations  $u$  and  $v$  with  $v = \text{sinv}(u)$  amounts to checking whether  $N_v$  and  $\text{sinv}(N_u)$  have a computation in common.

► **Lemma 5.6.** *There are  $u \in \mathcal{L}(N_u), v \in \mathcal{L}(N_v)$  with  $v = \text{sinv}(u)$  iff  $\mathcal{L}(N_v) \cap \mathcal{L}(\text{sinv}(N_u)) \neq \emptyset$ .*

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

► **Definition 5.7.** A *free automata reduction* on a test  $N_1$  to  $N_n$  is a sequence of operations

(FRA1)  $N_i, N_j \mapsto_{\text{free}} \varepsilon$ , if  $\mathcal{L}(N_j) \cap \mathcal{L}(\text{sinv}(N_i)) \neq \emptyset$ .

(FRA2)  $N_i, N_j \mapsto_{\text{free}} N_j, N_i$ , if  $\mathcal{O}(N_i) I \mathcal{O}(N_j)$ .

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

► **Lemma 5.8.** *There is a test  $N_1, \dots, N_n$  and computations  $u_1 \in \mathcal{L}(N_1)$  to  $u_n \in \mathcal{L}(N_n)$  that freely reduce to  $1_{\mathbb{M}}$  iff there is a test  $N_1, \dots, N_n$  that admits a free automata reduction to  $\varepsilon$ .*

Together, Lemma 5.4 and Lemma 5.8 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 3.3.

## 6 Complexity for Fixed Graphs

We have seen that reachability under bounded context switching can always be decided in NP, even if the graph describing the storage mechanism is part of the input. In this section, we study how the complexity of the problem depends on the storage mechanism, i.e. the

graph. We fix the graph  $G$  and consider the problem  $\text{BCSREACH}(G)$ . We will see that for some graphs, the complexity is lower than NP: We exhibit a class of graphs  $G$  for which  $\text{BCSREACH}(G)$  is solvable in polynomial time and we describe those graphs for which the problem is NL-complete. Of course, for any graph  $G$ , the problem  $\text{BCSREACH}(G)$  is NL-hard, because reachability in directed graphs is. In some cases, we also have an NL upper bound.

A loop-free graph is a *clique* if any two distinct vertices are adjacent. By  $G^-$  we denote the graph obtained from  $G$  by removing all self-loops. If  $G^-$  is a clique, then valence systems over  $G$  are systems with access to a fixed number of independent counters, some of which are blind and some of which are partially blind.

► **Theorem 6.1.** *If  $G^-$  is a clique, then  $\text{BCSREACH}(G)$  is NL-complete. Otherwise,  $\text{BCSREACH}(G)$  is P-hard.*

In some cases,  $\text{BCSREACH}$  is P-complete. A loop-free graph is a *transitive forest* if it is obtained from the empty graph using *disjoint union* and *adding a universal vertex*. A universal vertex is a vertex that is adjacent to all other vertices. Adding one means that we take a graph  $G = (V, I)$  and add a new vertex  $v \notin V$  and make it adjacent to every vertex in  $G$ . Hence, we obtain  $(V \cup \{v\}, I \cup \{\{u, v\} \mid u \in V\})$ .

► **Theorem 6.2.** *If  $G^-$  is a transitive forest, then  $\text{BCSREACH}(G)$  is in P.*

In the area of graph monoids, transitive forests are an important subclass. For many decision problems, they characterize those graphs for which the problem becomes decidable [57, 43] or tractable [44]. Intuitively, the storage mechanisms represented by graphs  $G$  where  $G^-$  is a transitive forest are those obtained by *building stacks* and *adding counters*, see [57, 56].

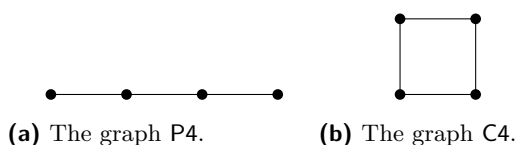
If  $G = (V, I)$  is a graph, then  $H$  is an *induced subgraph* of  $G$  if  $H$  is isomorphic to a graph  $(V', I')$ , where  $V' \subseteq V$  and  $I' = \{e \in I \mid e \subseteq V'\}$ . See Fig. 2 for the graphs C4 and P4.

► **Theorem 6.3.** *If C4 is an induced subgraph of  $G^-$ , then  $\text{BCSREACH}(G)$  is NP-complete.*

It is an old combinatorial result that a simple graph is a transitive forest if and only if it does not contain the two graphs P4 and C4 as induced subgraphs [53]. Hence, if one could also show that  $\text{BCSREACH}(G)$  is NP-hard when  $G^- = \text{P4}$ , then Theorem 6.2 would cover all cases with polynomial complexity (unless  $\text{P} = \text{NP}$ ). However, we currently do not know whether  $\text{BCSREACH}(\text{P4})$  is NP-hard.

**Proof Sketches.** The rest of this section is devoted to sketching the proofs of Theorems 6.1, 6.2, and 6.3. The first step is a reformulation of the problem  $\text{BCSREACH}(G)$  if  $G$  is obtained from two disjoint graphs  $G_0$  and  $G_1$  by drawing edges everywhere between  $G_0$  and  $G_1$ . Suppose  $G_i = (V_i, I_i)$  is a graph for  $i = 0, 1$  such that  $V_0 \cap V_1 = \emptyset$ . Then the graph  $G_0 \times G_1$  is defined as  $(V, I)$ , where  $V = V_0 \cup V_1$  and  $I = I_0 \cup I_1 \cup \{\{v_0, v_1\} \mid v_0 \in V_0, v_1 \in V_1\}$ .

The reformulation also involves valence automata, which can read input. Let  $G = (V, I)$  be a graph and let  $\mathcal{O} = \{o^+, o^- \mid o \in V\}$ . A *valence automaton* over  $G$  is a tuple  $A = (Q, \Sigma, q_0, E, q_f)$ , where  $Q$  is a finite set of *states*,  $\Sigma$  is an alphabet,  $q_0 \in Q$  is its *initial*



■ **Figure 2.** The graphs P4 and C4.

state,  $E \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times (\mathcal{O} \cup \{\varepsilon\}) \times Q$  is its set of *transitions*, and  $q_f \in Q$  is its *final state*. A *configuration* is a tuple  $(q, u, v)$ , where  $q \in Q$ ,  $u \in \Sigma^*$ , and  $v \in \mathcal{O}^*$ , where  $v$  is right-invertible. Intuitively, a transition  $(q, s, w, q')$  changes the state from  $q$  to  $q'$ , reads the input  $s$ , and puts  $w$  into the storage. We write  $(q, u, v) \rightarrow (q', u', v')$  if there is a transition  $(q, s, w, q')$  such that  $u' = us$  and  $v' = vw$ . For any  $k \in \mathbb{N}$ , the *language accepted by  $A$  with at most  $k$  context switches* is denoted  $\mathcal{L}_k(A)$  and defined as the set of all  $u \in \Sigma^*$  such that from  $(q_0, \varepsilon, \varepsilon)$ , we can reach  $(q_f, u, w)$  for some  $w \in \mathcal{O}^*$  with  $[w]_{\mathbb{M}} = 1_{\mathbb{M}}$  and  $cs(w) \leq k$ . The following problem will be used to reformulate  $\text{BCSREACH}(G \times H)$ .

**Intersection under bounded context switching** ( $\text{BCSINT}(G, H)$ )

**Given:** Alphabet  $\Sigma$ , valence automata  $A, B$  over graphs  $G, H$ , resp., with input alphabet  $\Sigma$ , and bounds  $k, \ell, m$  in unary.  
**Decide:** Is the intersection  $\mathcal{L}_k(A) \cap \mathcal{L}_\ell(B) \cap \Sigma^{\leq m}$  non-empty?

We are now ready to state the reformulation, which is not difficult to prove.

► **Proposition 6.4.** *If  $G = G_0 \times G_1$ , then  $\text{BCSREACH}(G)$  is logspace-interreducible with  $\text{BCSINT}(G_0, G_1)$ .*

We can use Proposition 6.4 to show that adding a universal vertex does not change the complexity.

► **Proposition 6.5.** *If  $G$  has a universal vertex  $v$ , then  $\text{BCSREACH}(G)$  reduces to  $\text{BCSREACH}(G \setminus v)$  in logspace.*

This can be deduced from Proposition 6.4 as follows. If  $v$  is a universal vertex, then  $G = (G \setminus v) \times H$ , where  $H$  is a one-vertex graph. In this situation, a valence automaton over  $H$  is equivalent to a one-counter automaton (OCA). It is folklore that an  $n$ -state OCA accepts a word of length  $m$  if and only if it does so with counter values at most  $O((mn)^2)$  [22]. We can thus compute in logspace a finite automaton for the language  $R = \mathcal{L}_\ell(B) \cap \Sigma^{\leq m}$ . This means, our instance of  $\text{BCSINT}(G \setminus v, H)$  reduces to emptiness of  $\mathcal{L}_k(A) \cap R$ . Using the automaton for  $R$ , this is easily turned into an instance of  $\text{BCSREACH}(G \setminus v)$ . Note that Proposition 6.5 yields the upper bound of Theorem 6.1. The P-hardness follows from P-hardness of reachability in pushdown automata.

The P upper bound in Theorem 6.2 follows from Proposition 6.5 and the following.

► **Proposition 6.6.** *If  $\text{BCSREACH}(G_i)$  is in P for  $i = 0, 1$ , then  $\text{BCSREACH}(G_0 \cup G_1)$  is in P as well.*

Proposition 6.6 is shown using a saturation procedure similar to the one in Section 5. In the latter, we shortcut paths that read two (complementary) instructions. Here, in contrast, we find states  $p, q$  between which there is an arbitrarily long path that reads instructions  $w$  over one graph  $G_i$  for  $i = 0, 1$  such that  $[w]_{\mathbb{M}} = 1_{\mathbb{M}}$  and  $cs(w) \leq k$ . Then, we add an  $\varepsilon$ -transition between  $p$  and  $q$ .

Finally, let us comment on the NP-hardness in Theorem 6.3. If  $G = \text{C4}$ , this is the well-known NP-hardness of reachability under bounded context switching. If  $G$  contains self-loops, we employ Proposition 6.4: If  $G^- = \text{C4}$ , then  $G = G_0 \times G_1$  for some graphs where each  $G_i$  contains two non-adjacent vertices. In this case, it is known that that valence automata over  $G_i$  accept the same languages as those over  $G_i^-$  [57, 56]. Therefore, the formulation in terms of  $\text{BCSINT}(G_0, G_1)$  allows us to conclude hardness.

## 7 Conclusion

We have shown that for every storage represented by a graph monoid, reachability under bounded context switches (BCSREACH) is decidable in NP. To this end, we show that after some preprocessing in a saturation procedure, any computation with bounded context switches decomposes into quadratically many blocks. These blocks then cancel and commute with each other so as to reduce to the identity element. Thus, one can guess a decomposition into blocks and verify the cancellation and commutation relations among them.

For the subclass of graph monoids whose underlying simple graph is a transitive forest, we have provided a polynomial-time algorithm (Theorem 6.2). However, we leave open whether there are other graph monoids for which the problem is in P.

One has NP-hardness in the case that the underlying simple graph contains C4 as an induced subgraph, which corresponds to the classical case of bounded context switching in concurrent recursive programs. Since transitive forests are precisely those simple graphs that contain neither C4 nor P4 as induced subgraphs [53], showing NP-hardness for P4 would imply that Theorem 6.2 captures all graphs with polynomial-time algorithms (unless  $P = NP$ ). Unfortunately, the known hardness techniques for problems involving graph groups or Mazurkiewicz traces over P4 [1, 43, 44, 57] do not seem to apply.

Moreover, there is a variety of under-approximations for concurrent recursive programs [36, 11, 18, 41, 24, 12, 51]. It appears to be a promising direction for future research to study generalizations of these under-approximations to valence systems.



---

## References

---

- 1 IJ. J. Aalbersberg and H. J. Hoogeboom. Characterizations of the decidability of some problems for regular trace languages. *Mathematical Systems Theory*, 22(1):1–19, 1989.
- 2 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.
- 3 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.
- 4 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.
- 5 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.
- 6 C. Aiswarya. *Verification of communicating recursive programs via split-width*. PhD thesis, École normale supérieure de Cachan, France, 2014.
- 7 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.
- 8 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.
- 9 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.
- 10 S. Akshay, P. Gastin, S. N. Krishna, and I. Sarker. Towards an efficient tree automata based technique for timed systems. In *CONCUR*, volume 85 of *LIPICs*, pages 39:1–39:15. Dagstuhl, 2017.
- 11 M. F. Atig, B. Bollig, and P. Habermehl. Emptiness of multi-pushdown automata is 2etime-complete. In *DLT*, volume 5257 of *LNCS*, pages 121–133. Springer, 2008.
- 12 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.
- 13 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.
- 14 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.
- 15 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.
- 16 A. Bouajjani and M. Emmi. Bounded phase analysis of message-passing programs. *STTT*, 16(2):127–146, 2014.
- 17 A. Bouajjani, M. Emmi, and G. Parlato. On sequentializing concurrent programs. In *SAS*, volume 6887 of *LNCS*, pages 129–145. Springer, 2011.
- 18 L. Breveglieri, A. Cherubini, C. Citrini, and S. Crespi-Reghizzi. Multi-push-down languages and grammars. *Int. J. Found. Comput. Sci.*, 7(3):253–292, 1996.
- 19 P. Buckheister and Georg Zetsche. Semilinearity and context-freeness of languages accepted by valence automata. In *MFCS*, volume 8087 of *LNCS*, pages 231–242. Springer, 2013.
- 20 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. Dagstuhl, 2017.
- 21 P. Chini, R. Meyer, and P. Saivasan. Fine-grained complexity of safety verification. In *TACAS*, volume 87 of *LNCS*. Springer, 2018.
- 22 D. Chistikov, W. Czerwinski, P. Hofman, M. Pilipczuk, and M. Wehar. Shortest paths in one-counter systems. In *FOSSACS*, pages 462–478, 2016.

- 23 E. D’Osualdo, R. Meyer, and G. Zetsche. First-order logic with reachability for infinite-state systems. In *LICS*, pages 457–466. ACM, 2016.
- 24 M. Emmi, S. Qadeer, and Z. Rakamaric. Delay-bounded scheduling. In *POPL*, pages 411–422. ACM, 2011.
- 25 J. Esparza, P. Ganty, and R. Majumdar. A perfect model for bounded verification. In *LICS*, pages 285–294. IEEE, 2012.
- 26 J. Esparza, P. Ganty, and T. Poch. Pattern-based verification for multithreaded programs. *ACM ToPLaS*, 36(3):9:1–9:29, 2014.
- 27 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.
- 28 P. Ganty, R. Majumdar, and B. Monmege. Bounded underapproximations. In *CAV*, volume 6174 of *LNCS*, pages 600–614. Springer, 2010.
- 29 S. Ginsburg and E. Spanier. Bounded ALGOL-like languages. *Trans. Amer. Math. Soc.*, 113:333–368, 1964.
- 30 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.
- 31 A. Heussner, J. Leroux, A. Muscholl, and G. Sutre. Reachability analysis of communicating pushdown systems. *LMCS*, 8(3), 2012.
- 32 Martin Huschenbett, Dietrich Kuske, and Georg Zetsche. The monoid of queue actions. *Semigroup Forum*, 95:475–508, 2017.
- 33 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.
- 34 C. Köcher. Rational, recognizable, and aperiodic sets in the partially lossy queue monoid. In *STACS*, LIPIcs, pages 45:1–45:14. Dagstuhl, 2018.
- 35 C. Köcher and D. Kuske. The transformation monoid of a partially lossy queue. In *CSR*, volume 10304 of *Lecture Notes in Computer Science*, pages 191–205. Springer, 2017.
- 36 S. La Torre, P. Madhusudan, and G. Parlato. A robust class of context-sensitive languages. In *LICS*, pages 161–170. IEEE, 2007.
- 37 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.
- 38 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.
- 39 S. La Torre, P. Madhusudan, and G. Parlato. The language theory of bounded context-switching. In *LATIN*, pages 96–107. Springer, 2010.
- 40 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.
- 41 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.
- 42 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.
- 43 M. Lohrey and B. Steinberg. The submonoid and rational subset membership problems for graph groups. *Journal of Algebra*, 320(2):728–755, 2008.
- 44 M. Lohrey and G. Zetsche. Knapsack in graph groups. *Theory of Computing Systems*, 62:192–246, 2018.
- 45 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. ACM, 2008.

- 46 P. Madhusudan and G. Parlato. The tree width of auxiliary storage. In *POPL*, pages 283–294. ACM, 2011.
- 47 M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multi-threaded programs. In *PLDI*, pages 446–455. ACM, 2007.
- 48 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.
- 49 S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *TACAS*, volume 3440 of *LNCS*, pages 93–107. Springer, 2005.
- 50 S. Qadeer and D. Wu. KISS: Keep it simple and sequential. In *PLDI*, pages 14–24. ACM, 2004.
- 51 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.
- 52 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.
- 53 E. S. Wolk. A note on "the comparability graph of a tree". *Proceedings of the American Mathematical Society*, 16(1):17–20, 1965.
- 54 G. Zetsche. Silent transitions in automata with storage. In *ICALP*, volume 7966 of *LNCS*, pages 434–445. Springer, 2013.
- 55 G. Zetsche. Monoids as storage mechanisms. *Bulletin of the EATCS*, 120:237–249, 2016.
- 56 G. Zetsche. *Monoids as Storage Mechanisms*. PhD thesis, Technische Universität Kaiserslautern, 2016.
- 57 G. Zetsche. The emptiness problem for valence automata over graph monoids, 2018. To appear in *Information and Computation*.

## A Proofs for Section 4

► **Lemma 4.5.** *For a word  $w$ , we have  $[w]_{\mathbb{M}} = 1_{\mathbb{M}}$  iff  $w$  admits a reduction.*

**Proof.** 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 [56, 54]. 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 [56, 54] 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  $|s.t| < |w'| \leq |w|$ , induction yields  $s.t \mapsto_{red}^* \varepsilon$  and thus  $w \mapsto_{red}^* s.a.b.t \mapsto_{red} s.t \mapsto_{red}^* \varepsilon$ . ◀

► **Lemma 4.7.** *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  $\pi$  reduces  $w$  to  $\varepsilon$ , for each  $x \in \{1, \dots, n\}$ , there is exactly one  $y \in \{1, \dots, n\}$ ,  $x \neq y$  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  $R_\pi$ , 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  $v$ . ◀

► **Lemma 4.9.**  $\leq_w$  has a minimal nesting.

**Proof.** We argue that  $\leq_w$  is transitive and antisymmetric. As the domain of  $\leq_w$  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  $(s_1 R_\pi s_2) \leq_w (t_1 R_\pi t_2)$  and  $(t_1 R_\pi t_2) \leq_w (r_1 R_\pi r_2)$  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  $(s_1 R_\pi s_2) \leq_w (r_1 R_\pi r_2)$ .

For antisymmetry, assume  $(s_1 R_\pi s_2) \leq_w (t_1 R_\pi t_2)$  and  $(s_1 R_\pi s_2) \leq_w (t_1 R_\pi t_2)$ . This implies that we can write  $w = w^{(1)}.s_1.w^{(2)}.t_1.w^{(3)}.s_1.w^{(4)}.s_2.w^{(5)}.t_2.w^{(6)}.s_1.w^{(7)}$ , a contradiction to the fact that  $s_1$  has a unique occurrence in  $w$ . ◀

► **Lemma 4.10.** *If  $s_1 R_\pi s_2$ , then  $[s_1.s_2]_{\mathbb{M}} = 1_{\mathbb{M}}$ .*

**Proof.** 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  $[s_1.s_2]_{\mathbb{M}} = [u.t.r.v]_{\mathbb{M}} = [u.v]_{\mathbb{M}} = 1_{\mathbb{M}}$ .  $\blacktriangleleft$

► **Lemma 4.13.** *The relation  $R_\pi$  never relates blocks from the same context.*

**Proof.** 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  $b$ ,  $\pi$  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.  $\blacktriangleleft$

## B Proofs for Section 5

► **Lemma 5.5.** *If  $u, v \in \mathcal{O}^*$  are irreducible, dependent with  $[u]_{\mathbb{M}} \cdot [v]_{\mathbb{M}} = 1_{\mathbb{M}}$ , then  $v = \text{sinv}(u)$ .*

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

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 in which canceling occurs, 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  $b = \text{sinv}(a)$  and  $v' = \text{sinv}(u')$ , which implies  $v = \text{sinv}(u)$ . 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  $b = \text{sinv}(a)$ . 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  $v' = \text{sinv}(u')$  by induction. We conclude  $\text{sinv}(u) = \text{sinv}(u'.a) = \text{sinv}(a).\text{sinv}(u') = b.v' = v$  as desired.  $\blacktriangleleft$

## C Proofs for Section 6

We begin with the proof of Proposition 6.4, which consists of three lemmas.

► **Lemma C.1.** *If  $G = G_0 \times G_1$ , then  $\text{BCSREACH}(G)$  reduces to  $\text{BCSINT}(G_0, G_1)$  in logspace.*

**Proof.** Suppose  $S = (Q, \rightarrow)$  is a valence system over the graph  $G = G_0 \times G_1$  and we are given the context switching bound  $k$  and states  $q_{init}$  and  $q_{fin}$ . Let  $G_i = (V_i, E_i)$  and  $\mathcal{O}_i = \{o^+, o^- \mid o \in V_i\}$  for  $i = 0, 1$ . Let  $\Sigma = Q \times [0, k] \times Q$ . Let  $q_0 = q_{init}$ ,  $p_{n+1} = q_{fin}$ .

The idea is to construct  $A$  and  $B$  so that a word

$$(p_1, s_1, q_1)(p_2, s_2, q_2) \cdots (p_n, s_n, q_n)$$

in the intersection  $\mathcal{L}_k(A) \cap \mathcal{L}_k(B) \cap \Sigma^{\leq k}$  witnesses a computation

$$q_0 \xrightarrow{u_0} p_1 \xrightarrow{v_1} q_1 \xrightarrow{u_1} \cdots \xrightarrow{u_{n-1}} p_n \xrightarrow{v_n} q_n \xrightarrow{u_n} p_{n+1}$$

in  $S$  where  $u_0, u_n \in \mathcal{O}_0^*$  and  $u_i \in \mathcal{O}_0^+$  for  $i \in [1, n-1]$  and  $v_i \in \mathcal{O}_1^+$  for  $i \in [1, n]$  and  $cs(v_i) = s_i$  for  $i \in [1, n]$  and  $cs(u_0 v_1 u_1 \cdots v_n u_n) \leq k$ . One checks easily that in this case, we have

$$cs(u_0 v_1 u_1 \cdots v_n u_n) = cs(u_0) + \sum_{i=1}^n (cs(v_i) + cs(u_i) + 2) = cs(u_0) + \sum_{i=1}^n (s_i + cs(u_i) + 2).$$

Note that if we can construct  $A$  and  $B$  in logspace so that a word in the intersection  $\mathcal{L}_k(A) \cap \mathcal{L}_k(B) \cap \Sigma^{\leq k}$  exists if and only if there is a computation as above, then we have indeed a logspace reduction from  $\text{BCSREACH}(G)$  to  $\text{BCSINT}(G_0, G_1)$ .

We accomplish this by constructing in logspace valence automata  $A$  and  $B$  over  $G_0$  and  $G_1$ , respectively, for which

$$\mathcal{L}_k(A) = \left\{ (p_1, s_1, q_1)(p_2, s_2, q_2) \cdots (p_n, s_n, q_n) \mid \text{for each } i \in [0, n], \text{ we have } q_i \xrightarrow{u_i} p_{i+1} \right. \\ \left. \text{with } u_0, u_n \in \mathcal{O}_0^*, u_1, \dots, u_{n-1} \in \mathcal{O}_0^+ \text{ and } cs(u_0) + \sum_{i=1}^n (s_i + cs(u_i) + 2) \leq k \right\} \quad (1)$$

and

$$\mathcal{L}_k(B) = \left\{ (p_1, s_1, q_1)(p_2, s_2, q_2) \cdots (p_n, s_n, q_n) \mid \text{for each } i \in [1, n], \text{ we have } p_i \xrightarrow{v_i} q_i \right. \\ \left. \text{for some } v_i \in \mathcal{O}_1^+ \text{ with } cs(v_i) = s_i \text{ and } cs(v_1 \cdots v_n) \leq k \right\}. \quad (2)$$

It is not difficult to construct  $A$  and  $B$ . The state set of  $A$  is  $Q_A = Q \times 2^{\mathcal{O}_0} \times [-1, k]$ . In its right-most component, it counts the number of context switches observed during the run including the ones contributed by edges reading letters in  $\Sigma$ . In order to update this counter, it used its middle component. Here, it keeps track of the set of operations seen in the current context. Note that the state set of  $A$  has polynomial size because we treat the graph  $G$  as constant. The automaton  $A$  has edges for simulating runs  $u_i$  in  $S$  that are labeled  $\varepsilon$ . Moreover, for each  $(p, s, q) \in \Sigma$ ,  $A$  has an edge that changes the left component from  $p$  to  $q$  and adds  $s$  to the counter.

The state set of  $B$  is  $\{*\} \cup Q \times Q \times 2^{\mathcal{O}_1} \times [0, k]$ . Initially,  $B$  is in the state  $*$ . For each  $(p, s, q) \in \Sigma$  and  $p \xrightarrow{x} p'$  for  $x \in \mathcal{O}_1$ ,  $B$  has an edge labeled  $(p, s, q)$  from  $*$  to  $(p', q, \{x\}, s)$ . In states from  $Q \times Q \times 2^{\mathcal{O}_1} \times [0, k]$ ,  $B$  operates analogous to  $A$ , except that the second  $Q$ -component remains constant and the right-most component counts downward. On these

edges,  $B$  reads no input. In addition, from a state  $(q, q, T, 0)$  with  $T \subseteq \mathcal{O}_1$ ,  $B$  can go back to the state  $*$ , which is also its final state.

With  $A$  and  $B$  set up this way, it is clear that (1) and (2) are satisfied. Thus we have indeed that  $q_{init}$  can reach  $q_{fin}$  with at most  $k$  context switches if and only if  $\mathcal{L}_k(A) \cap \mathcal{L}_k(B) \cap \Sigma^{\leq k} \neq \emptyset$ .  $\blacktriangleleft$

In the rest of this section, we will also use the notation  $\mathcal{L}(A) = \bigcup_{k \geq 0} \mathcal{L}_k(A)$  for valence automata  $A$ .

► **Lemma C.2.** *Given a valence automaton  $A$  and a unary bound  $k$ , one can construct in logspace a valence automaton  $A'$  with  $\mathcal{L}(A') = \mathcal{L}_k(A') = \mathcal{L}_k(A)$ .*

**Proof.** Let  $A = (Q, \Sigma, q_0, E, q_f)$ . Then,  $A'$  has states  $\{*\} \cup Q \times 2^{\mathcal{O}} \times [0, k]$ . It simulates computations of  $A$ . In the third component it counts the number of context switches it sees on the storage. In order to maintain this counter, it stores in the second component the set of operations in  $\mathcal{O}$  occurring in the current context. The initial state of  $A'$  is  $(q_0, \emptyset, 0)$  and the final state is  $*$ . In order to reach the final state,  $A'$  has a transition that reads input  $\varepsilon$  and adds  $\varepsilon$  to the storage from every state  $(q_f, U, \ell)$  to  $*$ .  $\blacktriangleleft$

► **Lemma C.3.** *If  $G = G_0 \times G_1$ , then  $\text{BCSINT}(G_0, G_1)$  reduces to  $\text{BCSREACH}(G)$  in logspace.*

**Proof.** Suppose we are given an alphabet  $\Sigma$ , valence automata  $A = (Q_A, \Sigma, q_{0,A}, E_A, q_{f,A})$  and  $B = (Q_B, \Sigma, q_{0,B}, E_B, q_{f,B})$  over  $G_0$  and  $G_1$ , respectively, such that  $\mathcal{L}(A), \mathcal{L}(B) \subseteq \Sigma^*$  and unary bounds  $k, \ell, m \in \mathbb{N}$ . According to Lemma C.2, we may assume that  $\mathcal{L}(A) = \mathcal{L}_k(A)$  and  $\mathcal{L}(B) = \mathcal{L}_\ell(B)$ . Let us call a sequence of transitions in a valence automaton an  $a$ -path if it reads  $a \in \Sigma$  from the input. Hence, it consists of some  $\varepsilon$ -transitions, an  $a$ -transition, and again some  $\varepsilon$ -transitions.

We construct a valence automaton  $C$  over  $G$  as follows. The state set of  $C$  is a product of several components, among them the state sets  $Q_A$  and  $Q_B$ . When  $C$  reads a letter  $a \in \Sigma$ , it first simulates an  $a$ -path of  $A$  and then an  $a$ -path of  $B$ . Moreover, it has a counter that counts the number of read input symbols and makes sure that at most  $m$  of them are read. It is obvious how to set up the state set and transitions of  $C$  appropriately so that for some states  $q_{init}$  and  $q_{fin}$  in  $C$ , we have:  $q_{init} \xrightarrow{w} q_{fin}$  with  $[w]_{\mathbb{M}} = 1_{\mathbb{M}}$  if and only if  $\mathcal{L}(A) \cap \mathcal{L}(B) \cap \Sigma^{\leq m} \neq \emptyset$ . Moreover, since we know that  $\mathcal{L}(A) = \mathcal{L}_k(A)$  and  $\mathcal{L}(B) = \mathcal{L}_\ell(B)$ , we have that  $\mathcal{L}(A) \cap \mathcal{L}(B) \cap \Sigma^{\leq m} \neq \emptyset$  implies  $\mathcal{L}_k(A) \cap \mathcal{L}_\ell(B) \cap \Sigma^{\leq m} \neq \emptyset$  and thus our construction yields  $q_{init} \xrightarrow{w} q_{fin}$  with  $[w]_{\mathbb{M}} = 1_{\mathbb{M}}$  and  $cs(w) \leq k + \ell + 2m$ .  $\blacktriangleleft$

Proposition 6.4 now follows from Lemmas C.1 and C.3.

► **Proposition 6.4.** *If  $G = G_0 \times G_1$ , then  $\text{BCSREACH}(G)$  is logspace-interreducible with  $\text{BCSINT}(G_0, G_1)$ .*

**Proof.** Lemmas C.1 and C.3 show the two reductions.  $\blacktriangleleft$

We are now ready to prove Proposition 6.5.

► **Proposition 6.5.** *If  $G$  has a universal vertex  $v$ , then  $\text{BCSREACH}(G)$  reduces to  $\text{BCSREACH}(G \setminus v)$  in logspace.*

**Proof.** Since  $v$  is universal, we may apply Lemma C.1 in the case where  $G_0 = G \setminus v$  and  $G_1$  is the subgraph of  $G$  induced by  $v$ . Hence, it suffices to show that  $\text{BCSINT}(G_0, G_1)$  reduces to  $\text{BCSREACH}(G \setminus v)$ . Therefore, suppose we are given an alphabet  $\Sigma$ , valence automata  $A$  and  $B$  over  $G_0$  and  $G_1$ , respectively, and bounds  $k, \ell, m$ .

The rest of our proof employs a classical infinite-state model. A *one-counter automaton* is a tuple  $C = (Q, \Gamma, q_0, E, q_f)$ , where  $Q$  is a finite set of *states*,  $\Gamma$  is an alphabet,  $q_0 \in Q$  is its *initial state*,  $q_f \in Q$  is its *final state*, and  $E \subseteq Q \times (\Gamma \cup \{\varepsilon\}) \times \{-1, 1, 0, = 0\} \times Q$  is its set of *transitions*. A *configuration* is a pair  $(q, n) \in Q \times \mathbb{N}$ . The transition relation  $(p, m) \xrightarrow{w} (q, n)$  between configurations  $(p, m)$  and  $(q, n)$  is defined as expected for  $w \in \Gamma^*$ . Here,  $-1, 1, 0, = 0$  stand for *decrement, increment, no change, and zero test*, respectively. A word  $w \in \Gamma^*$  is *accepted* by the automaton if  $(q_0, 0) \xrightarrow{w} (q_f, n)$  for some  $\mathcal{L}(C)$ .

We construct in two steps a one-counter automaton  $C$  with  $\mathcal{L}(C) = \mathcal{L}_\ell(B)$ . First, we use Lemma C.2 to construct a valence automaton  $B'$  with  $\mathcal{L}(B') = \mathcal{L}_\ell(B)$ . Now we can construct  $C$ . Since  $G_1$  contains only one vertex, it is easy to construct in logspace a one-counter automaton  $C$  with  $\mathcal{L}(C) = \mathcal{L}(B')$ : If  $v$  has a loop, then a valence automaton over  $G_1$  is an automaton with access to a counter that assumes values in  $\mathbb{Z}$  and has an increment and a decrement operation. If  $v$  has no loop, then a valence automaton over  $G_1$  is an automaton with a counter that assumes values in  $\mathbb{N}$  and, again, has an increment and a decrement operation. In each case, we can easily simulate the monoid using a counter that assumes only values in  $\mathbb{N}$  and has zero tests available.

Thus, we have reduced our problem to the question of whether  $\mathcal{L}_k(A) \cap \mathcal{L}(C) \cap \Sigma^{\leq m} \neq \emptyset$ . It is folklore that if a one-counter automaton with  $n$  states accepts any word, then it does so in a computation during which the counter values are bounded by  $O(n^2)$  (see [22] and the references therein). This implies that if  $C$  has  $n$  states and accepts a word in  $\Sigma^{\leq m}$ , then it does so in some computation with counter values at most  $a(mn)^2 + b$  for some constants  $a, b \in \mathbb{N}$ .

Let us construct a finite automaton  $D$  that simulates all computations of  $C$  where the counter does not exceed  $a(mn)^2 + b$  and which read an input word of length  $\leq m$ . Then,  $D$  has  $(a(mn)^2 + b)nm$  states and can clearly be constructed in logspace. Moreover, we have  $\mathcal{L}(D) = \mathcal{L}(C) \cap \Sigma^{\leq m}$  and thus  $\mathcal{L}_k(A) \cap \mathcal{L}_\ell(B) \cap \Sigma^{\leq m} \neq \emptyset$  if and only if  $\mathcal{L}_k(A) \cap \mathcal{L}(D) \neq \emptyset$ . Using a simple product construction, we can now obtain a valence automaton  $A'$  from  $A$  and  $D$  so that  $\mathcal{L}_k(A') = \mathcal{L}_k(A) \cap \mathcal{L}(D)$ . Finally, checking emptiness of  $\mathcal{L}_k(A')$  is an instance of  $\text{BCSREACH}(G_0)$ .  $\blacktriangleleft$

Our next goal is to prove Proposition 6.6. For the proof, it will be convenient to use a reformulation of the problem  $\text{BCSREACH}$ . A valence system  $S = (Q, \rightarrow)$  is said to be *k-bounded* if  $p \xrightarrow{w} q$  for states  $p, q \in Q$  implies  $cs(w) \leq k$ .

**BCS reachability with promise** ( $\text{BCSRP}(G)$ )

**Given:** Bound  $k$  in unary, a  $k$ -bounded valence system  $S$ , states  $q_{init}, q_{fin}$ .

**Decide:** Is there a run from  $(q_{init}, \varepsilon)$  to  $(q_{fin}, w)$  so that  $[w]_{\mathbb{M}} = 1_{\mathbb{M}}$ ?

Of course, the problem  $\text{BCSRP}(G)$  is very similar to  $\text{BCSREACH}(G)$ . What makes it useful is that in order to solve  $\text{BCSRP}(G)$  we need not worry about discovering paths with too many context switches: We may assume that all paths are guaranteed to contain at most  $k$ . In particular, our procedure will find paths in a valence system over  $G = G_0 \cup G_1$  by repeatedly finding paths in valence systems over  $G_0$  or  $G_1$ . The formulation in terms of  $\text{BCSRP}(G)$  relieves us from keeping track of how many context switches we have made in the paths for the  $G_i$  when composing them to paths over  $G$ .

**► Lemma C.4.** *For each graph  $G$ , the problems  $\text{BCSREACH}(G)$  and  $\text{BCSRP}(G)$  are inter-reducible via logspace reductions.*



**Proof.** Of course,  $\text{BCSRP}(G)$  reduces trivially to  $\text{BCSREACH}(G)$ . Conversely, suppose  $S$  is a valence system over  $G$  and we want to decide whether there is a run from  $(q_{init}, \varepsilon)$  to  $(q_{fin}, w)$  so that  $[w]_{\mathbb{M}} = 1_{\mathbb{M}}$  and  $cs(w) \leq k$ .

We turn  $S$  into a valence system  $S'$  that keeps track of the number of context switches in its states and is therefore  $k$ -bounded. If  $G = (V, I)$ ,  $\mathcal{O} = \{o^+, o^- \mid o \in V\}$ ,  $S = (Q, \rightarrow)$ , then  $S'$  has states  $Q' = \{*\} \cup Q \times 2^{\mathcal{O}} \times [0, k]$ . In its middle component, it maintains the list of operations occurring in the current context. This is used to update the right-most component, which counts the number of context switches  $S'$  has seen during the computation. Hence, we have transitions:

$$\begin{aligned} (p, U, \ell) &\xrightarrow{x} (q, U \cup \{x\}, \ell) && \text{for each } p \xrightarrow{x} q \text{ in } S \text{ where } U \cup \{x\} \text{ is dependent} \\ &&& \text{and } \ell \in [0, k] \\ (p, U, \ell) &\xrightarrow{x} (q, \{x\}, \ell + 1) && \text{for each } p \xrightarrow{x} q \text{ in } S \text{ where } U \cup \{x\} \text{ is not dependent} \\ &&& \text{and } \ell \in [0, k - 1] \\ (p, U, \ell) &\xrightarrow{\varepsilon} (q, U, \ell) && \text{for each } p \xrightarrow{\varepsilon} q \text{ in } S \end{aligned}$$

Now reachability of  $(q_{fin}, w)$  from  $(q_{init}, \varepsilon)$  with  $[w]_{\mathbb{M}} = 1_{\mathbb{M}}$  and  $cs(w) \leq k$  is equivalent to reachability of some  $((q_{fin}, U, \ell), w)$  from  $((q_{init}, \emptyset, 0), \varepsilon)$  with  $[w]_{\mathbb{M}} = 1_{\mathbb{M}}$ . Since we want to provide a single final state in our reduction, we also add transitions

$$(q_{fin}, U, \ell) \xrightarrow{\varepsilon} * \quad \text{for each } U \subseteq \mathcal{O} \text{ and } \ell \in [0, k]$$

Then  $S'$  is clearly  $k$ -bounded. As initial state, we take  $q'_{init} = (q_{init}, \emptyset, 0)$  and as a final state, we take  $q'_{fin} = *$ . Then we clearly have a run from  $(q'_{init}, \varepsilon)$  to  $(q'_{fin}, w')$  with  $[w']_{\mathbb{M}} = 1_{\mathbb{M}}$  if and only if there is a run from  $(q_{init}, \varepsilon)$  to  $(q_{fin}, w)$  with  $[w]_{\mathbb{M}} = 1_{\mathbb{M}}$  and  $cs(w) \leq k$ .  $\blacktriangleleft$

We are now ready to prove Proposition 6.6.

**► Proposition 6.6.** *If  $\text{BCSREACH}(G_i)$  is in  $\mathsf{P}$  for  $i = 0, 1$ , then  $\text{BCSREACH}(G_0 \cup G_1)$  is in  $\mathsf{P}$  as well.*

**Proof.** According to Lemma C.4, it suffices to show that if  $\text{BCSRP}(G_0)$  and  $\text{BCSRP}(G_1)$  are in  $\mathsf{P}$ , then  $\text{BCSRP}(G_0 \cup G_1)$  is in  $\mathsf{P}$ . To this end, we use a saturation algorithm.

Suppose we are given a  $k$ -bounded valence system  $S = (Q, \rightarrow)$  over  $G = G_0 \cup G_1$ . Let  $G_i = (V_i, I_i)$  and  $\mathcal{O}_i = \{o^+, o^- \mid o \in V\}$  for  $i = 0, 1$ . We add edges labeled  $\varepsilon$  according to the following rule: If from  $(p, \varepsilon)$ , one can reach  $(q, w)$  with  $[w]_{\mathbb{M}} = 1_{\mathbb{M}}$  and  $w \in \mathcal{O}_i^*$  for some  $i \in \{0, 1\}$ , then we add an edge  $p \xrightarrow{\varepsilon} q$ . Note that we can decide in polynomial time whether this is the case: Restricting  $S$  to edges with labels in  $\mathcal{O}_i^*$  yields a valence system that inherits  $k$ -boundedness from  $S$ ; thus, we answer an instance of  $\text{BCSRP}(G_i)$ . Moreover, adding an  $\varepsilon$ -transition preserves  $k$ -boundedness as well. Finally, since we are only adding transitions labeled  $\varepsilon$ , this procedure terminates after adding at most  $|Q|^2$  transitions and thus in polynomial time. Let  $S' = (Q, \rightsquigarrow)$  be the resulting valence system.

We claim that  $q_{init} \xrightarrow{\varepsilon} q_{fin}$  if and only if  $q_{init} \xrightarrow{w} q_{fin}$  for some  $w$  with  $[w]_{\mathbb{M}} = 1_{\mathbb{M}}$ . Here, the “only if” direction follows easily by induction on the number of steps performed in the saturation algorithm.

For the converse, we prove by induction on  $|w|$  that for any  $p, q \in Q$ , if  $p \xrightarrow{w} q$  with  $[w]_{\mathbb{M}} = 1_{\mathbb{M}}$ , then  $p \xrightarrow{\varepsilon} q$ . Suppose we have  $p \xrightarrow{w} q$  with  $[w]_{\mathbb{M}} = 1_{\mathbb{M}}$ .

If  $|w| = 0$ , we are done. Moreover, if  $w \in \mathcal{O}_i^+$  for some  $i \in \{0, 1\}$ , then we have introduced  $p \xrightarrow{\varepsilon} q$  during the saturation. Otherwise, we can decompose  $w = u_1 \cdots u_n$  so that  $u_j \in \mathcal{O}_0^+ \cup \mathcal{O}_1^+$  and  $u_j \in \mathcal{O}_i^+$  iff  $u_{j+1} \in \mathcal{O}_{1-i}^+$  for  $j \in [1, n]$ .

Then it follows from the definition of  $\cong$  that there is some  $\ell \in [1, n]$  with  $u_j \cong \varepsilon$ : Otherwise, it follows by induction on the number of applied equivalences (i.e.  $o_1^\pm . o_2^\pm \cong o_2^\pm . o_1^\pm$  for  $o_1 I o_2$  or  $o^+ o^- \cong \varepsilon$ ), that every word  $w'$  with  $w \cong w'$  has a decomposition  $w' = u'_1 \cdots u'_n$  with  $u'_j \cong u_j$  for  $j \in [1, n]$ . Hence, let  $u_\ell \cong \varepsilon$  and  $w = xu_\ell y$ .

Let  $p', q'$  be states so that  $p \xrightarrow{x} p' \xrightarrow{u_\ell} q' \xrightarrow{y} q$ . Since  $u_\ell \cong \varepsilon$ , the saturation procedure has added the transition  $p' \xrightarrow{\varepsilon} q'$ . Hence, we have  $p \xrightarrow{xy} q$  with  $[xy]_{\mathbb{M}} = [xu_\ell y]_{\mathbb{M}} = [w]_{\mathbb{M}} = 1_{\mathbb{M}}$  and thus by induction  $p \xrightarrow{\varepsilon} q$ .  $\blacktriangleleft$

► **Theorem 6.3.** If C4 is an induced subgraph of  $G^-$ , then  $\text{BCSREACH}(G)$  is NP-complete.

**Proof.** Since we have shown NP membership for all graphs, it suffices to show NP-hardness in the case  $G^- = \text{C4}$ . In that case, we have  $G = G_0 \times G_1$ , where each  $G_i$  consists of two vertices, which may or may not carry self-loops.

According to Proposition 6.4, it suffices to show that  $\text{BCSINT}(G_0, G_1)$  is NP-hard. Consider a 3CNF-SAT instance  $\varphi = \bigwedge_{j=1}^m C_j$ , where each  $C_j$  is a clause over the variables  $\{x_1, \dots, x_n\}$ . We encode an assignment of the variables by a word  $w \in \{0, 1\}^n$  and say that  $w$  satisfies  $C_j$  (resp.  $\varphi$ ) if the corresponding assignment satisfies  $C_j$  (resp.  $\varphi$ ).

Since each  $G_i$  consists of two non-adjacent vertices, it is known that valence automata over  $G_i$  accept exactly the context-free languages [57, 56]. Moreover, the translation from PDAs (or context-free grammars) into valence automata over  $G_i$  can obviously be performed in logarithmic space. For a word  $w \in \{0, 1\}^*$ , let  $w^{\text{rev}}$  denote its reversal. Consider the context-free languages

$$\begin{aligned} \mathcal{L}_0 &= \{w_1 \# w_1^{\text{rev}} \# w_2 \# w_2^{\text{rev}} \cdots w_m \# w_m^{\text{rev}} \# \mid \text{for } j \in [1, m], w_j \in \{0, 1\}^n \text{ satisfies } C_j\} \\ \mathcal{L}_1 &= \{w_0 \# w_1 \# w_1^{\text{rev}} \# \cdots w_{m-1} \# w_{m-1}^{\text{rev}} \# w_m \# \mid \text{for } j \in [0, m], w_j \in \{0, 1\}^n\} \end{aligned}$$

over  $\Sigma = \{0, 1, \#\}$ . On the one hand, we have  $\mathcal{L}_0 \cap \mathcal{L}_1 \cap \Sigma^{2m(n+1)} \neq \emptyset$  if and only if  $\varphi$  is satisfiable. On the other hand, we can construct a PDA for each  $\mathcal{L}_i$  in logarithmic space, and hence also valence automata  $A_i$  over  $G_i$  such that  $\mathcal{L}(A_i) = \mathcal{L}_i$  for  $i = 0, 1$ . Since in  $G_i$ , there are no edges between distinct vertices, every computation has 0 context switches, meaning  $\mathcal{L}(A_i) = \mathcal{L}_0(A_i)$ . Thus,  $\varphi$  is satisfiable if and only if  $\mathcal{L}_0(A_0) \cap \mathcal{L}_0(A_1) \cap \Sigma^{2m(n+1)} \neq \emptyset$ .  $\blacktriangleleft$