Pattern-Based Verification for Multithreaded Programs

JAVIER ESPARZA, Fakultät für Informatik, Technische Universität München, Germany
PIERRE GANTY, IMDEA Software Institute, Madrid, Spain
TOMÁŠ POCH, Charles University Prague, Faculty of Mathematics and Physics, Czech Republic

Pattern-based verification checks the correctness of program executions that follow a given *pattern*, a regular expression over the alphabet of program transitions of the form $w_1^* \dots w_n^*$. For multithreaded programs, the alphabet of the pattern is given by the reads and writes to the shared storage. We study the complexity of pattern-based verification for multithreaded programs with shared counters and finite variables. While unrestricted verification is undecidable for abstracted multithreaded programs with recursive procedures and PSPACE-complete for abstracted multithreaded while-programs (even without counters), we show that pattern-based verification is NP-complete for both classes, even in the presence of counters. We then conduct a multiparameter analysis to study the complexity of the problem on its three natural parameters (number of threads+counters+variables, maximal size of a thread, size of the pattern) and on two parameters related to thread structure (maximal number of procedures per thread and longest simple path of procedure calls). We present an algorithm that for a fixed number of threads, counters, variables, and pattern size solves the verification problem in $st^{O(lsp+\lceil \log(pr+1)\rceil)}$ time, where st is the maximal size of a thread, pr is the maximal number of procedures per thread, and lsp is the longest simple path of procedure calls.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification

General Terms: Algorithms, Languages, Verification, Reliability

Additional Key Words and Phrases: Concurrent programming, context-free languages, multithreaded programs, safety, underapproximation

ACM Reference Format:
Pierre Ganty, Javier Esparza, and Tomáš Poch. 2014. Pattern-based verification for multithreaded programs. ACM Trans. Program. Lang. Syst. 36, 3, Article 9 (August 2014), 29 pages.
DOI: http://dx.doi.org/10.1145/2629644

1. INTRODUCTION

The analysis and verification of multithreaded programs is one of the most active research areas in software model checking. This is due, on the one hand, to the increasing relevance of multicore architectures and, on the other hand, to the difficulty of conceiving, reasoning about, and debugging concurrent software. Automated analysis tools must cope with the very intractable nature of the analysis problems. Multithreaded programs with possibly recursive procedures communicating through global variables are Turing powerful even for programs having only two threads and three variables, all of them Boolean. If communication takes place through message passing, the programs are Turing powerful even after applying the usual program analysis abstraction

A preliminary version of this article appeared in the ACM-SIGPLAN Symposium on the Principles of Programming Languages, 2011.
Authors' addresses: Javier Esparza, TUM, Munich, Germany; email: esparza@in.tum.de; Pierre Ganty, IMDEA Software Institute, Madrid, Spain; email: pierre.ganty@imdea.org; Tomáš Poch, Charles University Prague, Faculty of Mathematics and Physics, Czech Rep.; email: tomas.poch@d3s.mff.cuni.cz.

DOI: http://dx.doi.org/10.1145/2629644

that replaces all conditions in alternative constructs and loops by nondeterminism (see Ramalingam [2000] for more details).

Context bounding, proposed by Qadeer and Rehof [2005], is the most successful proposal to date for overcoming untractability. It restricts the problem further by exploring only those computations with a bounded, fixed number of *contexts*. A *context* is a segment of the computation during which only one thread accesses the global variables; a *context switch* takes place when the identity of this thread changes. Reachability of a program point in a program with d Boolean variables by a computation with at most k context switches (the *context-bounded reachability problem*) is NP-complete when k is given in unary and can be checked by means of an algorithm polynomial in the size of the program and exponential both in k and d [Qadeer and Rehof 2005; Lal et al. 2008; Lal and Reps 2009]. Hague and Lin [2012] extend the approach to the case of recursive multithreaded programs with bounded-reversal counters. Context bounding has been implemented in several model checkers, like CHESS, ZING, KISS, jMoped, and others [Andrews et al. 2004; Qadeer and Wu 2004; Musuvathi and Qadeer 2007; Suwimonteerabuth et al. 2008; Lal and Reps 2009; La Torre et al. 2009], and experiments with these tools have provided evidence that many concurrency errors manifest themselves in computations with few context switches.

While context bounding has been very successful, it also has important limitations; it restricts the number of communication events between threads. While a thread can perform arbitrarily many reads and writes to the global variables during a context, these writes are not observed by the other threads, and so only the values of the variables immediately before the context switch amount to a communication. So in a computation with k context switches, threads communicate at most k times.

In this article, we study a technique, *pattern-based verification*, that allows one to check executions with an arbitrary number of communication events. The starting point is Kahlon's important observation [Kahlon 2009b] that the theory of *bounded languages* developed in the mid-1960s by Ginsburg and Spanier [Ginsburg 1966] can be applied to the verification problem. Kahlon uses the theory to prove decidability of safety analysis for the particular case of multithreaded programs, all of whose executions conform to a *pattern*, a regular expression of the form $w_1^* \ldots w_n^*$ over the alphabet of program instructions. Observe that the executions of such a program can be arbitrarily long. Kahlon leaves the complexity of the safety analysis aside.

In pattern-based verification, we consider general multithreaded programs and use patterns to specify classes of potential executions of the program. It can be automatically verified whether the executions of the program that conform to the pattern satisfy the property of interest. Long et al. [2012] use it as a component of a CEGAR loop. In their approach, single counterexamples are automatically generalized to patterns, and pattern-based verification is used to exclude potentially infinite families of counterexamples in one single step.

In this article, we develop the theory of pattern-based verification for multithreaded procedural programs. Previous work by Ganty et al. [2012] has shown that pattern-based verification is strictly more expressive than context bounding.[1] Here we focus on the complexity analysis. Like Hague and Lin, we consider multithreaded programs with procedures and counters. We reduce the reachability problem for those executions captured by a pattern to a language-theoretic problem called *Cooperation Modulo a Pattern*, or CMP for short: checking nonemptiness of the asynchronous product of a given set of context-free languages *and* a given pattern. By putting together classical

[1]This is achieved by exhibiting a family of two-thread programs, parameterized by a number n, such that reachability analysis for a *fixed* pattern proves reachability of a program point for all n, but such that the number of context switches needed to reach the program point goes to infinity when n grows.

results by Ginsburg and Spanier [Ginsburg 1966] and more recent results by Verma et al. [2005], we first show that CMP is NP-complete. In the second and main part of the article, we conduct a multiparameter analysis of CMP. The size of an instance of CMP is a function of five parameters: number of threads, number of counters, number of shared variables (for reasons that will be made clearer, those three parameters will be grouped into a single parameter), maximal size of a thread, and size of the pattern. For every subset of parameters, we determine the complexity of CMP when the parameters in the subset (and no others) have a fixed value. At first sight, the results look disappointing: in all interesting cases, the problem is NP-hard. But then we refine the analysis by taking into account the *structure* of the threads. For this, we introduce two further parameters: the maximal number of procedures per thread, denoted by pr, and the longest simple path in the call graph (a path is simple if it visits each node at most once), denoted by lsp. (In particular, for nonrecursive programs, where the call graph is acyclic, lsp is equal to the depth of the call graph. Notice that, while a thread can easily have thousands of instructions and hundreds of procedures, lsp is typically much smaller.) We show that for a fixed number of threads, counters, shared variables, and pattern size, the verification problem can be solved in $st^{O(lsp+\lceil\log(pr+1)\rceil)}$ time, where st is the maximal size of a thread.

The article is organized as follows. Section 2 presents our program model and the reduction of a reachability problem to an equivalent instance of the CMP problem. Section 3 introduces the parameters of the problems, while Section 4 shows that CMP is NP-complete. Sections 5 and 6 contain a detailed complexity analysis of CMP. The NP-hard cases are covered by means of reductions from different NP-complete problems in Section 5. Our main result, the $st^{O(lsp+\lceil\log(pr+1)\rceil)}$ algorithm mentioned earlier, is presented in Section 6. Finally, Section 7 contains conclusions and discusses related work.

2. PROGRAM MODEL AND FORMAL MODEL

2.1. Program Model

We model a sequential program by a *system of flow graphs*, a nonempty set of flow graphs containing one flow graph for each procedure. Each system of flow graphs has an *initial flow graph* corresponding to the `main` procedure. *Nodes* of a flow graph correspond to *control points* of the program, and edges to *sequential statements*. A sequential statement is either a *condition* (a Boolean expression), an *assignment*, or a *procedure call*. Each flow graph has a distinguished *initial node* and a distinguished *final node*. All nodes are reachable from the initial node and have a path to the final node.

A multithreaded program is a tuple of systems of flow graphs, one for each program thread. Dynamic thread creation is not allowed. Threads communicate through shared global variables. Variables either have a finite range or are *counters*: in this case, they range over the natural numbers and have therefore an infinite range, but threads can only access them through the commands `c++` and `c--`, and `assume(c==0)`, `assume(c!=0)`.

Figure 1 shows a program with three threads, each of them containing only one flow graph. Threads `A` and `B` have procedure calls. More precisely, `A` and `B` have recursive calls. The two threads cooperate using recursion and shared variable `x` in order to decrement the counter `val`. The third thread completes its execution whenever `val` reaches zero.

2.2. Formal Model

We model a multithreaded program as a set of context-free grammars such that the terminating executions of the program correspond to the asynchronous product of the languages of the grammars. Although we recall basic notions, we assume the

```
counter val=_H_;
variable x=0;
```

```
A() {
a1: assume (x==1);
a2: x=0;
a3: if * { A(); }
a4: assume (x==0);
a5: x=1;
a6: val--;
za:}
```

```
B() {
b1: x=1;
b2: if * { B(); }
b3: x=0;
zb:}
```

```
C() {
c1: assume(val==0);
zc:}
```

$$\begin{aligned}
&A_1 \to (A, r, 1, x)\, A_2\\
&A_2 \to (A, w, 0, x) A_3\\
&A_3 \to A_1\, A_4\\
&A_3 \to A_4\\
&A_4 \to (A, r, 0, x)\, A_5\\
&A_5 \to (A, w, 1, x)\, A_6\\
&A_6 \to (A, -, val)\, Z_A\\
&Z_A \to \epsilon
\end{aligned}$$

$$\begin{aligned}
&B_1 \to (B, w, 1, x)\, B_2\\
&B_2 \to B_1\, B_3\\
&B_2 \to B_3\\
&B_3 \to (B, w, 0, x)\, Z_B\\
&Z_B \to \epsilon
\end{aligned}$$

$$\begin{aligned}
&C_1 \to (C, {=}0, val)\, Z_C\\
&Z_C \to \epsilon
\end{aligned}$$

Fig. 1. In the C-like program, _H_ stands for a constant and $*$ stands for nondeterministic choice.

reader is familiar with the basics of language theory, including regular and context-free languages (see, e.g., Hopcroft and Ullman [1979]). The rest of the section is devoted to precisely define our model.

The asynchronous product. An *alphabet* Σ is a finite nonempty set of symbols. Given two languages $L_1 \subseteq \Sigma_1^*$ and $L_2 \subseteq \Sigma_2^*$, their asynchronous product, denoted $L_1 \parallel L_2$, is the language L over the alphabet $\Sigma = \Sigma_1 \cup \Sigma_2$ such that $w \in L$ if and only if the projections of w to Σ_1 and Σ_2 belong to L_1 and L_2, respectively. [2] Since asynchronous product is associative, we write $L_1 \parallel \cdots \parallel L_n$ or $\parallel_{i=1}^n L_i$. The asynchronous product has two important special cases: if $\Sigma_1 \cap \Sigma_2 = \emptyset$, then $L_1 \parallel L_2$ is equal to the shuffle of L_1 and L_2; if $\Sigma_1 = \Sigma_2$, then $L_1 \parallel L_2 = L_1 \cap L_2$.

Context-free and regular languages. A *context-free grammar* is a tuple $G = (\mathcal{X}, \Sigma, \mathcal{P}, S)$ where $\mathcal{X}$ is a finite nonempty set of *variables,*[3] Σ is an alphabet, $\mathcal{P} \subseteq \mathcal{X} \times (\Sigma \cup \mathcal{X})^*$ is a finite set of *productions* (the production (X, w) may also be noted $X \to w$), and $S \in \mathcal{X}$ is the *axiom*. Given two strings $u, v \in (\Sigma \cup \mathcal{X})^*$, we write $u \Rightarrow v$ if there exists a production $(X, w) \in \mathcal{P}$ and some words $y, z \in (\Sigma \cup \mathcal{X})^*$ such that $u = yXz$ and $v = ywz$. We use $\Rightarrow^*$ to denote the reflexive transitive closure of $\Rightarrow$. Let $X \in \mathcal{X}$, and we call $X \Rightarrow^* \alpha$ an *X-derivation* if $\alpha \in \Sigma^*$ and we call it a *partial* X-derivation whenever $\alpha \notin \Sigma^*$. We simply say (partial) derivation for a (partial) S-derivation. The language of a grammar is the set $L(G) = \{w \in \Sigma^* \mid S \Rightarrow^* w\}$. A language L is *context free* if $L = L(G)$ for some context-free grammar G. A context-free grammar is *regular* if each production is in $\mathcal{X} \times (\Sigma \cdot \mathcal{X} \cup \mathcal{X} \cup \{\varepsilon\})$. A language L is *regular* if $L = L(G)$ for some regular grammar G.

Multithreaded programs. Let P be a multithreaded program with (pairwise disjoint) sets T, V, C of threads, global variables, and counters, respectively. We show how to

[2]Observe that the $L_1 \parallel L_2$ depends on L_1, L_2 **and** also their underlying alphabet Σ_1 and Σ_2.
[3]We sometimes say *nonterminals* to avoid ambiguities with the program variables.

assign to P a set of context-free grammars, one per element in $T \cup V \cup C$, such that the asynchronous product of their associated languages corresponds to the terminating executions of P.

For the sake of simplicity, we present the translation for the case in which V contains only one variable x (from a theoretical point of view, all variables can be easily encoded into one). We also assume w.l.o.g that upon termination all counters have value 0.

Each grammar is defined over a subalphabet of Σ, the alphabet of *actions*, which we define next:

(1) $(t, r, v, x), (t, w, v, x)$, meaning that thread t reads variable x and gets value v or sets the value of x to v;
(2) $(t, +, c), (t, -, c)$, meaning that thread t increases/decreases counter c;
(3) $(t, =0, c), (t, \neq 0, c)$, meaning that thread t observes that the current value of counter c is zero or nonzero, respectively.

The *actions of thread* t, denoted by Σ_t, are those having t as the first component. The *actions on store* s (i.e., global variable or counter), denoted Σ_s, are those having s as their last component. Observe that distinct threads t, t' have disjoint alphabets Σ_t and $\Sigma_{t'}$. Distinct stores s, s' also have disjoint alphabets Σ_s and $\Sigma_{s'}$; however, Σ_t and Σ_s, where t is a thread and s is a store, may intersect. Observe that $\Sigma = \bigcup_{t \in T} \Sigma_t \cup \bigcup_{s \in V \cup C} \Sigma_s$. We define $\{G_x\}_{x \in T \cup V \cup C}$ to be the set of grammars associated to P. The set contains a grammar G_t over Σ_t for every thread $t \in T$, and a grammar G_s over Σ_s for every store $s \in V \cup C$.

Let F_t be the system of flow graphs of thread t. The grammar G_t has a nonterminal for each node of F_t. The axiom of G_t is given by the initial node of the initial flow graph of F_t. The set of productions contains a special production $Z_t \to \varepsilon$ for the final node Z_t of t, plus productions corresponding to the edges of F_t. We define the productions associated to an edge $X \overset{\ell}{\to} Y$.

We consider first the case in which ℓ corresponds to a condition or an assignment on global variable x. For each condition x==c, the grammar G_t has a production

$$X \to (t, r, c, x)\, Y,$$

and for every assignment x=f(x) and every value v productions

$$X \to (t, r, v, x)\,(t, w, f(v), x)\, Y.$$

The cases in which ℓ manipulates a counter or calls a procedure are encoded as follows:

—if ℓ is c++, c--, assume(c==0), or assume(c!=0), then G_t has a production

$$X \to (t, +, c)\, Y, \quad X \to (t, -, c)\, Y, \quad X \to (t, =0, c)\, Y, \text{ or } X \to (t, \neq 0, c)\, Y; \qquad (1)$$

—if $\ell = \textit{call } P$, then G_t has a production

$$X \to P_0\, Y, \qquad (2)$$

where P_0 is the initial node of procedure P. G_t has no other productions.

Given a global variable x, the associated grammar G_x has a nonterminal X_v for each possible value v of x, and productions

$$X_v \to (t, r, v, x)\, X_v, \quad X_v \to (t, w, v', x)\, X_{v'}, \quad X_v \to \varepsilon \qquad (3)$$

for every thread t and values v, v'.

Finally, we give the grammar G_c for a counter c. A counter can be easily encoded as a pushdown automaton with one stack symbol (and a bottom of stack symbol), which in turn can be translated into a context-free grammar. The grammar G_c is the result of this process, but for completeness we describe it in full. G_c has three nonterminals: the axiom S_0 and two further nonterminals 1 and #. If the initial value of the counter is v_0, then G_c has a production $S_0 \to 1^{v_0}\#$ (if v_0 is large we can also use a more compact encoding in which, loosely speaking, G_c encodes v_0 in binary).

Further, G_c contains the following productions for every thread t:

$$\# \to (t, =0, c)\,\# \qquad \# \to (t, +, c)\,1\,\# \qquad \# \to \varepsilon, \tag{4}$$

$$1 \to (t, \neq 0, c)\,1 \qquad 1 \to (t, +, c)\,1\,1 \qquad 1 \to (t, -, c) \tag{5}$$

Program executions as an asynchronous product of languages. Given the set $\{G_x\}_{x\in T\cup V\cup C}$ of grammars associated to P, we define $L(P)$, *the set of terminating executions* of P, as the language

$$\|_{t\in T}\; L(G_t)\; \|_{x\in V}\; L(G_x)\; \|_{c\in C}\; L(G_c).$$

It is easy to see that $L(P)$ contains the words $w \in \Sigma^*$ satisfying the following property: for every thread t, the projection of w onto Σ_t is a word of $L(G_t)$; and for every store s, the projection of w onto Σ_s is a word of $L(G_s)$.

The total size of the grammars is polynomial in the size of the program (if the initial values of the counters are described using the compact encoding) but exponential in the size D of the finite domain of the global variable (where we assume that D is written in binary).

Example 2.1. Figure 1 shows a multithreaded program in a `C`-like language. To ease the presentation, let us assume that `_H_` is equal to 8. The program has three threads, `A`, `B`, and `C`. Threads `A` and `B` consist of a single recursive procedure and thread `C` consists of a single `assume` statement. The program has a counter `val` initialized to 8 and one global variable `x` whose initial value is 0. The terminating executions of the program are those in which all threads reach their `z` line (i.e., `za`, `zb`, or `zc`). The program has terminating executions, but even the shortest ones are relatively long:

$$\begin{aligned}
&(B, w, 1, x)\\
&((A, r, 1, x)(A, w, 0, x)(B, w, 1, x))^7\\
&(A, r, 1, x)(A, w, 0, x)(A, r, 0, x)(A, w, 1, x)(A, -, val)\\
&((B, w, 0, x)(A, r, 0, x)(A, w, 1, x)(A, -, val))^7\\
&(C, = 0, val).
\end{aligned}$$

In particular, observe that the execution has 30 context switches (i.e., alternations between symbols of Σ_A, Σ_B, or Σ_C).

Programs with more than one global variable. We have assumed that the multithreaded program P has only one variable `x`. From a theoretical point of view, since variables have finite domains, all variables can be easily encoded into one. However, from a practical point of view, it is convenient to remove this assumption. This can be achieved at the price of a slightly more complicated translation into grammars, which we now sketch.

We assume for simplicity that all global variables have the same finite range of values. Each global variable $x \in V$ now comes with a lock that ensures exclusive access to x when evaluating a condition or performing an assignment. Every access to x requires acquiring the lock.

For each thread $t \in T$ and variable $x \in V$, we add to the alphabet of actions Σ new symbols (t, l, x), (t, u, x), meaning that thread t locks/unlocks variable x. The alphabet

Σ_t of the thread t is thus extended with a pair of lock/unlock symbols for each variable $x \in V$.

Next, we revisit the productions of G_t arising from conditions and assignments. To simplify the presentation, assume that conditions (assignments) are of the form `x == f(y)` and `x=f(y)`, respectively, where x, y are global variables and f is some function. For a condition `x == f(y)`, the grammar G_t includes productions

$$X \rightarrow (t, l, x)(t, l, y)\,(t, r, v_y, y)(t, r, f(v_y), x)\,(t, u, y)(t, u, x)\,Y \tag{6}$$

for every value v_y: first the thread acquires locks on x and y, then it checks the condition `x==f(y)`, and then it unlocks x and y. The case of an assignment is similar: just substitute $(t, w, f(v_y), x)$ for $(t, r, f(v_y), x)$. The productions modeling counter manipulation and procedure calls remain the same.

Finally, we modify the definition of the grammars G_x for each global variable $x \in V$. The grammar G_x has a nonterminal X_v for each possible value v of x, and productions

$$X_v \rightarrow (t, l, x)X'_v, \quad X'_v \rightarrow (t, u, x)X_v, \quad X'_v \rightarrow (t, r, v, x)X'_v, \quad X'_v \rightarrow (t, w, v', x)X'_{v'} \tag{7}$$

for every thread t and values v, v'. This ensures that a thread can only read or write a variable after acquiring a lock.

Pattern-based verification. Pattern-based verification only explores the executions of a multithreaded program conforming to what we call in this article *patterns*. Patterns are regular expressions of the form $w_1^* w_2^* \ldots w_n^*$, where $n \geq 1$ and for every i, $1 \leq i \leq n$, w_i is a nonempty word over the alphabet Σ of the actions of the program [Kahlon 2009b; Ganty et al. 2012; Long et al. 2012].[4]

We study the *pattern verification problem*, defined as follows:

Definition 2.2 *(Pattern Verification).*
Instance: A program P, a pattern $\boldsymbol{p}$.

Question: Is some word of $L(\boldsymbol{p})$ an execution of P? (Formally, $L(P) \parallel L(\boldsymbol{p}) \overset{?}{\neq} \emptyset$.)

Consider for instance the terminating execution of Ex. 2.1 and the pattern

$$\begin{aligned}\boldsymbol{p} = \; & (B, w, 1, x)^* \\ & ((A, r, 1, x)(A, w, 0, x)(B, w, 1, x))^* \\ & ((A, r, 1, x)(A, w, 0, x)(A, r, 0, x)(A, w, 1, x)(A, -, val))^* \\ & ((B, w, 0, x)(A, r, 0, x)(A, w, 1, x)(A, -, val))^* \\ & (C, =0, val)^*\end{aligned}$$

obtained by leaving unspecified the number of times that each segment must occur (i.e., exponents are replaced by Kleene stars). Loosely speaking, this pattern corresponds to the potential executions of the program in which the recursive calls to A and B are interleaved. By solving the pattern verification problem, we can automatically check whether the program has executions of this form. Moreover, the verification algorithm returns (an encoding of) all the tuples of numbers such that, after replacing the Kleene stars by them, we obtain a real execution of the program. Finally, observe that the same pattern can be used for any value of `_H_`.

[4]In the literature, patterns are also called *bounded expressions* and languages included in some bounded expressions are called *bounded languages*.

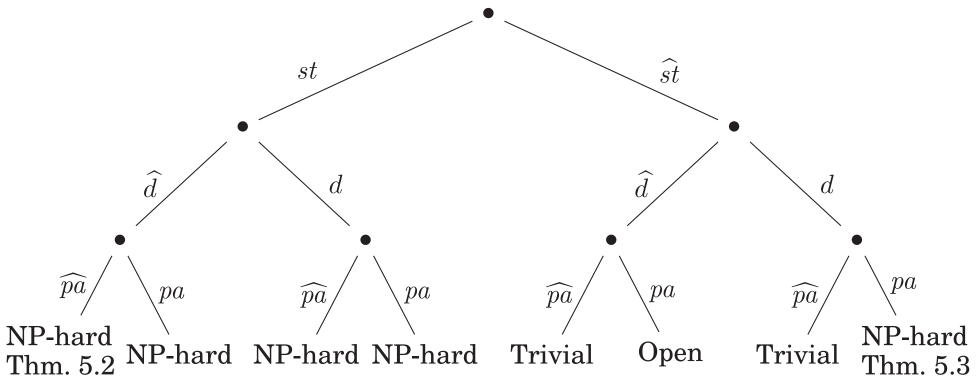

Fig. 2. Binary tree for the fixed-parameter analysis.

3. OUTLOOK AND PARAMETERS

In the rest of the article, we analyze the complexity of the pattern verification problem in the size of the input (defined later) and then conduct a more detailed multiparameter analysis.

In Section 4, we prove that the problem is in NP and also show that it is NP-complete. In Section 5, we analyze the impact on the complexity of the five natural parameters of the problem:

—*Number of threads*, denoted by t.
—*Number of global variables*, denoted by v.
—*Number of counters*, denoted by c.
—*Maximal size of a thread*, denoted by st.
The size of a thread is the number of transitions of all its flow graphs put together. We define the size of a thread t_i by st_i, and for a program with threads $\langle t_1, \ldots, t_t \rangle$, we let $st = \max_{i=1}^{t} st_i$.
—*Size of the pattern*, denoted by pa.
The size of a pattern $\boldsymbol{p} = w_1^* \ldots w_n^*$ denoted $|\boldsymbol{p}|$ is given by $\sum_{i=1}^{n} |w_i|$, where $|w|$ is the length of w. Observe that $pa \geq n \geq 1$.

We study the complexity when a subset of those parameters has a fixed value. However, since each thread, variable, and counter contributes one grammar, the parameters t, v, and c can be summarized into one: $d := t + v + c$. We thus study the complexity when a subset of d, st, and pa has a fixed value. There are eight possible cases, corresponding to fixing the parameters of one of the eight subsets. We determine the complexity for seven cases—we leave the (not very interesting) case in which d and st are fixed, but pa is not, open. For these seven cases, we show that the problem stays NP-complete unless both st and pa are fixed, in which case it becomes trivial (because, loosely speaking, there are only a fixed number of different instances). Figure 2 summarizes these results; a parameter wearing a hat reads as a fixed value parameter.

Since these results are not very discriminating (i.e., we get no nontrivial polynomial or subexponential cases), in Section 6 we refine the analysis by taking into account the *structure* of the threads. For this we define two new parameters:

—*Maximal number of procedures in a thread*, denoted by pr.
For a program with threads $\langle t_1, \ldots, t_t \rangle$, we define $pr = \max_{i=1}^{t} pr_i$, where pr_i denotes the number of procedures of thread t_i. Observe that in every case $pr \geq 1$.
—*Maximal length of simple paths in the call graph of a thread*, denoted by lsp.

The call graph of a thread is defined as usual: the graph has a node for each procedure and an edge from procedure P_1 to procedure P_2 if the flow graph of P_1 contains a call to P_2. A path of the call graph is *simple* if it visits each node at most once. For a program with threads $\langle t_1, \ldots, t_t \rangle$, we denote by lsp_i the length (defined as the number of nodes) of the longest simple path in the call graph of T_i and define $lsp = \max_{i=1}^{t} lsp_i$.

Observe that $pr \geq lsp \geq 1$. We show in Theorem 6.12 that for fixed d and pa, the pattern verification problem can be solved in time $st^{O(lsp+\lceil \log(pr+1) \rceil)}$, that is, in subexponential time in the size of the threads. Observe that, even though in practice the maximal number of procedures of a thread can be large, lsp is usually small. In the particular case of nonrecursive programs, lsp is the maximal call depth.

Finally, notice that the previous list of parameters does not contain the size D (written in binary) of the data domain. The complexity is exponential in D, and it is easy to show that this exponential blowup is unavoidable. For this reason, we assume the data domain has fixed size.

3.1. Parameters: From the Program Model to the Formal Model

The translation of a program into a collection of context-free grammars allows us to formulate the pattern verification problem and their associated parameters in language-theoretic terms.

The grammars $G_1, \ldots, G_d$ derived from a multithreaded program can all be put in *program normal form*, if necessary, with a constant increase in the number of variables and productions.

A grammar $G = (\mathcal{X}, \Sigma, \mathcal{P}, S)$ is in program normal form if every variable appears in some production, and productions are of the form $X \to w$, where $w \in \{\varepsilon\} \cup (\Sigma \times \mathcal{X}) \cup \mathcal{X}^2$. So the pattern verification problem reduces to the following language-theoretic problem:

Definition 3.1 *(Cooperation Modulo a Pattern (CMP)).*
Instance: Context-free grammars $G_1, \ldots, G_d$ in program normal form, and a pattern $\boldsymbol{p}$ over Σ (the union of the alphabets of all the G_is).
Question: Is $\|_{i=1}^{d} L(G_i) \parallel L(\boldsymbol{p}) \neq \varnothing$?

The *size of a grammar* in program normal form is the number of its productions. Notice that since every node of each procedure of the multithreaded program is reachable, the size of a grammar in program normal form is at least the number of variables.

We map the parameters of the pattern verification problem to parameters of CMP. The number of threads, counters, and variables (d) is equal to the number of grammars; the maximal size of a thread (st) is equal to the maximal size (as defined earlier) of the corresponding grammars in program normal form; and the size of the pattern (pa) is mapped to itself. We now describe the formal parameters corresponding to the maximal number of procedures in a thread (pr) and to the maximal length of simple paths in the call graph of a thread (lsp).

The translation from program threads to grammars "forgets" the procedure structure, but this obstacle can be easily overcome. Recall that each procedure has one single initial node. We further assume that every procedure can be invoked through a chain of calls from the main procedure (procedures that cannot be called at all can be easily detected and removed). Under this assumption, the initial nodes of the procedures of a thread t_i correspond to the *procedure variables* of its corresponding grammar, say, $G_i = (\mathcal{X}_i, \Sigma_{t_i}, \mathcal{P}_i, S_i)$, defined as follows: a variable X is a procedure variable if $X = S_i$ (corresponding to the initial node of the main procedure), or if G_i has a production of the form $Y \to XZ$ (these productions come from procedure calls, and X is the initial node of the callee). The number of procedures of a thread is then equal to the number of procedure variables in its associated grammar. The *call graph* of a grammar G_i corresponding to the thread is defined as follows: the nodes are the procedure variables, and there is an edge from X to Y if $X \Rightarrow^* vYw$ for some $v, w \in (\Sigma \cup \mathcal{X}_i)^*$. It is easy to see that lsp_i is equal to the length of the longest simple path in the call graph of G_i.

4. NP-COMPLETENESS OF CMP

In this section, we prove the decidability[5] of CMP. We further prove that it is NP-complete in the size of the input, defined as the sum of the sizes of the grammars and the pattern.

4.1. CMP Is NP-Hard

We show[6] that CMP is NP-hard even for regular grammars and fixed pattern $\boldsymbol{p} = a^*$. From a programming point of view, this means that the verification problem is already NP-hard for multithreaded procedureless programs, and the simplest pattern (actually, in this case the pattern plays no active role; we include it because the formal definition of CMP requires a pattern).

THEOREM 4.1. *The following problem is NP-hard:*
Instance: *Regular grammars $G_1, \ldots, G_d$ in program normal form over alphabet $\{a\}$.*
Question: *Is $\|_{i=1}^{d} L(G_i) \parallel L(\boldsymbol{p}) \neq \varnothing$ for the pattern $\boldsymbol{p} = a^*$?*

PROOF. We first observe that since the alphabets of $G_1, \ldots, G_d$ and $\boldsymbol{p}$ are given by $\{a\}$, the asynchronous product ($\|$) behaves exactly like language intersection ($\cap$). Hence, in order to ease understanding, the proof uses $\cap$ instead of $\|$. The proof is by reduction from 3-CNF-SAT. Let Ψ be a propositional formula with d variables and m clauses $c_1, \ldots, c_m$. We define for each clause c_i a regular grammar G_i over the alphabet $\{a\}$ such that $\bigcap_{i=1}^{m} L(G_i) \neq \varnothing$ if and only if Ψ is satisfiable. Clearly, we then also have that $\bigcap_{i=1}^{m} L(G_{c_i}) \cap L(\boldsymbol{p}) \neq \varnothing$ holds for $\boldsymbol{p} = a^*$ if and only if Ψ is satisfiable.

We need some preliminaries. We first assign to each variable v of Ψ a prime number n_v, written in binary. We sketch how to compute these numbers in polynomial time. It is well known that the ith prime number p_i satisfies $p_i < i \ln i + i \ln\ln i$ (this is an easy consequence of the Prime Number Theorem describing the asymptotic distribution of prime numbers). So we can compute d primes by applying a primality test (an algorithm to decide whether a given number is a prime) to all numbers between 1 and $d \ln d + d \ln\ln d$. This procedure requires to test $O(d \ln d)$ numbers written in binary, each of them of size $O(\log(d \ln d)) = O(\log d)$, in polynomial time in d. To achieve this, we can use any primality test that, given a number in binary with x bits, decides whether it is a prime in $O(2^x)$ time. So it even suffices to take the most primitive primality test: divide by all numbers between 2 and 2^{x-1}, and check if any of these divisions has rest 0.

We also need the preliminary notion of a number being a witness of the truth of a clause. Given a clause c and a variable v, we say that a number $0 \leq k$ is a (c, v)-*witness* if v appears positively in c and $k \equiv 0 \mod n_v$ or v appears negatively in c and $k \not\equiv 0 \mod n_v$. Further, k is a c-*witness* if it is a (c, x)-witness, a (c, y)-witness, or a (c, z)-witness where x, y, and z are the three variables occurring in c. For instance, if $c = x \vee \neg y \vee z$ and $n_x = 2, n_y = 3, n_z = 5$, then k is a c-witness if $k \equiv 0 \mod 2$, or $k \not\equiv 0 \mod 3$, or $k \equiv 0 \mod 5$, that is, if $k \neq 3, 9, 21, 27$. Given an assignment ϕ to the variables of Ψ, let n_ϕ be the product of the numbers of the variables set to true by ϕ. We

[5]The decidability of CMP when all the alphabets of all the languages coincide was originally proved by Ginsburg [1966]. Although our result is more general, its proof is largely inspired by the proof of Ginsburg.
[6]The proof is due to Mikołaj Bojańczyk.

claim that ϕ satisfies c if and only if n_ϕ is a c-witness. For this, assume that ϕ satisfies c. Then ϕ sets some variable x appearing positively in c to true or some variable y appearing negatively in c to false. In the first case, by definition of n_ϕ, we have $n_\phi \equiv 0 \mod n_x$, and so n_ϕ is a (c, x)-witness; in the second case, we have $n_\phi \not\equiv n_y$, and so n_ϕ is a (c, y)-witness. For the other direction of the claim, assume that n_ϕ is a c-witness. Then n_ϕ is a (c, x)-witness for some variable x of c. By the definition of (c, x)-witness, there are two cases: either $n_\phi \equiv 0 \mod n_x$ and x appears positively in c, or $n_\phi \not\equiv 0 \mod n_x$ and x appears negatively in c. In the first case, the assignment ϕ sets x to true, and so, since x appears positively in c, the assignment satisfies c. The second case is symmetric. This concludes the proof of the claim.

We are now ready to describe the reduction from 3-CNF-SAT. For each clause c let n_c be the product of the primes of the three variables occurring in c. We define a grammar G_c in program normal form over the alphabet $\{a\}$. The grammar G_c has the numbers $0, 1, 2, \ldots, n_c - 1$ as grammar variables; 0 as axiom; productions $k \to a\ (k \oplus_c 1)$ for every $0 \le k \le n_c - 1$, where $\oplus_c$ is addition modulo n_c; and a further production $k \to \epsilon$ for each c-witness $k \le n_c - 1$. Clearly, we have $L(G_c) = \{a^k \mid k \text{ is a } c\text{-witness}\}$. By the previous claim, an assignment ϕ satisfies c if and only if n_ϕ is a c-witness, and so ϕ satisfies c if and only if $a^{n_\phi} \in L(G_c)$. So $\bigcap_{i=1}^{m} L(G_{c_i}) \neq \emptyset$ if and only if there is an assignment n_ϕ that satisfies all clauses $c_1, \ldots, c_m$ of Ψ, that is, if and only if Ψ is satisfiable. □

4.2. CMP Is in NP

We show that CMP is in NP. The direct approach would be to show that if $\|_{i=1}^{d} L(G_i) \parallel L(\boldsymbol{p}) \neq \emptyset$, then there is a witness $w \in \|_{i=1}^{d} L(G_i) \parallel L(\boldsymbol{p})$ of polynomial length. However, it is easy to construct instances of size k for which the shortest witness is the word a^{2^k} (see also Theorem 5.2). So we proceed differently, in two steps: first, we polynomially reduce CMP to a problem about Parikh images of context-free grammars, and then we show that this problem is in NP.

We recall some basic notions about multisets.

Multisets. A *multiset* $\mathbf{m} : \Sigma \to \mathbb{N}$ maps each symbol of Σ to a natural number. $\mathbb{M}[\Sigma]$ denotes the set of all multisets over Σ. The empty multiset is denoted $\emptyset$. The size of a multiset $\mathbf{m}$ is $|\mathbf{m}| = \sum_{\sigma \in \Sigma} \mathbf{m}(\sigma)$. Given two multisets $\mathbf{m}, \mathbf{m}' \in \mathbb{M}[\Sigma]$, and we define $\mathbf{m} \oplus \mathbf{m}' \in \mathbb{M}[\Sigma]$ as the multiset satisfying $(\mathbf{m} \oplus \mathbf{m}')(a) = \mathbf{m}(a) + \mathbf{m}'(a)$ for every $a \in \Sigma$. Given $\mathbf{m} \in \mathbb{M}[\Sigma]$ and $c \in \mathbb{N}$, we define $c \cdot \mathbf{m}$ as the multiset satisfying $(c \cdot \mathbf{m})(a) = c \cdot \mathbf{m}(a)$ for every $a \in \Sigma$. By fixing a linear order on Σ, every multiset $\mathbf{m}$ can be seen as a vector of $\mathbb{N}^k$ where $k = |\Sigma|$, and vice versa.

Parikh image. The *Parikh image* of a word $w \in \Sigma^*$ is the multiset $\Pi(w) : \Sigma \to \mathbb{N}$ that assigns to each $a \in \Sigma$ the number of occurrences of a in w. The Parikh image of a language L defined over alphabet Σ, denoted by $\Pi(L)$, is the set of Parikh images of its words. Given $\Sigma' \subseteq \Sigma$, we further define $\Pi_{\Sigma'}(L)$ as the projection of $\Pi(L)$ onto Σ'.

We prove that CMP can be reduced to the following problem:

Definition 4.2 *(Nondisjointness of Parikh Images (nDPK)).*
Instance: Context-free grammars $G_1, \ldots, G_d$ in program normal form over a common alphabet Σ and a subalphabet $\Sigma' \subseteq \Sigma$.
Question: Is $\bigcap_{i=1}^{d} \Pi_{\Sigma'}(L(G_i)) \neq \emptyset$?

We proceed with the following reduction step.

LEMMA 4.3. *Given a CMP problem instance, that is, context-free grammars $G_1, \ldots, G_d$ and a pattern $\boldsymbol{p} = w_1^* \ldots w_n^*$ over alphabet Σ (the union of all the alphabet of G_is), we can effectively construct an alphabet $\widetilde{\Sigma}$ disjoint from Σ and context-free grammars*

$G'_1, \ldots, G'_d$ each over alphabet $\Sigma \cup \widetilde{\Sigma}$ such that

$$\|_{i=1}^{d} L(G_i) \parallel L(\boldsymbol{p}) \neq \emptyset \quad \textit{iff} \quad \bigcap_{i=1}^{d} \Pi_{\widetilde{\Sigma}}(L(G'_i)) \neq \emptyset.$$

Proof. Define (i) $\widetilde{\Sigma} = \{a_1, \ldots, a_n\}$ to be an alphabet disjoint from Σ, (ii) $\widetilde{\boldsymbol{p}}$ to be the pattern over $\Sigma \cup \widetilde{\Sigma}$ given by $(a_1 w_1)^* \ldots (a_n w_n)^*$, and (iii) L_i to be $L(G_i)$. We claim that $\bigcap_{i=1}^{d} \Pi_{\widetilde{\Sigma}}\big(L_i \parallel L(\widetilde{\boldsymbol{p}})\big) \neq \emptyset$ if and only if $\|_{i=1}^{d} L_i \parallel L(\boldsymbol{p}) \neq \emptyset$.

Let $w \in \|_{i=1}^{d} L_i \parallel L(\boldsymbol{p})$. Because Σ (the alphabet of $L(\boldsymbol{p})$) is the union of all the alphabets of L_is and the definition of $\parallel$, we find that $\|_{i=1}^{d} L_i \parallel L(\boldsymbol{p}) = \bigcap_{i=1}^{d}(L_i \parallel L(\boldsymbol{p}))$, and hence that $w \in L_i \parallel L(\boldsymbol{p})$ for each $1 \leq i \leq d$. Further, the definition of $\boldsymbol{p}$ shows that there exist $t_1, \ldots, t_n \in \mathbb{N}$ such that $w = w_1^{t_1} \ldots w_n^{t_n}$. Next, since Σ and $\widetilde{\Sigma}$ are disjoint, the definition of $\parallel$ and that of $\widetilde{\boldsymbol{p}}$ show that $(a_1 w_1)^{t_1} \ldots (a_n w_n)^{t_n} \in L_i \parallel L(\widetilde{\boldsymbol{p}})$ for every $1 \leq i \leq d$. For the same reason and by definition of Π, we further find that $(t_1, \ldots, t_n) \in \Pi_{\widetilde{\Sigma}}(L_i \parallel L(\widetilde{\boldsymbol{p}}))$ for every $1 \leq i \leq d$, and hence that $(t_1, \ldots, t_n) \in \bigcap_{i=1}^{d} \Pi_{\widetilde{\Sigma}}(L_i \parallel L(\widetilde{\boldsymbol{p}}))$, and we are done.

For the other implication, let $(t_1, \ldots, t_n)$ be a vector of $\bigcap_{i=1}^{d} \Pi_{\widetilde{\Sigma}}\big(L_i \parallel L(\widetilde{\boldsymbol{p}})\big)$. Thus, the definition of $\widetilde{\boldsymbol{p}}$ and its alphabet $\Sigma \cup \widetilde{\Sigma}$ shows that $(a_1 w_1)^{t_1} \ldots (a_n w_n)^{t_n} \in L_i \parallel L(\widetilde{\boldsymbol{p}})$ for every $1 \leq i \leq d$. Next, $\widetilde{\Sigma} \cap \Sigma = \emptyset$ and the definition of $\boldsymbol{p}$ show that $w_1^{t_1} \ldots w_n^{t_n} \in L_i \parallel L(\boldsymbol{p})$ for every $1 \leq i \leq d$, and so finally we have $w_1^{t_1} \ldots w_n^{t_n} \in \|_{i=1}^{d} L_i \parallel L(\boldsymbol{p})$ by definition of $\parallel$ and because $\boldsymbol{p}$ is defined over Σ (the union of the alphabets of all the G_is). This concludes the proof of the claim.

Let us consider the language $L_i \parallel L(\widetilde{\boldsymbol{p}})$. Observe that the language L_i is context free, while $L(\widetilde{\boldsymbol{p}})$ is a regular. Since context-free languages are closed under asynchronous product with regular languages [Hopcroft and Ullman 1979], $L_i \parallel L(\widetilde{\boldsymbol{p}})$ is context free. Moreover, since the constructions proving closure under these operations are effective, we conclude the lemma's proof by defining G'_i to be the grammar over $\Sigma \cup \widetilde{\Sigma}$ such that $L(G'_i) = L_i \parallel L(\widetilde{\boldsymbol{p}})$. □

For the complexity analysis in Section 6, we need a careful analysis of the relation between G_i and G'_i. In particular, we need to determine the relation not only between their sizes but also between their maximal number of procedure variables and their length of the longest simple path in the call graph.

Lemma 4.4. *Given $\boldsymbol{p} = w_1^* \ldots w_n^*$ over alphabet Σ and a grammar G in program normal form over some alphabet included in Σ, we can compute in polynomial time a grammar G^f in program normal form over alphabet $\Sigma \cup \widetilde{\Sigma}$ such that $\widetilde{\Sigma} = \{a_1, \ldots, a_n\}$ is disjoint from Σ and:*

—*$L(G^f) = L(G) \parallel L(\widetilde{\boldsymbol{p}})$ where $\widetilde{\boldsymbol{p}} = (a_1 w_1)^* \ldots (a_n w_n)^*$;*
—*Let st and st^f be the size of G and G^f, respectively. Similarly let pr and pr^f be the number of procedure variables in G and G^f. Finally, let lsp and lsp^f be the length of the longest simple path in the call graph of G and G^f. Then $st^f \in O(pa^3 \cdot st)$, $pr^f \in O(pa^2 \cdot pr)$, and $lsp^f \in O(pa^2 \cdot lsp)$, where pa is the size of $\boldsymbol{p}$.*

Proof. We sketch the construction of G^f. A detailed proof is given in Appendix A.1. Let $G^{\widetilde{\boldsymbol{p}}}$ be a regular grammar with $O(pa)$ variables such that $L(G^{\widetilde{\boldsymbol{p}}}) = L(\widetilde{\boldsymbol{p}})$ (this grammar clearly exists). The variables of G^f are triples $[\mathrm{Q}_1 X \mathrm{Q}_2]$, where X is a variable of G and $\mathrm{Q}_1, \mathrm{Q}_2$ are variables of $G^{\widetilde{\boldsymbol{p}}}$ such that Q_2 is reachable from Q_1. The productions are chosen to satisfy that $[\mathrm{Q}_1 X \mathrm{Q}_2] \Rightarrow^* w$ holds in G^f if and only if (1) $X \Rightarrow^* u$ holds in G, where u results from w by deleting all symbols from $\widetilde{\Sigma}$, and (2) $\mathrm{Q}_1 \Rightarrow^* w\, \mathrm{Q}_2$ in $G^{\widetilde{\boldsymbol{p}}}$.

The construction is similar to the triple construction used to transform a pushdown automaton into an equivalent context-free grammar.

Let us start with st^f, the size of G^f, which is defined as the number of productions in G^f. Each production of G^f is defined from a production of G and up to three variables of $G^{\mathbf{p}}$. Therefore, we find that st^f is at most $O(pa^3 \cdot st)$.

For pr^f, the number of procedure variables in G^f, observe that in the previous construction, each variable X of G yields $O(pa^2)$ variables $[\mathrm{Q}_1 X \mathrm{Q}_2]$ in G^f, and that $[\mathrm{Q}_1 X \mathrm{Q}_2]$ is a procedure variable in G^f only if X is a procedure variable in G. Therefore, pr^f is at most $O(pa^2 \cdot pr)$.

For lsp^f, the length of the longest simple path in the call graph of G^f, observe that, by the previous construction, if we project a path $[\mathrm{Q}_1 X_1 \mathrm{Q}'_1] \dots [\mathrm{Q}_n X_n \mathrm{Q}'_n]$ of the call graph of G^f onto the variables of G (i.e., onto $X_1 X_2 \dots X_n$), we obtain a path of the call graph of G. Therefore, lsp^f is at most $O(pa^2 \cdot lsp)$. □

4.2.1. nDPK is in NP. Using Lemma 4.3, we can show that CMP is decidable. The central proof arguments are (a) given a context-free language, its Parikh image is an effectively computable semilinear set [Parikh 1966], and (b) emptiness of the intersection of semilinear sets is decidable [Ginsburg 1966]. For our NP-completeness result, we need further results [von zur Gathen and Sieveking 1978; Verma et al. 2005] showing that semilinear sets are exactly the sets definable by (existential) Presburger formulas, and that satisfiability of existential Presburger formulas is NP-complete. We briefly recall these notions.

Given $k \geq 1$, $c \in \mathbb{N}^k$, and $P = \{p_1, \dots, p_m\} \subseteq \mathbb{N}^k$, we denote by $L(c; P)$ the subset of $\mathbb{N}^k$ defined as follows:

$$L(c; P) = \{\mathbf{m} \in \mathbb{N}^k \mid \exists \lambda_1, \dots, \lambda_m \in \mathbb{N} : \mathbf{m} = c \oplus (\lambda_1 \cdot p_1) \oplus \cdots \oplus (\lambda_m \cdot p_m)\}.$$

A set $S \subseteq \mathbb{N}^k$ is *linear* if $S = L(c; P)$ for some $c \in \mathbb{N}^k$ and some finite $P \subseteq \mathbb{N}^k$. A *semilinear set* is a finite union of linear sets.

Existential Presburger formulas ϕ are defined by the following grammar and interpreted over natural numbers:

$$t ::= 0 \mid 1 \mid x \mid t_1 + t_2 \qquad \phi ::= t_1 = t_2 \mid t_1 > t_2 \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \exists x . \phi_1 .$$

Given an existential Presburger formula ϕ, we denote by $[\![\phi]\!]$ the set of valuations of the free variables of ϕ that make ϕ true; ϕ is satisfiable if and only if $[\![\phi]\!]$ is nonempty. Satisfiability of existential Presburger formulas is an NP-complete problem (see, e.g., von zur Gathen and Sieveking [1978]).

A set $S \subseteq \mathbb{N}^k$ is (existential) Presburger definable if $S = [\![\phi]\!]$ for some (existential) Presburger formula ϕ. It is well known that a set is Presburger definable if and only if it is existential Presburger definable if and only if it is semilinear.

We use the following result by Verma et al. [2005, Theorem 4]: given a context-free grammar G over Σ, one can compute in linear time an existential Presburger formula ϕ_G such that $[\![\phi_G]\!] = \Pi(L(G))$. We briefly sketch the proof for future reference. Let $G = (\mathcal{X}, \Sigma, \mathcal{P}, S)$. A result of Esparza [1997] characterizes $\Pi(L(G))$ as the set of all multisets $\mathbf{m} \in \mathbb{M}[\mathcal{P}]$ that are the solution of a certain system of linear equations. Intuitively, the system of equations falls into two parts: (1) Flow equations about the production and consumption pattern of productions. The occurrence of a production produces and consumes variables (e.g., $X \to YaZ$ consumes one instance of X and produces one instance of Y and Z). Flow equations capture the constraint that each variable must be produced and consumed an equal number of times. (2) Connectivity equations. The flow equations allow one to produce variables "out of nowhere," as long as they are consumed again. The connectivity equations capture that a production with head X can only fire if there is a chain of productions *connecting* the axiom of the grammar to X, such that every production in the path fires. Then, Verma et al. [2005, Theorem 4] show that this set of multisets is Presburger definable by explicitly constructing an existential Presburger formula in linear time in the size of G. We refer the reader to the corresponding publications [Verma et al. 2005; Esparza 1997] for more details.

THEOREM 4.5. *CMP is in NP.*

PROOF. Following Lemmas 4.3 and 4.4, it suffices to show that nDPK is in NP. Let $G_1, \ldots, G_d$ be an instance of nDPK, and let ϕ_{G_i} be the existential Presburger formula of Verma et al. [2005, Theorem 4] defining $\Pi(L(G_i))$. We assume that for each i, j, $1 \le i < j \le d$, we have that ϕ_{G_i} and ϕ_{G_j} share no variable. We further assume variables of the form $x^{(i)}_{a_j}$ that counts the number of occurrences of $a_j \in \widetilde{\Sigma}$ in $\Pi(L(G_i))$. Let

$$\Psi = \phi_{G_1} \wedge \cdots \wedge \phi_{G_d} \wedge x^{(1)}_{a_1} = \cdots = x^{(d)}_{a_1} \wedge x^{(1)}_{a_n} = \cdots = x^{(d)}_{a_n}.$$

We have that Ψ is satisfiable if and only if $\bigcap_{i=1}^{d} \Pi_{\widetilde{\Sigma}}(L(G_i)) \neq \varnothing$. Since existential formulas are closed under conjunction, Ψ is an existential Presburger formula. Since satisfiability of existential Presburger formulas is NP-complete (see, e.g., von zur Gathen and Sieveking [1978]), the result follows. □

5. A MULTIPARAMETER ANALYSIS

We determine the complexity of CMP for instances in which the value of one or more of the three parameters d (number of grammars), st (maximal size of a grammar), and pa (size of the pattern) are *fixed*, that is, bounded by a fixed constant not part of the input. Since each parameter can be fixed or not, there are in principle eight possible cases. We use $\widehat{p}$ to denote that a parameter p is fixed and p to denote that it is not fixed. So, for instance, the case in which d is fixed but st and pa are not is denoted by CMP($\widehat{d}, st, pa$).

Figure 2 shows our seven complexity results as a full binary tree, where at each node we choose between fixing a parameter or not. Leaves are labeled either with the upper bound "Trivial" (meaning that the problem has only finitely many instances, which can be tabled), with the lower bound "NP-hard" (recall that even if no parameter is fixed, the problem is in NP), or with "Open," which has the expected meaning. Since we are mostly interested in the dependency of the complexity when st, d, or both grow, we leave out the open case. The path leading to Theorem 5.2 indicates that if st is not fixed, then CMP is NP-hard, even if d and pa are fixed. It follows that the problems associated with all the other leaves of the left subtree are NP-hard too. In the right subtree, Theorem 5.3 proves that if st is fixed but neither d nor pa is fixed, then CMP is NP-hard, which does not give information about the other leaves. Sections 5.1 and 5.2 consider the cases in which st is arbitrary and fixed, respectively.

5.1. Threads of Arbitrary Size

We assume that st is not fixed. We show that CMP remains NP-complete even for two grammars and fixed pattern a^*, that is, that CMP($st, \widehat{d}, \widehat{pa}$) is NP-complete. The proof is by reduction from the 0-1 Knapsack problem.

Definition 5.1 *(0-1 Knapsack Problem).*
Instance: (1) A set of objects $\{o_1, \ldots, o_m\}$ and their associated weights $\{w_1, \ldots, w_m\}$, which are positive integers given in binary. (2) A positive integer W given in binary.
Question: Is there a subset $S \subseteq \{o_1, \ldots, o_m\}$ such that the weight of S is equal to W?

THEOREM 5.2. *The following problem is NP-hard:*
Instance: *Two context-free grammars G_1, G_2 in program normal form with alphabet $\{a\}$.*
Question: *Is $L(G_1) \parallel L(G_2) \parallel L(\boldsymbol{p}) \neq \varnothing$ for pattern $\boldsymbol{p} = a^*$?*

PROOF. Let us first observe that since the alphabets of G_1, G_2, and $\boldsymbol{p}$ are given by $\{a\}$, the asynchronous product ($\|$) behaves exactly like language intersection ($\cap$). Hence, to ease the reader understanding, the proof uses $\cap$ instead of $\|$. The proof is by reduction from the 0–1 Knapsack problem: let $\{o_1, \dots, o_m\}$, $\{w_1, \dots, w_m\}$, W be an instance of the 0-1 Knapsack problem, and b be the maximal number of bits needed to encode any of the integers $\{w_1, \dots, w_m, W\}$. Define G to be the grammar over unary alphabet $\{a\}$ with productions given by the union of the sets (8) through (13) shown later. Intuitively, a derivation of G nondeterministically selects a subset of objects as follows. The object o_i is selected by applying the production $S_i \to S_i^{(b)}$ (8) and is omitted by applying $S_i \to S_{i+1}$ (9). If o_i has been selected, then the derivation outputs a^{w_i} through the variable $S_i^{(b)}$ using the productions in (10) and (11) and then comes back to S_{i+1} using production (12). Formally, we have $S_i^{(b)} \Rightarrow^* a^{w_i}\, S_{i+1}$. Indeed, observe that $w_i = \sum_{j=0}^{b} j\text{th bit of } w_i \times 2^j$, and the productions of (10)–(11) follow the binary encoding of w_i: if the jth bit is 0, then the derivation moves to the next bit, and if it is 1, then the grammar outputs a^{2^j} through A_j. The productions of (13) make use of a well-known encoding to ensure $\{w \mid A_k \Rightarrow^* w\} = \{a^{2^k}\}$ for every $0 \le k \le b$. Finally, the axiom of G is S_1.

$$\left\{S_i \to S_i^{(b)} \mid 1 \le i \le m\right\} \tag{8}$$

$$\{S_i \to S_{i+1} \mid 1 \le i \le m-1\} \cup \{S_m \to \varepsilon\} \tag{9}$$

$$\left\{S_i^{(k)} \to A_k\, S_i^{(k-1)} \mid 1 \le i \le m \wedge 1 \le k \le b \wedge \text{bit } k \text{ of } w_i \text{ is } 1\right\} \tag{10}$$

$$\left\{S_i^{(k)} \to S_i^{(k-1)} \mid 1 \le i \le m \wedge 1 \le k \le b \wedge \text{bit } k \text{ of } w_i \text{ is } 0\right\} \tag{11}$$

$$\left\{S_i^{(0)} \to S_{i+1} \mid 1 \le i \le m-1\right\} \cup \left\{S_m^{(0)} \to \varepsilon\right\} \tag{12}$$

$$\{A_k \to A_{k-1}\, A_{k-1} \mid 1 \le k \le b\} \cup \{A_0 \to aZ\} \cup \{Z \to \varepsilon\} \tag{13}$$

We now turn to W and define the grammar G_W by

$$\left\{W^{(k)} \to A_k\, W^{(k-1)} \mid 1 \le k \le b \wedge \text{bit } k \text{ of } W \text{ is } 1\right\} \tag{14}$$

$$\left\{W^{(k)} \to W^{(k-1)} \mid 1 \le k \le b \wedge \text{bit } k \text{ of } W \text{ is } 0\right\} \cup \left\{W^{(0)} \to \varepsilon\right\}, \tag{15}$$

where $W^{(b)}$ is the axiom. From the reasoning earlier, we find that $L(G) = \{a^W\}$.

Clearly, G and G_W can be computed in polynomial time and are in program normal form. Moreover, it is easily seen that $L(G) \cap L(G_W) \cap L(\boldsymbol{p}) \neq \varnothing$ if and only if there is a subset $S \subseteq \{o_1, \dots, o_m\}$ such that the weight of S is W. □

Notice the similarities and differences with Theorem 4.1. Theorem 5.2 shows NP-hardness using a constant number of context-free grammars over a unary alphabet; hence, the pattern necessarily is a^*. Although the NP-result of Theorem 4.1 is also on the unary alphabet, it uses d regular languages.

5.2. Fixed-Sized Threads

We assume that *st* is fixed. The most interesting case is that in which neither d nor *pa* is fixed, that is, CMP($\widehat{st}, d, pa$). This corresponds to multithreaded programs in which each thread has at most a fixed size, but the number of threads is arbitrary. We prove that this case remains NP-complete. Actually, we prove that NP-completeness holds even for the special case of regular grammars, which will be important for the refined multiparameter analysis of Section 6.

Theorem 5.3. *The following problem is NP-hard:*
Instance: *Regular grammars $G_1, \dots, G_d$ in program normal form of fixed size, a pattern* $\boldsymbol{p}$ *over the union of all their alphabets.*
Question: *Is $\|_{i=1}^{d} L(G_i) \parallel L(\boldsymbol{p}) \neq \varnothing$?*

Proof. By reduction from 3-CNF-SAT. Let Ψ be a propositional formula with k variables $x_1, \dots, x_k$ and d clauses $c_1, \dots, c_d$. We define for each clause c_i a regular grammar G_i such that $\|_{i=1}^{d} L(G_i) \neq \varnothing$ if and only if Ψ is satisfiable. Let $c_i = \ell_{i_1} \vee \ell_{i_2} \vee \ell_{i_3}$, where $1 \leq i_1 < i_2 < i_3 \leq k$ and $\ell_j \in \{x_j, \overline{x}_j\}$. Define G_i as a regular grammar over the alphabet $\Sigma_i = \{x_{i_1}, \overline{x}_{i_1}, x_{i_2}, \overline{x}_{i_2}, x_{i_3}, \overline{x}_{i_3}\}$ accepting the language consisting of all the possible valuations of x_{i_1}, x_{i_2} and x_{i_3} (in that order) making c_i true. Note that this language contains no more than eight words and so G_i is of fixed size.

We then go on proving that $\|_{i=1}^{d} L(G_i) \neq \emptyset$ if and only if Ψ is satisfiable.

First, we claim that no word of $\|_{i=1}^{d} L(G_i)$ contains more than one literal of the same variable. Assume some $w \in \|_{i=1}^{d} L(G_i)$ contains occurrences of both x_j and $\overline{x}_j$. Choose Σ_i such that $\{x_j, \overline{x}_j\} \subseteq \Sigma_i$, and let w_i be the projection of w onto Σ_i. Then w_i also contains occurrences of both x_j and $\overline{x}_j$ and, by the definition of $\parallel$, we have $w_i \in L(G_i)$. But this contradicts that w_i is a valuation of the literals of c_i making c_i true, and the claim is proved.

Consider now any word $w \in \|_{i=1}^{d} L(G_i)$. By the previous claim, w encodes an assignment of Ψ. Moreover, for every clause c_i, the projection w_i is a satisfying assignment of c_i. So w is a satisfying assignment of Ψ.

Finally, we have to find a pattern $\boldsymbol{p}$ that preserves emptiness; that is, we want a pattern $\boldsymbol{p}$ such that $\|_{i=1}^{d} L(G_i) \neq \emptyset$ if and only if $\|_{i=1}^{d} L(G_i) \parallel L(\boldsymbol{p}) \neq \emptyset$. For this, it is easily seen that $\boldsymbol{p} = x_1^*(\overline{x}_1)^* \cdots x_k^*(\overline{x}_k)^*$ is such a pattern, and we are done. $\square$

In the remaining cases, at least one of d and pa is fixed as well. Consider the case in which we fix the length pa of $\boldsymbol{p}$. This means that the number n of words in $\boldsymbol{p}$ is fixed and so is the length of each of them. Since st is fixed, so is each st_i, which means that G_i has a fixed number of productions. Moreover, because $\boldsymbol{p}$ has fixed size, we conclude from the construction of Appendix A.1 that G_i' has a fixed number of productions. Observe that fixing pa also yields that all grammars G_i' are over a common alphabet of fixed size al. Therefore, we obtain that there is only a fixed number of possible grammars G_i', and hence a fixed number of possible instances of nDPK. This means that all the instances of CMP$(\widehat{st}, d, \widehat{pa})$ are mapped to a fixed finite number of instances of nDPK in polynomial time as shown in Lemmas 4.3 and 4.4. It follows that the problem is trivially polynomial.

We leave the case in which we fix d but not pa open. The case is not very interesting in our setting, because if both st and d are fixed, then we only have a fixed finite number of possible tuples of grammars. On the other hand, since the length of the pattern is arbitrary, the reduction from CMP to nDPK maps the instances of CMP into an infinite number of instances of nDPK, and we cannot apply the argument above.

6. A REFINED MULTIPARAMETER ANALYSIS: TAKING PROCEDURAL STRUCTURE INTO ACCOUNT

In this section, we examine in more detail the left subtree of Figure 2 (grammars of arbitrary size). The question is whether we can get better upper bounds for CMP when the procedural call graph of the grammars is "simple." We refine our analysis by considering also the parameters lsp and pr.

We first show that if the pattern can be arbitrarily long, then fixing pr and/or lsp does not help: CMP remains NP-complete (Theorem 6.2 in Section 6.1).

We then proceed to prove that if the length of the pattern is fixed, then fixing *pr* or *lsp* *does help*. We do so by proving that in this case, CMP can be solved in $st^{O(lsp+\lceil\log(pr+1)\rceil)}$ time. Therefore, if *pr* is fixed, then, since $pr \geq lsp$, we get a polynomial algorithm in *st*. If only *lsp* is fixed, then we get a $st^{O(\lceil\log(pr+1)\rceil)}$ algorithm, which is subexponential in *st*.

This complexity result is the main result of the article, and we divide its presentation into two parts. The case of regular grammars is considered in Section 6.2, and the general case in Section 6.3. The case of regular grammars is proved by reduction to the emptiness problem for bounded-reversal counter machines, which is known to be polynomial [Gurari and Ibarra 1981]. Intuitively, bounded-reversal counter machines are counter machines whose counters can be in "nonincreasing" or "nondecreasing" mode and can only change mode a bounded number of times. The counter machine guesses a sequence of words generated by the grammars and uses the counters to check that, for each letter of the alphabet, all the words contain the same number of occurrences of the letter. The general case is proved by reduction to the case of regular grammars. The key of the proof is a procedure that, given a grammar $G = (\mathcal{X}, \Sigma, \mathcal{P}, S)$ in program normal form, constructs in time $st^{O(lsp+\lceil\log(pr+1)\rceil)}$ a regular grammar A such that $\Pi(L(A)) = \Pi(L(G))$. While the existence of A is an easy consequence of Parikh's theorem [Parikh 1966], the complexity bound requires one to apply a recent alternative proof that explicitly constructs the grammar A [Esparza et al. 2011].

6.1. Arbitrary Pattern Length

The proof of our first result is by reduction from the bounded Post Correspondence Problem [Garey and Johnson 1979].

Definition 6.1 *(Bounded Post Correspondence Problem).*
Instance. Two sequences $a = (a_1, \ldots, a_d)$ and $b = (b_1, \ldots, b_d)$ of words over an alphabet Σ and a positive integer $K \leq d$.
Question: Is there a nonempty sequence $i_1, \ldots, i_k$ of $k \leq K$ (not necessarily distinct) integers, each between 1 and d, such that $a_{i_1}a_{i_2}\ldots a_{i_k} = b_{i_1}b_{i_2}\ldots b_{i_k}$?

THEOREM 6.2. *The following problem is NP-hard:*
Instance: *Two context-free grammars G_1, G_2 in program normal form, each of them with one procedure variable, a regular grammar R in program normal form, and a pattern $\boldsymbol{p}$ over the union of all previous alphabets.*
Question: *Is $L(G_1) \parallel L(G_2) \parallel L(R) \parallel L(\boldsymbol{p}) \neq \emptyset$?*

PROOF. Let a, b, K be an instance of the bounded Post Correspondence Problem. Assume $\Gamma = \{1, \ldots, d\}$ is disjoint from Σ. We construct the context-free grammars:

$$G_1 = (\{X\}, \Sigma \cup \Gamma, \{X \to a_i\ X\, i \mid 1 \leq i \leq d\} \cup \{X \to \varepsilon\}, X)$$
$$G_2 = (\{Y\}, \Sigma \cup \Gamma, \{Y \to b_i\ Y\ i \mid 1 \leq i \leq d\} \cup \{Y \to \varepsilon\}, Y).$$

Define the regular grammar R over Γ such that $L(R) = \bigcup_{i=1}^{K} \Gamma^i$, and the pattern $\boldsymbol{p}$ over $\Sigma \cup \Gamma$ such that $\boldsymbol{p} = (a_1^* \ldots a_d^*)^K(1^* \ldots d^*)^K$. The language R over Γ coincides with the set of nonempty sequences $i_1, \ldots, i_k$ of $k \leq K$ (not necessarily distinct) integers, each between 1 and d. So by enforcing that a witness in $L(G_1) \parallel L(G_2) \parallel L(R)$, if any, contains no more than K occurrences of integers between 1 and d, R implements the bound K. The pattern $\boldsymbol{p}$ is defined so as to not exclude any possible witness in $L(G_1) \parallel L(G_2) \parallel L(R)$, that is, $L(G_1) \parallel L(G_2) \parallel L(R) \neq \emptyset$ if and only if $L(G_1) \parallel L(G_2) \parallel L(R) \parallel L(\boldsymbol{p}) \neq \emptyset$. Observe that, since $K \leq d$, the size of $\boldsymbol{p}$ is polynomial in the size of the instance. Notice that G_1 and G_2 can be easily put in program normal form: replace a production $X \to a_i\ X\, i$ by productions $X \to a_i\ X_i'$, $X_i' \to X X_i''$, $X_i'' \to i\, Z$, $Z \to \varepsilon$, where X_i', X_i'', and Z are fresh variables. Finally, observe that X is the only procedure variable. It follows easily from the construction that $L(G_1) \parallel L(G_2) \parallel L(R) \parallel L(\boldsymbol{p}) \neq \emptyset$ if and only if the bounded PCP instance is positive. $\square$

Notice that in this reduction, neither the number of words in $\boldsymbol{p}$ nor their length is fixed. By means of a more involved reduction, it is possible to show NP-hardness with a single word only (but arbitrarily long). The proof is rather technical, and we present it in Appendix A.2.

6.2. Fixed-Length Patterns: The Case of Regular Grammars

We prove that CMP($st, \widehat{d}, \widehat{pa}$) can be solved in $st^{O(lsp+\lceil \log(pr+1)\rceil)}$ time. This section considers the case in which the grammars have the simplest possible procedural structure: grammars have a single procedure variable (the axiom). This corresponds to the case of regular grammars, that is, the case $pr = lsp = 1$. So our task is to provide an st^c algorithm for some constant c or, since d and pa are fixed, a $st^{f(d,pa)}$ algorithm for some function f.

As a first step, we observe that, because of the reduction from CMP to nDPK shown in Section 4.2, it suffices to prove the following: given regular grammars $A_1, \ldots, A_d$ over a common alphabet Σ of size al and some subalphabet Σ', $\bigcap_{i=1}^{d} \Pi_{\Sigma'}(L(A_i)) = \emptyset$ can be checked in time $O((|A_1| + \cdots + |A_d|)^{h(d,al)})$ for some function h (observe that instances of CMP with fixed pa are reduced to instances of nDPK with fixed al). The proof is by reduction to the emptiness problem for bounded-reversal counter machines, which is known to be polynomial [Gurari and Ibarra 1981].

An *m-counter machine* is a tuple $M = (Q, \delta, q_0, F)$, where Q is a finite nonempty set of states, $\delta \subseteq Q \times \{1, \ldots, m\} \times \{inc, dec, skip, zero\} \times Q$ is the set of transitions, q_0 is the initial state, and F is the set of final states. A *configuration* of M is a tuple $(q, n_1, \ldots, n_m)$, where $q \in Q$ and $n_i \in \mathbb{N}$ for every $1 \leq i \leq m$. Intuitively, $(q, i, op, q') \in \delta$ means that if M is in state q and op is enabled[7] w.r.t. the value of counter i, then M updates counter i according to op and moves to state q'. Formally, given two configurations $c = (q, n_1, \ldots, n_m)$ and $c' = (q', n'_1, \ldots, n'_m)$, we write $c \to c'$ if there is a transition $(q, i, op, q') \in \delta$ such that one of the following holds:

—$op = inc$, $n'_i = n_i + 1$, and $n'_j = n_j$ for every $j \in \{1, \ldots, m\}$, $j \neq i$;
—$op = dec$, $n_i > 0$, $n'_i = n_i - 1$, and $n'_j = n_j$ for every $j \in \{1, \ldots, m\}$, $j \neq i$;
—$op = skip$, and $n'_j = n_j$ for every $j \in \{1, \ldots, m\}$;
—$op = zero$, $n_i = 0$, and $n'_j = n_j$ for every $j \in \{1, \ldots, m\}$.

A *run* of M is a sequence $c_0 \to c_1 \to \cdots \to c_n$ such that $c_0 = (q_0, 0, \ldots, 0)$. Moreover, the previous run is *accepting* if the state of c_n is given by $(q_f, 0, \ldots, 0)$ where $q_f \in F$.

A run is *k-bounded-reversal* if every counter alternates at most k times from a nonincreasing mode to a nondecreasing mode (or vice versa).

THEOREM 6.3 (BY GURARI AND IBARRA [1981]). *Let m and r be fixed positive integers. Given an m-counter machine M, the problem of deciding whether M has an r-bounded-reversal accepting run can be solved in polynomial time.*

The following proposition reduces nDPK for regular grammars to the emptiness problem for bounded-reversal counter machines.

PROPOSITION 6.4. *Let $A_1, \ldots, A_d$ be regular grammars over common alphabet Σ and subalphabet $\Sigma' \subseteq \Sigma$. There is a counter machine M of size $O(|A_1| + \cdots + |A_d|)$ with*

[7] *inc* and *skip* are enabled in every state, while *dec* and *zero* are not.

$2 \cdot |\Sigma'|$ *counters such that* $\bigcap_{i=1}^{d} \Pi_{\Sigma'}(L(A_i)) \neq \emptyset$ *if and only if M has a d-bounded-reversal accepting run.*

Proof. The construction is simple, and we only sketch the details. M will operate on two sets of counters, C_1 and C_2, where $C_i = \{c_{ia} \mid a \in \Sigma'\}$ for $i = 1, 2$.

Assume $A_i = (Q_i, \Sigma, \delta_i, q_{0i})$ for each $1 \leq i \leq d$; we first construct a regular grammar A over alphabet Σ with variables $\bigcup_{i=1}^{d} Q_i$, axiom q_{01}, and production rules obtained by replacing in $\bigcup_{i=1}^{n} \delta_i$ productions $q \to \varepsilon$ where $q \in Q_i$ and $1 \leq i < d$ with $q \to q_{0i+1}$. It is routine to check that $L(A) = L(A_1) \cdot \ldots \cdot L(A_d)$. For simplicity, assume d is even (the case where d is odd is similar). Hence, we define the counter machine M from A (which is now seen as an automaton) as follows. Replace

—each production $q \to a\, q'$ of δ_1 by a transition that increases c_{1a} if $a \in \Sigma'$ else *skip*;
—each production $q \to a\, q'$ of δ_d by a transition that decreases c_{1a} if $a \in \Sigma'$ else *skip*;
—each production $q \to q'$ of δ_i, where $1 \leq i \leq d$, by a transition *skip*;
—each production $q \to a\, q'$ of δ_i, where $1 < i < d$ and i even, by a sequence of two transitions, the first decreasing c_{1a} and the second increasing c_{2a} if $a \in \Sigma'$ else *skip*;
—each production $q \to a\, q'$ of δ_i, where $1 < i < d$ and i odd, by a sequence of two transitions, the first decreasing c_{2a} and the second increasing c_{1a} if $a \in \Sigma'$ else *skip*;
—each production $q \to q'$ where $q \in Q_i$ and $q' \in Q_{i+1}$ with $1 < i < d$ and i even by a sequence of $|\Sigma'|$ transitions checking that c_{1a} equals to 0 for each $a \in \Sigma'$;
—each production $q \to q'$ where $q \in Q_i$ and $q' \in Q_{i+1}$ with $1 < i < d$ and i odd by a sequence of $|\Sigma'|$ transitions checking that c_{2a} equals to 0 for each $a \in \Sigma'$. □

From Proposition 6.4 and Theorem 6.3, it follows that:

Corollary 6.5. *Let* $A_1, \ldots, A_d$ *be regular grammars over common alphabet* Σ *and subalphabet* $\Sigma' \subseteq \Sigma$. *Assume d and al, the size of* Σ, *are fixed. Then nDPK, which is the problem of deciding* $\bigcap_{i=1}^{d} \Pi_{\Sigma'}(L(A_i)) \neq \emptyset$, *can be solved in polynomial time.*

6.3. Fixed-Length Patterns: The Case of Context-Free Grammars

We show that CMP$(st, \widehat{d}, \widehat{pa})$ can be solved in $st^{O(lsp+\lceil \log(pr+1) \rceil)}$ for context-free grammars in program normal form.

Before proceeding, the following notation is called for. Given two languages L_1 and L_2 over Σ, we write $L_1 =_\Pi L_2$ (resp. $L_1 \subseteq_\Pi L_2$) to denote that $\Pi(L_1)$ is equal to (resp. included in) $\Pi(L_2)$. Also, given $w, w' \in \Sigma^*$, we abbreviate $\{w\} =_\Pi \{w'\}$ to $w =_\Pi w'$.

Let $G_1, \ldots, G_d$ and $\boldsymbol{p}$ be an instance of CMP$(st, \widehat{d}, \widehat{pa})$, and let $G'_1, \ldots, G'_d$ be the instance of nDPK obtained as a result of the reduction of Section 4.2. Observe that since pa is fixed, $G'_1, \ldots, G'_d$ share the common alphabet $\Sigma \cup \widetilde{\Sigma}$ of fixed size. Furthermore, following Lemma 4.4 and since pa is fixed, we find that $st' \in O(st)$, $lsp' \in O(lsp)$, and $pr' \in O(pr)$. Therefore, it suffices to show that $\bigcap_{i=1}^{d} \Pi_{\widetilde{\Sigma}}(L(G'_i)) \neq \emptyset$ can be decided in $st'^{O(lsp'+\lceil \log(pr'+1) \rceil)}$ time. We proceed by reduction to the regular case of Section 6.2. We compute for each G'_i a regular grammar (or nondeterministic automaton) A_i in time $st_i'^{O(lsp'_i+\lceil \log(pr'_i+1) \rceil)}$ such that $L(A_i) =_\Pi L(G'_i)$. It then suffices to apply the result of Corollary 6.5 for regular grammars.

So we are left with the following problem: given a grammar $G = (\mathcal{X}, \Sigma, \mathcal{P}, S)$ in program normal form, construct in time $st^{O(lsp+\lceil \log(pr+1) \rceil)}$ a regular grammar A such that $L(A) =_\Pi L(G)$. For this we proceed as follows. Let $K = lsp + \lceil \log(pr+1) \rceil$. We define a family of regular grammars $A^k_G = (Q^k_G, \Sigma, \delta^k_G, q_0)$ such that A^k_G can be computed in time $st^{O(k)}$ and prove that $L(A^K_G) =_\Pi L(G)$.

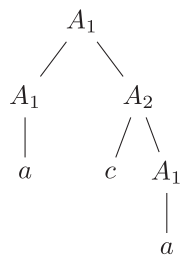

Fig. 3. A parse tree of $A_1 \to A_1A_2|a,\ A_2 \to bA_2aA_2|cA_1$, where A_1 is the axiom.

Definition 6.6. The regular grammar $A_G = (\mathcal{Q}_G, \Sigma, \delta_G, q_0)$ is defined as follows:

—$\mathcal{Q}_G$ is the set of all multisets $\mathbf{m} \in \mathbb{M}[\mathcal{X}]$ and the axiom q_0 is the multiset $\Pi(S)$;
—$\delta_G = \{\emptyset \to \epsilon\} \cup \delta'_G$, where δ'_G contains a production $\mathbf{m} \to \alpha \cdot \mathbf{m}'$ if and only if $\mathcal{P}$ contains a production $X \to \alpha\beta$, such that $\alpha \in \Sigma^*$, $\beta \in \mathcal{X}^*$, and $\mathbf{m}' \oplus \Pi(X) = \mathbf{m} \oplus \Pi(\beta)$.

For every $k \geq 1$, the grammar $A^k_G = (\mathcal{Q}^k_G, \Sigma, \delta^k_G, q_0)$ is the restriction of A to the multisets $\mathbf{m} \in \mathbb{M}[\mathcal{X}]$ containing at most k elements.

Observe that $|\mathcal{Q}^k_G| = O(|\mathcal{X}|^k)$ and $|\delta^k_G| \leq |\mathcal{Q}^k_G|^2 \cdot |\Sigma|$. Therefore, A^k_G can indeed be computed in time $st^{O(k)}$.

We have to prove $L(A^K_G) =_\Pi L(G)$. It was shown in Esparza et al. [2011, Proposition 2.1] that $L(A^k_G) \subseteq_\Pi L(G)$ holds for every $k \geq 1$, and moreover that the equality $L(A^k_G) =_\Pi L(G)$ holds for $k = |\mathcal{X}| + 1$. However, this is not good enough for our purposes, because $|\mathcal{X}|$ can be arbitrarily larger than K. So we have to strengthen this result. Since $L(A^K_G) \subseteq_\Pi L(G)$, it suffices to prove $L(G) \subseteq_\Pi L(A^K_G)$. In Section 6.3.1, we show that this $L(G) \subseteq_\Pi L(A^K_G)$ is implied by a condition on the *dimension* of the parse trees of G, a notion taken from Esparza et al. [2011]. In Section 6.3.2, we prove that this condition holds.

6.3.1. A Technique for Proving $L(G) \subseteq_\Pi L(A^K_G)$. We first introduce some definitions about parse trees. A labeled tree (t, λ) consists of a finite tree t and a labeling function λ, which maps each node of t onto some label taken from a finite set. A labeled tree is a *parse tree* for $G = (\mathcal{X}, \Sigma, \mathcal{P}, S)$ if it satisfies the following properties: (1) the axiom labels the root, nodes with children are labeled by variables, leaves are labeled by an element of $\Sigma \cup \{\varepsilon\}$, and nodes labeled by ε are the only child of their parent; and (2) every node n with children $\{n_1, \ldots, n_k\}$ where $k \geq 1$ is such that $(\lambda(n), \lambda(n_1) \ldots \lambda(n_k)) \in \mathcal{P}$. In the sequel, we mean parse tree when we write tree. Because it is convenient, we identify nodes and the subtrees of which they are root. The *yield* of tree t is denoted $\Delta(t)$ and is given by the concatenation from left to right of the labels of the leaves of t. Every derivation $S \Rightarrow^* w$ can be parsed into a tree t such that $\Delta(t) = w$. We extend the definition of yield to sets of trees as follows: let T be a set of trees of G; then $\Delta(T) = \{\Delta(t) \mid t \in T\} \subseteq L(G)$. Figure 3 shows the tree of the derivation $A_1 \Rightarrow A_1A_2 \Rightarrow aA_2 \Rightarrow acA_1 \Rightarrow aca$.

Let t be a tree for a grammar G. A child of t is called *proper* if its root is not a leaf, that is, if it is labeled with a variable. The *dimension* $d(t)$ of a tree t is inductively defined as follows. If t has no proper children, then $d(t) = 0$. Otherwise, let $t_1, t_2, \ldots, t_r$ be the proper children of t sorted such that $d(t_1) \geq d(t_2) \geq \cdots \geq d(t_r)$. Then

$$d(t) = \begin{cases} d(t_1) & \text{if } r = 1 \text{ or } d(t_1) > d(t_2) \\ d(t_1) + 1 & \text{if } d(t_1) = d(t_2) \end{cases} .$$

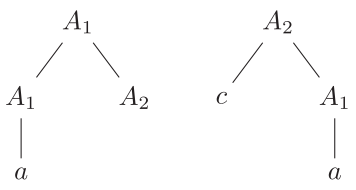

Fig. 4. A decomposition t_1, t_2 such that $t = t_1 \cdot t_2$ is the parse tree of Figure 3.

The set of trees of G of dimension k is denoted by $\mathcal{T}^{(k)}$, and the set of all trees is denoted by $\mathcal{T}$. It is routine to check that $\Delta(\mathcal{T}) = L(G)$.

Our technique is based on "tree surgery" of a tree t: intuitively, surgery cuts a "piece" out from a subtree of t and "inserts" it into another subtree. The description of the procedure requires some notations.

We write $t = t_1 \cdot t_2$ to denote that t_1 is a tree, except that exactly one leaf ℓ is labeled by a variable, say, A, instead of a terminal; that the tree t_2 is a tree with root A; and that the tree t is obtained from t_1 and t_2 by replacing the leaf ℓ of t_1 by the tree t_2. Figure 4 shows an example.

Finally, say that two trees t, t' are *equivalent* if (1) the label of their root coincides and (2) the two multisets given by the labels of all the nodes of t and t' coincide. Note that if t and t' are equivalent, then $\Delta(t) =_\Pi \Delta(t')$.

The results of Esparza et al. [2011] link the dimension of trees to the languages of the regular grammars A_G^k: their result implies that, for any G in program normal form, $\Delta(\mathcal{T}^{(i)}) \subseteq_\Pi L(A_G^{k+1})$ holds for every $0 \le i \le k$. Applying this result, we immediately obtain:

PROPOSITION 6.7. *If $\Delta(\mathcal{T}) \subseteq_\Pi \bigcup_{i=0}^{k} \Delta(\mathcal{T}^{(i)})$ for some $k \ge 0$, then $L(A_G^{k+1}) =_\Pi L(G)$.*

PROOF. If $\Delta(\mathcal{T}) \subseteq_\Pi \bigcup_{i=0}^{k} \Delta(\mathcal{T}^{(i)})$, then, by Esparza et al. [2011, Lemmas 2.2 and 2.4], we have $\Delta(\mathcal{T}) \subseteq_\Pi L(A_G^{k+1})$, and hence $L(G) \subseteq_\Pi L(A_G^{k+1})$ and finally $L(A_G^{k+1}) =_\Pi L(G)$ since $L(A_G^k) \subseteq_\Pi L(G)$ for every $k \ge 1$ by Esparza et al. [2011, Proposition 2.1]. $\square$

This proposition yields a technique for proving $L(G) \subseteq_\Pi L(A_G^{k+1})$: show that for every tree t there is another tree t' of dimension at most k such that $\Delta(t) =_\Pi \Delta(t')$. Therefore, $L(G) \subseteq_\Pi L(A_G^K)$ holds if $\Delta(\mathcal{T}) \subseteq_\Pi \bigcup_{i=0}^{K-1} \Delta(\mathcal{T}^{(i)})$.

6.3.2. Proving $\Delta(\mathcal{T}) \subseteq_\Pi \bigcup_{i=0}^{K-1} \Delta(\mathcal{T}^{(i)})$. We start with a lemma showing that when the dimension of a tree is "high enough," then there exists a path from the root to a leaf where the same procedure variable repeats.

LEMMA 6.8. *If a tree t of G is such that $d(t) \ge lsp$, then there is a path from the root to a leaf in which two distinct nodes n_a and n_b are such that $\lambda(n_a) = \lambda(n_b) = Z$ for some procedure variable Z.*

PROOF. The definition of dimension and $d(t) \ge lsp$ show there must exist a path $\pi = n_0 n_1 \dots n_s$ in t such that n_0 is the root of t and n_s has two children n_ℓ and n_r (in the left-to-right order) such that $d(n_\ell) = d(n_r) = lsp - 1$. Now by repeating the previous reasoning (taking n_ℓ as the root), one can build a path π' from n_0 (the root of t) to a leaf along which there exist $lsp + 1$ nodes $\{n_0, n_{i_1}, \dots, n_{i_{lsp}}\}$ such that each of them is labeled by a procedure variable. Finally, we necessarily have two distinct nodes n_a and n_b along π' such that $\lambda(n_a) = \lambda(n_b) = Z$ and Z is a procedure variable. $\square$

We need a further property of the dimension of a tree:

LEMMA 6.9. *Let t be a tree such that $d(t) \geq a + b$ for some positive integers a and b. Then t has at least 2^a disjoint subtrees of dimension at least b (where two subtrees of t are disjoint if their sets of nodes are disjoint).*

PROOF. We say that t is (a, b)-*good* if there is a set of 2^a pairwise disjoint subtrees of t of dimension at least b and call such set an (a, b)-*witness* of t. We proceed by induction on the number of nonleaf nodes of t. If t has only one nonleaf node, then it has no proper child, and hence $d(t) = 0$, which implies $a = b = 0$, and so the singleton set $\{t\}$ is an $(0, 0)$-witness of t. If t has more than one nonleaf node, then by the definition of dimension, there are two cases:

—t has two children t_1, t_2 such that $d(t_1) = d(t_2) = d(t) - 1$. Since the height of t_1 and t_2 is smaller than the height of t, we can apply the induction hypothesis, and so t_1 and t_2 are both $(a - 1, b)$-good. Let S_1, S_2 be $(a - 1, b)$-witnesses of t_1 and t_2. Since t_1 and t_2 are disjoint, the elements of $S_1 \cup S_2$ are pairwise disjoint, and hence $S_1 \cup S_2$ is an (a, b)-witness of t.
—t has an only child t' such that $d(t) = d(t')$. In this case, by induction hypothesis, t' is (a, b)-good. Let S' be an (a, b)-witness of t'. Then, S' is also an (a, b)-witness of t, and so t is (a, b)-good. □

Given a tree t, the *position* of a node is the word of $\{0, 1\}^*$ inductively defined as follows: the position of the root is the empty word; if the node at position w has two children, then the positions of its left and right children are $w0$ and $w1$, respectively; if the node at position w has one child, then the position of its child is $w1$.

The following theorem is the key to our result:

THEOREM 6.10. *Let G be a context-free grammar in program normal form. Define $K = lsp + \lceil \log(pr + 1) \rceil$. Then $\Delta(\mathcal{T}) \subseteq_\Pi \bigcup_{i=0}^{K-1} \Delta(\mathcal{T}^{(i)})$.*

PROOF. We need to prove that for every tree $t \in \mathcal{T}$, there exists a tree t' such that $\Delta(t) =_\Pi \Delta(t')$ and $d(t') \leq K - 1$. It suffices—to prove the result—to show that for every tree t, there exists an equivalent tree t' such that $d(t') \leq K - 1$.

Assume $d(t) \geq K$. Let $a = \lceil \log(pr + 1) \rceil$ and $b = lsp$. We conclude from $d(t) \geq K$ that $d(t) \geq a + b$, and hence, by Lemma 6.9, that t has at least $pr + 1$ disjoint subtrees of dimension at least lsp.

Next, Lemma 6.8 shows that a tree of dimension at least lsp contains a path that visits two nodes labeled by the same procedure variable. We call this variable a *repeat*.

The procedure is as follows:

While t has at least $pr + 1$ disjoint subtrees of dimension at least lsp, do:

—Pick two disjoint subtrees t_1, t_2 of t of dimension at least lsp containing a repeat *of the same procedure variable Z*. W.l.o.g. assume that the position of t_1 is lexicographically smaller than the position of t_2.
(Such trees exist by the earlier discussion and the pigeonhole principle, because the number of procedure variables is pr.)
—Factor t_1 and t_2 into $t_1^a \cdot (t_1^b \cdot t_1^c)$ and $t_2^a \cdot (t_2^b \cdot t_2^c)$, respectively, such that, for $i = 1, 2$, the trees t_i^b and t_i^c have their root labeled by Z.
—Let $t_1' = t_1^a \cdot t_1^c$ and $t_2' = t_2^a \cdot (t_2^b \cdot (t_1^b \cdot t_2^c))$.
—Simultaneously substitute t_1' for t_1 and t_2' for t_2 in t. Call the result t'.
—$t := t'$.

It is easy to see that since t_1^b, t_1^c, and t_2^c are labeled by Z, both t_1' and t_2' are parse trees of G. Moreover, t and t' are equivalent.

If the procedure terminates, then we are done, because it can only terminate with a tree t having at most pr disjoint subtrees of dimension at least lsp, and therefore, by Lemma 6.9, we have $d(t) \leq K - 1$. To prove termination, we first define a preorder on trees. Let $\#(t, w)$ denote the number of nodes of the subtree of t rooted at position w (if t does not have a node at position w, then $\#(t, w) = 0$). We write $t \prec t'$ if the lexicographically smallest position w such that $\#(t, w) \neq \#(t', w)$ satisfies $\#(t, w) < \#(t', w)$. To show that $\prec$ is a preorder, observe first that $t \prec t'$ implies $t' \not\prec t$. For transitivity, assume $t \prec t' \prec t''$. Then there exist lexicographically smallest positions w, w' such that $\#(t, w) < \#(t', w)$ and $\#(t', w') < \#(t'', w')$; if w is lexicographically smaller than or equal to w', then $\#(t, w) < \#(t', w) = \#(t'', w)$ and $\#(t, w'') = \#(t', w'') = \#(t'', w'')$ for every w'' lexicographically smaller than w; if w' is lexicographically smaller than w, then $\#(t, w') = \#(t', w') < \#(t'', w')$ and $\#(t, w'') = \#(t', w'') = \#(t'', w'')$ for every w'' lexicographically smaller than w'. In both cases we have $t \prec t''$.

Next, we show that, in the previous procedure, $t \prec t'$. It follows from the equivalence of t and t' that t' has as many nodes as t, and hence that the procedure terminates since the number of trees with a fixed number of nodes is finite. Therefore, let us show $t \prec t'$. Since t_1 and t_2 are disjoint, they have a least common ancestor u. Moreover, since t_1's position is lexicographically smaller than t_2, we have the following: u has two proper children, its left child u_1 is an ancestor of t_1, and its right child u_2 is an ancestor of t_2. Let w_1 be the position of u_1. Observe that we necessarily have that w_1 ends with 0. Then we have $\#(t, w_1) < \#(t', w_1)$ (because t_1^b is a subtree of u_1), and $\#(t, w) = \#(t', w)$ for any position w lexicographically smaller than w_1 (because the subtrees at these positions contain either both t_1 and t_2 as subtrees or none of them, and so tree surgery does not change their number of nodes). So $t \prec t'$. $\square$

Collecting these results, we get:

THEOREM 6.11. *Let G be a context-free grammar in program normal form and let $K = lsp + \lceil \log(pr + 1) \rceil$. We have $L(A_G^K) =_\Pi L(G)$. Moreover, A_G^K can be constructed in $st^{O(K)}$ time.*

PROOF. Theorem 6.10 and Proposition 6.7 show that $L(G) =_\Pi L(A_G^K)$.

The runtime follows immediately from the fact that the number of variables of the regular grammar A_G^K of Definition 6.6 for a context-free grammar G with n variables is $O(n^K)$. $\square$

Finally, putting together the results of Theorem 6.11 and Corollary 6.5, we obtain that:

THEOREM 6.12. *Let $G_1, \ldots, G_d$ be context-free grammars in program normal form over a common alphabet Σ of size al and a subalphabet $\Sigma' \subseteq \Sigma$. Assume d and al are fixed. Then nDPK, which is the problem of deciding $\bigcap_{i=1}^{d} \Pi(L(G_i)) \neq \emptyset$, can be solved in $st^{O(lsp+\lceil \log(pr+1) \rceil)}$ time.*

7. CONCLUSIONS

We have studied the complexity of pattern-based verification, an approach to the verification of multithreaded programs with counters essentially introduced by Kahlon [2009b]. The approach checks whether the program has executions conforming to a pattern, a regular expression of the form $w_1^* \ldots w_n^*$ over an alphabet of reads and writes to a global store. The pattern can be supplied by the programmer or be constructed by an automatic tool. The latter has been implemented by Long et al. [2012] in their CEGAR-based model checker, where pattern-based verification is used to exclude possibly infinite families of counterexamples. We have shown that pattern-based verification can be applied to programs with unbounded counters.

We have reduced the pattern-based verification problem to CMP, the problem of deciding whether the asynchronous product of a given set of context-free languages and a pattern is nonempty. Revisiting and putting together classical results by Ginsburg and Spanier [Ginsburg 1966] about bounded context-free languages, the characterization of the Parikh images of context-free languages given by Esparza [1997], the encoding of this characterization into existential Presburger arithmetic presented by Verma et al. [2005], and the fact that existential Presburger arithmetic reduces to solving a system of linear Diophantine equations (well known to be in NP [von zur Gathen and Sieveking 1978]), we have shown that CMP is in NP.

We have conducted a multiparameter analysis of CMP on the number of grammars, the maximal size of a grammar, and the size of the pattern. By requiring the value of a parameter to be fixed or not, we get eight cases. We have shown that all *except one* are either trivially polynomial or still NP-hard. Then we conducted a more detailed analysis of the case CMP($\widehat{d}, st, \widehat{pa}$), that is, the case where the number of grammars d and the size of the pattern pa is fixed but not the maximal size st of a grammar. By taking the structure of grammars into account (through parameters lsp and pr), we gave an $st^{O(lsp+\lceil\log(pr+1)\rceil)}$ time algorithm, which is the main technical contribution of the article. The result relies on a novel constructive proof of Parikh's theorem and results about the emptiness problem for bounded-reversal counter machines [Gurari and Ibarra 1981]. It follows that (1) by fixing lsp, we obtain a subexponential time algorithm for CMP($\widehat{d}, st, \widehat{pa}$), and (2) by fixing pr (hence lsp is fixed as well), we obtain a polynomial time algorithm for CMP($\widehat{d}, st, \widehat{pa}$).

Related work. The automatic verification of safety properties for multithreaded programs with possibly recursive procedures has been intensively studied in lpast years. The program is usually modeled as a set of pushdown systems communicating by some means. Several special cases with restricted communication have been proved decidable, including communication through locks satisfying certain conditions, linearly ordered multipushdown systems, and systems with acyclic communication structure (also satisfying some additional conditions) [Kahlon et al. 2005; Kahlon and Gupta 2007; Kahlon 2009a, 2009b; Atig et al. 2008, 2008; Hague 2011]. In all these papers, variables are assumed to have a finite range. More recently, Hague and Lin [2012] have considered multithreaded programs with reversal-bounded counters. A detailed comparison of reversal-bounded and pattern-based verification is left for future work.

Several recent papers study the automatic verification of parametric programs with an arbitrary number of threads [Kaiser et al. 2010; Hague 2011] and with dynamic creation of threads [Bouajjani et al. 2005; Atig et al. 2011], two features that we have not considered in this article. From a complexity point of view, pattern-based verification lies together with context bounding and communication through locks at the lower end of the spectrum. Other approaches require exponential time but do not belong to NP (or this is not known) or are superexponential. The only other case we know of a problem with polynomial complexity in the size of the program is the verification of single-index properties (close to local reachability) in systems communicating through locks [Kahlon et al. 2005].

APPENDIX

A.1. Construction of the Grammar G^f

Let $\widetilde{\boldsymbol{p}} = (a_1w_1)^* \ldots (a_nw_n)^*$ and let $a_iw_i = b_1^{(i)} \ldots b_{j_i}^{(i)}$ for every $1 \leq i \leq n$. Let $G^{\widetilde{\boldsymbol{p}}} = (\mathcal{X}^{\widetilde{\boldsymbol{p}}}, \Sigma, \delta^{\widetilde{\boldsymbol{p}}}, \mathrm{Q}_1^{(1)})$ be the regular grammar where

$$\mathcal{X}^{\widetilde{\boldsymbol{p}}} = \left\{ \mathrm{Q}_r^{(s)} \mid 1 \leq s \leq n \wedge 1 \leq r \leq j_s \right\}$$

$$\begin{aligned}\delta^{\widetilde{\boldsymbol{p}}} &= \left\{\mathrm{Q}_i^{(s)} \to b_i^{(s)}\mathrm{Q}_{i+1}^{(s)} \mid 1 \le s \le n \wedge 1 \le i < j_s\right\}\\ &\cup \left\{\mathrm{Q}_{j_s}^{(s)} \to b_{j_s}^{(s)}\mathrm{Q}_1^{(s')} \mid 1 \le s \le s' \le n\right\}\\ &\cup \left\{\mathrm{Q}_1^{(s)} \to \varepsilon \mid 1 \le s \le n\right\}.\end{aligned}$$

It is routine to check that $\{w \mid \mathrm{Q}_1^{(i)} \Rightarrow^* w \text{ for some } 1 \le i \le n\} = L(\widetilde{\boldsymbol{p}})$.

Given $G^{\widetilde{\boldsymbol{p}}}$ and a grammar $G = (\mathcal{X}, \Sigma', \mathcal{P}, S)$ in program normal form where $\Sigma' \subseteq \Sigma$, our goal is to define a grammar $G^f = (\mathcal{X}^f, \Sigma, \mathcal{P}^f, X_0)$ such that $L(G^f) = L(G) \parallel L(\widetilde{\boldsymbol{p}})$.

—$\mathcal{X}^f = \{X_0\} \cup \{[\mathrm{Q}_r^{(s)}X\mathrm{Q}_y^{(x)}] \mid X \in \mathcal{X} \wedge \mathrm{Q}_r^{(s)} \in \mathcal{X}^{\widetilde{\boldsymbol{p}}} \wedge \mathrm{Q}_y^{(x)} \in \mathcal{X}^{\widetilde{\boldsymbol{p}}} \wedge s \le x\}$.

—$\mathcal{P}^f$ is the set containing for every $1 \le s \le x \le n$ a production $X_0 \to [\mathrm{Q}_1^{(s)}S\mathrm{Q}_1^{(x)}]$ and:

—for every production $X \to \varepsilon \in \mathcal{P}$, $\mathcal{P}^f$ has a production

$$\left[\mathrm{Q}_r^{(s)}X\mathrm{Q}_r^{(s)}\right] \to \varepsilon; \tag{16}$$

—for every production $X \to Y \in \mathcal{P}$, $\mathcal{P}^f$ has a production

$$\left[\mathrm{Q}_r^{(s)}X\mathrm{Q}_v^{(u)}\right] \to \left[\mathrm{Q}_r^{(s)}Y\mathrm{Q}_v^{(u)}\right]; \tag{17}$$

—for every production $X \to \gamma\, Y \in \mathcal{P}$ ($\gamma \in \Sigma'$), $\mathcal{P}^f$ has a production

$$\left[\mathrm{Q}_r^{(s)}X\mathrm{Q}_v^{(u)}\right] \to \gamma \left[\mathrm{Q}_{r'}^{(s')}Y\mathrm{Q}_v^{(u)}\right] \quad \text{if } \mathrm{Q}_r^{(s)} \to \gamma \cdot \mathrm{Q}_{r'}^{(s')} \in \delta^{\widetilde{\boldsymbol{p}}}; \tag{18}$$

—for every production $X \to Z\,Y \in \mathcal{P}$, $\mathcal{P}^f$ has a production

$$\left[\mathrm{Q}_r^{(s)}X\mathrm{Q}_y^{(x)}\right] \to \left[\mathrm{Q}_r^{(s)}Z\mathrm{Q}_v^{(u)}\right]\left[\mathrm{Q}_v^{(u)}Y\mathrm{Q}_y^{(x)}\right]; \tag{19}$$

—for every production $\mathrm{Q}_r^{(s)} \to b_r^{(s)}\,\mathrm{Q}_v^{(u)} \in \delta^{\widetilde{\boldsymbol{p}}}$ such that $b_r^{(s)} \notin \Sigma'$, $\mathcal{P}^f$ has a production

$$\left[\mathrm{Q}_r^{(s)}X\mathrm{Q}_y^{(x)}\right] \to b_r^{(s)}\left[\mathrm{Q}_v^{(u)}X\mathrm{Q}_y^{(x)}\right]. \tag{20}$$

$\mathcal{P}^f$ has no other production.

In what follows, we use the notation $\underset{G}{\Rightarrow}$, which makes explicit the underlying grammar G.

LEMMA A.1. *Let $w \in \Sigma^*$ and w' be the projection of w onto Σ'. We have $[\mathrm{Q}_r^{(s)}X\mathrm{Q}_v^{(u)}] \underset{G^f}{\Rightarrow}^* w$ if and only if $\mathrm{Q}_r^{(s)} \underset{G^{\widetilde{\boldsymbol{p}}}}{\Rightarrow}^* w\,\mathrm{Q}_v^{(u)}$ and $X \underset{G}{\Rightarrow}^* w'$.*

PROOF. The proof for the only-if direction is by induction on the length of the $[\mathrm{Q}_r^{(s)}X\mathrm{Q}_x^{(y)}]$-derivation of $[\mathrm{Q}_r^{(s)}X\mathrm{Q}_x^{(y)}] \Rightarrow^* w$.

i = 1. Then $[\mathrm{Q}_r^{(s)}X\mathrm{Q}_r^{(s)}] \Rightarrow \varepsilon$. The definition of G^f shows that $X \to \varepsilon \in \mathcal{P}$, and so $X \Rightarrow \varepsilon$.

i > 1. We do a case analysis according to the definition of G^f.

—$[\mathrm{Q}_r^{(s)}X\mathrm{Q}_y^{(x)}] \Rightarrow [\mathrm{Q}_r^{(s)}Y\mathrm{Q}_y^{(x)}] \Rightarrow^* w$. It is trivially solved using the induction hypothesis.

—$[\mathrm{Q}_r^{(s)}X\mathrm{Q}_y^{(x)}] \Rightarrow b_r^{(s)}\,[\mathrm{Q}_v^{(u)}Y\mathrm{Q}_y^{(x)}] \Rightarrow^* b_r^{(s)}w'$ with $b_r^{(s)} \in \Sigma'$. The production $\mathrm{Q}_r^{(s)} \to b_r^{(s)}\,\mathrm{Q}_v^{(u)} \in \delta^{\widetilde{\boldsymbol{p}}}$ (which exists by definition of G^f) and induction hypothesis show that $\mathrm{Q}_r^{(s)} \Rightarrow b_r^{(s)}\,\mathrm{Q}_v^{(u)} \Rightarrow^* b_r^{(s)}w'\,\mathrm{Q}_y^{(x)}$. Also, the production $X \to b_r^{(s)}\,Y \in \mathcal{P}$ (which exists by definition of G^f) and induction hypothesis show that $X \Rightarrow b_r^{(s)}\,Y \Rightarrow^* b_r^{(s)}\,w''$, where w'' is the projection of w' onto Σ' and we are done.

—$[\mathrm{Q}_r^{(s)}X\mathrm{Q}_y^{(x)}] \Rightarrow [\mathrm{Q}_r^{(s)}Z\mathrm{Q}_v^{(u)}][\mathrm{Q}_v^{(u)}Y\mathrm{Q}_y^{(x)}] \Rightarrow^* w_1\,[\mathrm{Q}_v^{(u)}Y\mathrm{Q}_y^{(x)}] \Rightarrow^* w_1w_2$. By induction hypothesis, we have $\mathrm{Q}_r^{(s)} \Rightarrow^* w_1\,\mathrm{Q}_v^{(u)}$ and $Z \Rightarrow^* w_1'$, where w_1' is the projection of w_1 onto Σ'. Also,

$\mathrm{Q}_v^{(u)} \Rightarrow^* w_2\,\mathrm{Q}_y^{(x)}$ and $Y \Rightarrow^* w_2$, where w_2' is the projection of w_2 onto Σ'. Hence, we find that $\mathrm{Q}_r^{(s)} \Rightarrow^* w_1 w_2\,\mathrm{Q}_y^{(x)}$ and $X \Rightarrow^* w_1' w_2'$ since $X \to ZY \in \mathcal{P}$, which concludes this case since $w_1' w_2'$ is the projection of $w_1 w_2$ onto Σ'.

—$[\mathrm{Q}_r^{(s)} X \mathrm{Q}_y^{(x)}] \Rightarrow b_r^{(s)}\,[\mathrm{Q}_v^{(u)} X \mathrm{Q}_y^{(x)}] \Rightarrow^* b_r^{(s)} w'$ with $b_r^{(s)} \in \Sigma \setminus \Sigma'$. The production $\mathrm{Q}_r^{(s)} \to b_r^{(s)}\,\mathrm{Q}_v^{(u)} \in \delta^{\tilde{\boldsymbol{p}}}$ and induction hypothesis show that $\mathrm{Q}_r^{(s)} \Rightarrow b_r^{(s)}\,\mathrm{Q}_v^{(u)} \Rightarrow^* b_r^{(s)} w'\,\mathrm{Q}_y^{(x)}$. Also, since $b_r^{(s)} \notin \Sigma'$, the induction hypothesis shows that $X \Rightarrow^* w''$, where w'' is the projection of w' onto Σ' and we are done.

Using a similar induction on the length of $\mathrm{Q}_r^{(s)} \underset{G^{\tilde{\boldsymbol{p}}}}{\Rightarrow^*} w\,\mathrm{Q}_v^{(u)}$, the "if" direction is easily proved. □

A.2. A Finer Analysis

We have defined the size *pa* of a pattern $\boldsymbol{p} = w_1^* \dots w_n^*$ as $\sum_{i=1}^n |w_i|$. We can now zoom in and consider the size as a function of two parameters, the number n of words in the pattern, and *sw* given by $\max_{i=1}^n |w_i|$. Since the reduction of Theorem 6.2 requires a pattern with an arbitrarily large number of words, we study whether CMP stays NP-complete if on top of d and *pr* also n is fixed but not *sw*. We show that CMP remains NP-hard by reduction of the 0-1 Knapsack problem of Definition 5.1.

Consider the reduction from 0-1 Knapsack shown in Theorem 5.2. It does not yield a grammar with a fixed number of procedure variables because of the sets (13), (10), and (14) of productions.

To solve this problem, we first construct a grammar $G^\sharp$ with a fixed number *pr* of procedure variables that can still be used to encode big numbers, albeit by means of a more complicated encoding. Fix a number $b \geq 1$ and an alphabet $\Sigma = \{a_0, a_1, \dots, a_b\} \cup \{z_1, \dots, z_b\}$, and let $w = a_b z_b \cdots a_1 z_1 a_0$. We encode the number $k \leq 2^b$ by the word w^k. The grammar $G^\sharp = (\mathcal{X}^\sharp, \Sigma, \mathcal{P}^\sharp, X)$ has variables $\mathcal{X}^\sharp = \{X\} \cup \{A_1, \dots, A_b\}$ (X is the only procedure variable), and productions $\mathcal{P}^\sharp$ given by the union of the sets (21) to (25):

$$\{X \to A_b\} \tag{21}$$
$$\{X \to z_k A_{k-1} \mid 1 \leq k \leq b\} \tag{22}$$
$$\{A_k \to a_k X A_{k-1} \mid 1 \leq k \leq b\} \tag{23}$$
$$\left\{A_k \to a_j z_j A_k \mid b \geq j \wedge k \geq 0 \wedge j > k\right\} \tag{24}$$
$$\{A_0 \to a_0\}. \tag{25}$$

This grammar can be put in program normal form with a constant increase in *pr*. Thus, we can safely assume $G^\#$ is in program normal form and has a fixed number *pr* of procedure variables.

Consider the pattern $\boldsymbol{p} = w^*$ over the same alphabet Σ as $G^\#$. Because the alphabets of $G^\#$ and $\boldsymbol{p}$ coincide, the asynchronous product ($\|$) behaves exactly like language intersection ($\cap$). Hence, to ease the reader understanding, the rest of the proof uses $\cap$ instead of $\|$.

Our first lemma shows that $\{i \mid A_k \Rightarrow^* w^i\} = \{2^k\}$.

LEMMA A.2. $\{x \mid A_k \Rightarrow^* x\} \cap L(\boldsymbol{p}) = \{w^{2^k}\}$ *for every* $0 \leq k \leq b$.

PROOF. The proof is by induction on k.

$\mathbf{k = 0}$. The only word that can be derived from A_0 and follows $\boldsymbol{p}$ is given by $A_0 (\Rightarrow^{(24)})^* a_b z_b \dots a_1 z_1 A_0 \Rightarrow^{(25)} a_b z_b \dots a_1 z_1 a_0 = w^{2^0}$.

k > 0. We distinguish two cases: $k < b$ and $k = b$. For $k < b$, consider the following partial leftmost A_k-derivation:

$$\begin{array}{rcl} A_k & (\Rightarrow^{(24)})^* & a_b z_b \ldots a_{k+1} z_{k+1} A_k \\ & \Rightarrow^{(23)} & a_b z_b \ldots a_{k+1} z_{k+1} a_k X A_{k-1} \\ & \Rightarrow^{(22)} & a_b z_b \ldots a_{k+1} z_{k+1} a_k z_k A_{k-1} A_{k-1} \\ & (\Rightarrow^{(23)} \Rightarrow^{(22)})^* & a_b z_b \cdots a_1 z_1 A_0 A_0 A_1 \ldots A_{k-1} \\ & \Rightarrow^{(25)} & a_b z_b \cdots a_1 z_1 a_0 A_0 A_1 \ldots A_{k-1} \\ & = & w\, A_0 A_1 \ldots A_{k-1}. \end{array}$$

For the case $k = b$, consider

$$\begin{array}{rcl} A_b & (\Rightarrow^{(23)} \Rightarrow^{(22)})^* & a_b z_b \ldots a_1 z_1 A_0 A_0 A_1 \ldots A_{b-1} \\ & \Rightarrow^{(25)} & w\, A_0 A_1 \ldots A_{b-1}. \end{array}$$

We only need these two partial A_k-derivations, because every leftmost derivation that does not start like one of the two does not generate a word of $L(\boldsymbol{p})$ either. To conclude, we apply the induction hypothesis on $A_0 A_1 \ldots A_{k-1}$ to show that $A_k \Rightarrow^* w\, w^{2^0}\, w^{2^1} \cdots w^{2^{k-1}}$, and hence that $A_k \Rightarrow^* w^{2^k}$ since $1 + \sum_{i=0}^{k-1} 2^i = 2^k$. □

Using this lemma, we can already obtain a first reduction from the 0–1 Knapsack problem to CMP in polynomial time. If in the reduction of Theorem 5.2 the set (13) is replaced by the set $\mathcal{P}^\sharp$, we get $L(G) \cap L(G_W) \cap L(\boldsymbol{p}) \neq \varnothing$ if and only if there exists $S \subseteq \{o_1, \ldots, o_m\}$ such that S's weight is W. However, this reduction is not yet adequate because the variables $\{A_1, \ldots, A_b\}$ are procedure variables (see the sets (10) and (14) of productions). To fix this problem, we need a second lemma:

LEMMA A.3.

(*1*) $L(G^\sharp) \cap L(\boldsymbol{p}) = \{w^{2^b}\}$.
(*2*) $\big(\{a_b z_b \ldots a_{k+1}\}\, L(G^\sharp)\big) \cap L(\boldsymbol{p}) = \{w^{2^k}\}$ *for all* $1 \leq k \leq b - 1$.

PROOF. (1) Any derivation of $G^\sharp$ that generates a word of $L(\boldsymbol{p})$ must use the production (21) first, so that $X \Rightarrow^{(21)} A_b$. Applying Lemma A.2, we get $L(G^\sharp) \cap L(\boldsymbol{p}) = \{x \mid A_b \Rightarrow^* x\} \cap L(\boldsymbol{p}) = \{w^{2^b}\}$.

(2) Any derivation of $G^\sharp$ generating a word u such that $a_b z_b \ldots a_{k+1} u$ belongs to $L(\boldsymbol{p})$ must start with $X \Rightarrow^{(22)} z_{k+1}\, A_k$. As shown in the proof of Lemma A.2, the A_k-derivation must continue with $X \Rightarrow^{(22)} z_{k+1}\, A_k \Rightarrow^* w\, A_0 A_1 \cdots A_{k-1}$, and so finally lead to w^{2^k}. □

With the help of this lemma, we can now proceed as follows. Recall that we have already replaced set (13) in the reduction of Theorem 5.2 by $\mathcal{P}^\sharp$. Now we replace the set (10) by

$$\left\{ S_i^{(k)} \to a_b z_b \cdots a_{k+1}\, X S_i^{(k-1)} \mid 1 \leq i \leq m \wedge 1 \leq k \leq b \wedge \text{bit } k \text{ of } W \text{ is } 1 \right\}$$

and the set (14) by

$$\left\{ W^{(k)} \to a_b z_b \cdots a_{k+1}\, X W^{(k-1)} \mid 1 \leq k \leq b \wedge \text{bit } k \text{ of } W \text{ is } 1 \right\}.$$

This gives two grammars $G_1^{\bowtie}$ and $G_2^{\bowtie}$ with S_1 and $W^{(b)}$ as axioms, respectively. We have:

THEOREM A.4. *The following problem is NP-hard:*
Instance: *Two context-free grammars G_1, G_2 in program normal form over alphabet Σ, each of them with one procedure variable, and a pattern $\boldsymbol{p} = w^*$ consisting of a single word $w \in \Sigma^*$.*

Question: *Is* $L(G_1) \parallel L(G_2) \parallel L(\boldsymbol{p}) \neq \varnothing$ *?*

Proof. The proof is by reduction to 0-1 Knapsack. We construct $G_1^{\bowtie}$, $G_2^{\bowtie}$, and $\boldsymbol{p}$ as earlier. The proof of correctness for the reduction essentially follows the one of Theorem 5.2, where the result of Lemma A.3 is used when needed (because the alphabets of $G_1^{\bowtie}$, $G_2^{\bowtie}$, and $\boldsymbol{p}$ coincide, we can interchange $\cap$ and $\parallel$). Thus, we obtain that $L(G_1^{\bowtie}) \parallel L(G_2^{\bowtie}) \parallel L(\boldsymbol{p}) \neq \varnothing$ if and only if a subset of $\{o_1, \ldots, o_m\}$ has weight W.

It is routine to check the following: given a 0-1 Knapsack instance (1), $G_i^{\bowtie}$ is computable in polynomial time, (2) X is the only procedure variable of $G_i^{\bowtie}$ where $i \in \{1, 2\}$, and (3) $\boldsymbol{p} = w^*$ is computable in polynomial time. Note that $G_i^{\bowtie}$ is not in program normal form but can easily be brought into it by adding new variables and productions. The transformation does not add any procedure variable. □

ACKNOWLEDGMENTS

Many thanks to Mikołaj Bojańczyk for providing the proof of Theorem 4.1.

REFERENCES

Tony Andrews, Shaz Qadeer, Sriram K. Rajamani, Jakob Rehof, and Yichen Zie. 2004. *Zing: A Model Checker for Concurrent Software*. Technical Report MSR-TR-2004-10. Microsoft Research.

Mohamed Faouzi Atig, Benedikt Bollig, and Peter Habermehl. 2008. Emptiness of multi-pushdown automata is 2EXPTIME-complete. In *Proc. 12th Int. Conf. on Developments in Language Theory.* Springer, 121–133.

Mohamed Faouzi Atig, Ahmed Bouajjani, and Shaz Qadeer. 2011. Context-Bounded analysis for concurrent programs with dynamic creation of threads. *Logical Methods in Computer Science (LMCS)* 7, 4 (2011), 1–48.

Mohamed Faouzi Atig, Ahmed Bouajjani, and Tayssir Touili. 2008. On the reachability analysis of acyclic networks of pushdown systems. In *Proc. 19th Int. Conf. on Concurrency Theory.* Springer, 356–371.

Ahmed Bouajjani, Markus Müller-Olm, and Taissir Touili. 2005. Regular symbolic analysis of dynamic networks of pushdown systems. In *Proc. 16th Int. Conf. on Concurrency Theory.* Springer, 473–487.

Javier Esparza. 1997. Petri nets, commutative context-free grammars, and basic parallel processes. *Fundamenta Informaticae* 31 (1997), 13–26.

Javier Esparza, Pierre Ganty, Stefan Kiefer, and Michael Luttenberger. 2011. Parikh's theorem: A simple and direct automaton construction. *Inform. Process. Lett.* 111 (2011), 614–619.

Pierre Ganty, Rupak Majumdar, and Benjamin Monmege. 2012. Bounded underapproximations. *Formal Methods Syst. Des.* 40, 2 (2012), 206–231.

Michael R. Garey and David S. Johnson. 1979. *Computers and Intractability, A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York.

Seymour Ginsburg. 1966. *The Mathematical Theory of Context-Free Languages*. McGraw-Hill, New York, NY.

Eitan M. Gurari and Oscar H. Ibarra. 1981. The complexity of decision problems for finite-turn multicounter machines. *J. Comput. System Sci.* 22 (1981), 220–229.

Matthew Hague. 2011. Parameterised pushdown systems with non-atomic writes. In *Proc. 31st IARCS Annual Conf. on Foundation of Software Technology and Theoretical Computer Science (Leibniz International Proceedings in Informatics (LIPIcs)).*Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 457–468.

Matthew Hague and Anthony Widjaja Lin. 2012. Synchronisation- and reversal-bounded analysis of multithreaded programs with counters. In *Proc. 24th Int. Conf. on Computer Aided Verification.* Springer, 260–276.

John E. Hopcroft and Jeffrey D. Ullman. 1979. *Introduction to Automata Theory, Languages and Computation* (1st ed.). Addison-Wesley.

Vineet Kahlon. 2009a. Boundedness vs. unboundedness of lock chains: Characterizing decidability of pairwise CFL-reachability for threads communicating via locks. In *Proc. 24th Annual IEEE Symp. on Logic in Computer Science*. IEEE Computer Society, 27–36.

Vineet Kahlon. 2009b. Tractable Dataflow Analysis for Concurrent Programs via Bounded Languages. (July 2009). Patent WO/2009/094439.

Vineet Kahlon and Aarti Gupta. 2007. On the analysis of interacting pushdown systems. In *Proc. 30th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*. ACM, 303–314.

Vineet Kahlon, Franjo Ivancic, and Aarti Gupta. 2005. Reasoning about threads communicating via locks. In *Proc. 17th Int. Conf. on Computer Aided Verification.* Springer, 505–518.

Alexander Kaiser, Daniel Kroening, and Thomas Wahl. 2010. Dynamic cutoff detection in parameterized concurrent programs. In *Proc. 20th Int. Conf. on Computer Aided Verification.* Springer, 645–659.

Akash Lal and Thomas Reps. 2009. Reducing concurrent analysis under a context bound to sequential analysis. *Formal Methods Syst. Des.*, 1 (2009), 73–97.

Akash Lal, Tayssir Touili, Nicholas Kidd, and Thomas W. Reps. 2008. Interprocedural analysis of concurrent programs under a context bound. In *Proc. 14th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems.* Springer, 282–298.

Zhenyue Long, Georgel Calin, Rupak Majumdar, and Roland Meyer. 2012. Language-theoretic abstraction refinement. In *Proc. 15th Int. Conf. Fundamental Approaches to Software Engineering.* Springer, 362–376.

Madanlal Musuvathi and Shaz Qadeer. 2007. Iterative context bounding for systematic testing of multithreaded programs. In *Proc. 28th ACM-SIGPLAN Conf. on Programming Language Design and Implementation*. ACM, 446–455.

Rohit J. Parikh. 1966. On context-free languages. *J. ACM* 13, 4 (1966), 570–581.

Shaz Qadeer and Jakob Rehof. 2005. Context-Bounded model checking of concurrent software. In *Proc. 11th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems.*Springer, 93–107.

Shaz Qadeer and Dinghao Wu. 2004. KISS: keep it simple and sequential. In *Proc. 25th ACM-SIGPLAN Conf. on Programming Language Design and Implementation*. ACM, 14–24.

Ganesan Ramalingam. 2000. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM TOPLAS* 22, 2 (2000), 416–430.

Dejvuth Suwimonteerabuth, Javier Esparza, and Stefan Schwoon. 2008. Symbolic context-bounded analysis of multithreaded Java programs. In *Proc. of 15th Int. Model Checking Software Workshop.* Springer, 270–287.

Salvatore La Torre, Gennaro Parlato, and Parthasarathy Madhusudan. 2009. Reducing context-bounded concurrent reachability to sequential reachability. In *Proc. 21st Int. Conf. on Computer Aided Verification.* Springer, 477–492.

Kumar N. Verma, Helmut Seidl, and Thomas Schwentick. 2005. On the complexity of equational horn clauses. In *Proc. 20th Int. Conf. on Automated Deduction.* Springer, 337–352.

Joachim von zur Gathen and Malte Sieveking. 1978. A bound on solutions of linear integer equalities and inequalities. *Proc. Amer. Math. Soc.* 72 (1978), 155–158.

Received March 2012; revised December 2013; accepted January 2014