Complexity of Pattern-based Verification for Multithreaded Programs *

Javier Esparza
Fakultät für Informatik, Technische Universität München, Germany.
esparza@model.in.tum.de

Pierre Ganty
The IMDEA Software Institute, Madrid, Spain.
pierre.ganty@imdea.org

Abstract

Pattern-based verification checks the correctness of the 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 synchronization operations between threads. We study the complexity of pattern-based verification for *abstracted* multithreaded programs in which, as usual in program analysis, conditions have been replaced by nondeterminism (the technique works also for boolean programs). While unrestricted verification is undecidable for abstracted multithreaded programs with recursive procedures and PSPACE-complete for abstracted multithreaded while-programs, we show that pattern-based verification is NP-complete for both classes. We then conduct a multiparameter analysis in which we study the complexity in the number of threads, the number of procedures per thread, the size of the procedures, and the size of the pattern. We first show that no algorithm for pattern-based verification can be polynomial in the number of threads, procedures per thread, or the size of the pattern (unless P=NP). Then, using recent results about Parikh images of regular languages and semilinear sets, we present an algorithm exponential in the number of threads, procedures per thread, and size of the pattern, but polynomial in the size of the procedures.

Categories and Subject Descriptors: D.2.4 [Software Engineering]: Software/Program Verification.
General Terms: Verification, Languages, Algorithms, Reliability.
Keywords: concurrent programming, safety, context-free languages.

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 untractable 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 that replaces all conditions in alternative constructs and loops by nondeterminism.

Context-bounding, proposed by Qadeer and Rehof in [25], is the most successful proposal to date for overcoming untractability. It restricts the problem further by exploring only those computation 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 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 in k [20, 21, 25] . Context-bounding has been implemented in several model checkers, like CHESS, SPIN, SLAM, jMoped, and others [1, 5, 20, 28, 30], 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. In particular, it restricts the number of communications 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 value of the variable immediately before the context switch amounts to a communication. So in a computation with k context switches threads communicate at most k times. In this paper we study a more flexible technique, introduced by Kahlon in [15], that applies the theory of *bounded languages* developed in the mid-sixties by Ginsburg and Spanier [11] to the verification problem. Kahlon uses the theory to prove decidability of safety analysis for multithreaded programs whose executions conform to a *pattern*, a regular expression of the form $w_1^* \dots w_n^*$ over the alphabet of program instructions. Observe that the executions of such a program can be arbitrarily long. An equivalent, but in our opinion more fruitful point to view is to consider a pattern as a class of executions specified by the verifier. The executions of the program may conform to the pattern or not, but we can automatically verify whether those executions conforming to the pattern satisfy the property. In other words, the programmer specifies by means of a pattern those executions she/he is interested in. We call this point of view *pattern-based verification*.

The claim of [15] is that pattern-based verification provides a good compromise between expressivity and complexity. The ex-

* This research was sponsored by the Comunidad de Madrid's Program PROMETIDOS-CM (S2009TIC-1465), by the PEOPLE-COFUND's program AMAROUT (PCOFUND-2008-229599), and by the Spanish Ministry of Science and Innovation (TIN2010-20639).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
POPL'11, January 26–28, 2011, Austin, Texas, USA.
Copyright © 2011 ACM 978-1-4503-0490-0/11/01...$10.00

pressivity point has been considered in some detail in [9], where it is shown there that the pattern-based approach is strictly more expressive than context-bounding.[1] In this paper we study the computational complexity of pattern-based verification, thus completing the theoretical analysis. In a nutshell, we show that pattern-based verification, like context-bounding, is NP-complete, and we identify an interesting (and in a certain sense unique) polynomial case.

For the complexity analysis we reduce the reachability problem for multithreaded programs to a language theory problem called *non-Disjointness Modulo a Pattern*, or nDMP for short: checking non-emptiness of the intersection of a given set of context-free grammars *and* a given pattern. By putting together classical results by Ginsburg and Spanier [11] and more recent results by Verma, Seidl, and Schwentick [31] we first show that, like context-bounded reachability, nDMP is NP-complete. Interestingly, the algorithm we derive from the easiness proof relies on satisfiability checking of a Presburger formula, which contrasts with the fixed point evaluation used in context-bounding. In the second and main part of the paper, we conduct a multiparameter analysis of nDMP. The size of an instance of nDMP is a function of four parameters: the number of threads, the maximal number of procedures per thread, the maximal size of a procedure, and the size of the pattern. For every subset of parameters we determine the complexity of nDMP when the parameters in the subset (and no others) have a fixed value. While this gives 16 possible cases, the results can be easily summarized: apart from some trivial cases, in which the problem can be solved in constant time (for instance, when all four parameters have fixed values), the problem remains NP-complete for all subsets *except one*: the case in which the number of threads, procedures per thread and the size of the pattern are fixed, but the size of the procedures is not. We prove that this case is polynomial. The proof uses several recent results about Parikh images of regular languages and complexity of semilinear sets [19, 29].

The paper is organized as follows. Section 2 contains preliminaries. Section 3 presents our program and formal models, an analysis of the context-bounding technique, and the reduction of the pattern-based verification problem to nDMP. Section 4 shows that nDMP is NP-complete. Section 5 contains our multiparameter analysis of nDMP. The NP-hard cases are covered by means of reductions from different NP-complete problems. Our main result, the polynomial case mentioned above, is contained in Section 5.3. Finally, Section 6 contains conclusions and discusses related work.

2. Preliminaries

An *alphabet* Σ is a finite non-empty set of symbols. We assume the reader is familiar with the basics of language theory, including regular and context-free languages (see e.g. [13]).

Context-free Languages. A *context-free grammar* is a tuple $G = (\mathcal{X}, \Sigma, \mathcal{P}, S)$ where $\mathcal{X}$ is a finite non-empty set of *variables*, 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 $\Rightarrow$. 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 $\mathcal{X} \times \Sigma^*(\mathcal{X} \cup \{\varepsilon\})$. A language L is *regular* if $L = L(G)$ for some regular grammar G.

[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.

We sometimes use $L_X(G)$ with $X \in \mathcal{X}$ to denote the language $\{w \in \Sigma^* \mid X \Rightarrow^* w\}$.

Multisets. A *multiset* $\mathbf{m}\colon \Sigma \mapsto \mathbb{N}$ maps each symbol of Σ to a natural number. $\mathbb{M}[\Sigma]$ denotes the set of all multisets over Σ. We sometimes use the following notation: $[\![q_1, q_1, q_3]\!]$ denotes the multiset $\mathbf{m}$ such that $\mathbf{m}(q_1) = 2$, $\mathbf{m}(q_3) = 1$ and $\mathbf{m}(x) = 0$ for all $x \in \Sigma \setminus \{q_1, q_3\}$. 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.

3. Model and decision problem

3.1 Program model

We model a sequential program by a *system of flowgraphs*, a tuple of flowgraphs containing one flowgraph for each procedure. *Nodes* of a flowgraph correspond to *control points* of the program, and edges to *sequential statements*. A sequential statement is either a *condition* (a boolean combination of expressions $x \leq e$), an *assignment* $x := e$, or a *procedure call*. Each flowgraph has a unique node without incoming edges, the *initial node*, and a unique node without outgoing edges, the *final node*, different from the initial node. All nodes are reachable from the initial node and co-reachable from the final node.

A multithreaded program is modeled by a tuple of systems of flowgraphs, one for each program thread. Each system of flowgraphs uses a set of *channels* to send or receive messages; abusing language, we call a system of flowgraphs together with the set of channels it uses a *thread*. We denote by Ch_i the set of channels used by the i-th thread, and the set of all channels by Ch. The edges of the flowgraphs are labeled by sequential statements, by *send statements* $a!x$, indicating that the thread is willing to send the value of x through channel $a \in Ch$, or by *receive statements* $a?y$, indicating that the thread is willing to receive a value through channel a and assign it to variable y. We assume that each channel is *owned* by a thread: the owner of channel a can only contain send statements $a!x$, and all other threads can only contain receive statements $a?y$. Channels work as in CSP: they have capacity 0, i.e., a message is exchanged through channel a only if its owner executes a send statement, and all other threads having a in their sets of channels simultaneously execute a matching receive statement. So we allow multiparty synchronization. Figure 1 shows a model of a program with three threads. Each of the threads, contains only one flowgraph, with channels $\{a, b, c\}$, $\{a, b\}$, and $\{b, c\}$, respectively. The first thread owns channels a and c, the second thread owns channel b and the third owns no channel.

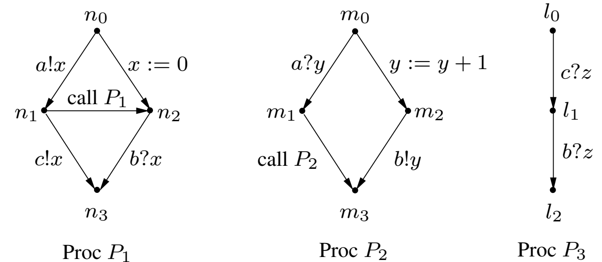

Figure 1. A model of a program with three threads.

During a program execution threads exchange values through channels. A *trace* of the program is the sequence of channels used

along some *full* execution. For instance, $aacb$ is a trace of the program of Figure 1 corresponding to (among others) the execution $(a!x, a?y)$ $call P_1$ $call P_2$ $(a!x, a?y)$ $(c!x, c?z)$ $call P_2$ $y := y + 1$ $(b?x, b!y, b?z)$.

Using standard techniques, verification of safety properties can be reduced to the reachability of some program point, which can be reduced to nonemptiness of the set of traces of a modified program (notice that the set of traces is nonempty iff the program can terminate, i.e, all threads can simultaneously reach their final node). Since this problem is undecidable even for single-thread while-programs, further restrictions are unavoidable. The classical program analysis abstraction consists of replacing all condition statements by non deterministic choice, which amounts to ignoring data, since data do not longer influence control-flow. We call the result an *abstracted program*. Unfortunately, trace emptiness is still undecidable for abstracted multithreaded programs [26], and PSPACE-complete for multithreaded while-programs. For this reason we restrict the problem further. However, before doing so we define a formal model for abstracted multithreaded programs.

Remark. Context-bounding is formulated in [24, 25] for *boolean programs*, programs in which all variables are boolean. In boolean programs data influences control, and so one could think that there is a deep conceptual difference with the program analysis abstraction. However, this is not the case. Since the number of valuations of the variables of a boolean program is finite, the program can be easily transformed into a dataless program whose program points are pairs consisting of a program point of the boolean program and a valuation of the variables. This is in fact how the context-bounding technique proceeds, since it models a boolean program as a pushdown system, a dataless formal model. In our presentation we stick to the program analysis abstraction for convenience and clarity, but the technique can be equally well applied to predicate abstractions and boolean programs.

3.2 Formal model

Let P be an abstracted multithreaded program with threads $t_1, \dots, t_n$ communicating over a set Ch of channels. We assign to P a tuple $G_1, \dots, G_n$ of context-free grammars over the alphabet Ch, such that w is a trace of P iff w belongs to $\bigcap_{i=1}^{n} L(G_i)$. We proceed in two steps. First we assign to P grammars $G'_1, \dots, G'_n$ with alphabets $Ch_1, \dots, Ch_n$ such that w is a trace of P iff for every $1 \le i \le n$ the projection of w onto Ch_i belongs to $L(G'_i)$. In a second step we transform these grammars into the final grammars $G_1, \dots, G_n$.

The grammar G'_i over the alphabet Ch_i generates the interprocedurally valid traces of t_i. These are the traces that t_i can generate in an environment always ready to match its send and receive statements. G'_i has a variable for each node of t_i, a production for each edge, and a further production for the final node of t_i. The production corresponding to an edge leading from node X to node Y and labelled by ℓ is defined as follows:

- if ℓ is a condition or an assignment, the production is $X \to Y$;
- if $\ell = call\ P$, the production is $X \to P_0 Y$, where P_0 is the initial node of procedure P;
- if $\ell = a!x$ or $\ell = a?y$, the production is $X \to aY$;
- the production for the final node Z of t is $Z \to \varepsilon$.

The three grammars for the program of Figure 1 have variables $\{n_0, \dots, n_3\}$, $\{m_0, \dots, m_3\}$, $\{l_0, l_1, l_2\}$, terminals $\{a, b, c\}$, $\{a, b\}$, $\{b, c\}$, and productions:

$$
\begin{array}{lll}
n_0 \to a n_1 \mid n_2 & m_0 \to a m_1 \mid m_2 & l_0 \to c l_1 \\
n_1 \to c n_3 \mid n_0 n_2 & m_1 \to m_0 m_3 & l_1 \to b l_2 \\
n_2 \to b n_3 & m_2 \to b m_3 & l_2 \to \varepsilon \\
n_3 \to \varepsilon & m_3 \to \varepsilon &
\end{array}
$$

Observe that the number of "proper" procedures (procedures that can be called, unlike P_3 in our example) is equal to the number of variables Z for which there is a production of the form $X \to ZY$. We call them *procedure variables*. In our example, those variables are n_0, m_0. It is easy to see that the traces of P under the program analysis abstraction are the words $w \in Ch^*$ satisfying the following property: for every thread t_i, the projection of w onto Ch_i is a word of $L(G'_i)$. This completes the first step.

For the second step, we slightly modify each G'_i: we set its alphabet to Ch, and add new productions. For each variable X of G'_i and for each channel a that *does not appear in* G'_i, we add a new production $X \to aX$. The grammar so obtained is denoted by G_i. In our example, we add productions $m_j \to c m_j$ to the second grammar for $j \in \{0, \dots, 3\}$, and productions $l_j \to a l_j$ to the third grammar for $j \in \{0, \dots, 2\}$. Observe that a grammar G_i is in a particular *program normal form*: all productions are of the form $X \to a\alpha$ or $X \to \beta\gamma$, where α is a variable and β, γ is either a variable or ε. All grammars now have Ch as alphabet. Since every channel is owned by some thread, it is easy to see that the set of traces of an abstracted program with threads $t_1, \dots, t_n$ is equal to $\bigcap_{i=1}^{n} L(G_i)$.

Since reachability of a program point can be easily reduced to checking nonemptiness of the set of traces of a modified program, our formal model reduces the reachability problem for abstracted programs to the nonemptiness problem for the intersection of context-free languages. Since this problem is undecidable, this does not immediately provide any algorithmic advantage. For this reason we restrict the problem and introduce pattern-based verification.

3.3 Pattern-based verification

Kahlon [15] has recently proposed to only explore the traces of a multithreaded program having a certain *shape*. Inspired by the work of Ginsburg and Spanier [11], he suggests to only explore traces conforming to what we call in this paper *communication patterns* (or just *patterns* for short). Patterns are regular expressions of the form $w_1^* w_2^* \dots w_n^*$, where $w_i \in Ch^* \setminus \{\varepsilon\}$[2]. We study the problem of deciding, given an abstract multithreaded program P and a pattern $\boldsymbol{p}$, whether some word of $L(\boldsymbol{p})$ is a trace of P. Given the formal model given above, this verification problem reduces to the following language-theoretic problem:

DEFINITION 1. ***Non Disjointness Modulo a Pattern (nDMP)***
Instance: *Context-free grammars $G_1, \dots, G_g$ in program normal form over an alphabet Σ, and a pattern $\boldsymbol{p}$ over Σ.*
Question: *Is $\bigcap_{i=1}^{g} L(G_i) \cap L(\boldsymbol{p}) \neq \varnothing$?*

3.4 Context bounding as pattern-based verification

Recall that in context bounding, instead of asking whether a given multithreaded program has a trace, we ask if it has a trace with at most k context switches. Before studying the complexity of nDMP, we sketch an argument showing that context bounding can be seen as a special case of pattern-based verification. We do not formalize the reduction, which would be very technical and tedious, but describe it in enough detail in order to (we hope) convince the reader.

---

[2] The languages of patterns are called *bounded languages* in the literature.

Consider a multithreaded boolean program P communicating through shared variables. Without loss of generality (see [24, 25] for details) we assume that P has one single shared variable g which can take v different values. We first show how to simulate P by a multithreaded program P' whose threads communicate through message passing. Let $t_1, \ldots, t_n$ be the threads of P. The program P' has threads $t'_1, \ldots, t'_n$. Each thread t'_i has a variable g_i that acts as a local "copy" of g.[3] At any given moment in time, every thread of P' is either *active* or *passive*. Loosely speaking, when t'_i is active it simulates the thread t_i; when it goes passive, it suspends the simulation, until its next active phase. More precisely, from its passive state a thread t'_i can either send a signal to all other threads through a channel a_i, by which it becomes active, or receive a signal through a channel a_j for some $j \neq i$, by which it remains passive. After one of the two happens, t'_i behaves as follows: (1) If t'_i has become active, then it resumes its simulation of t_i. Thread t'_i simulates t_i using the most recent value of g which is available from g_i. At any point t'_i may nondeterministically decide to suspend the simulation. In this case, t'_i communicates to all other processes the current value of g (available through g_i), say u, by sending a signal through a channel $b_{i,u}$. After sending this signal, t'_i becomes passive. (2) If t'_i has remained passive, then it waits for a signal through some channel $b_{j,u}$, where $j \neq i$, and when the signal arrives it updates the value of g_i to u.

Observe that a context of P is simulated by an "activity cycle" of P', i.e., a segment of the computation of P' starting at the moment a thread becomes active, and ending when it switches to the passive state. Since the trace of an activity cycle has length 2 (during a cycle the active thread sends exactly two signals), the traces of P' simulating computations of P with at most k contexts have length at most $2k$.

The problem of deciding if P has a full computation with at most k context switches can now be reduced to an instance of nDMP. The grammars of the instance are the result of applying the translation of Section 3.2 to P'. For the pattern, let $W = \{w_1, \ldots, w_{nv}\}$ be the set of all sequences of length 2 of the form $a_i b_{i,u}$ (there are nv of them), and let $\boldsymbol{p} = (w_1^* \ldots w_{nv}^*)^k$. Clearly, $L(\boldsymbol{p})$ contains (among others) all sequences obtained by concatenating at most k words of W. So all full computations of P with at most k contexts are simulated by computations of P' whose traces belong to $L(\boldsymbol{p})$. Therefore, if P has a full computation with at most k contexts, then the intersection of the languages of the grammars obtained from P' and $\boldsymbol{p}$ is nonempty. (The converse does not hold, but this only shows that the instance of nDMP explores *more* computations of P that context bounding with k context switches.)

4. NP-completeness of nDMP

The decidability of nDMP was proved in [11]. We show it is NP-complete. But we first define the size of an instance of nDMP, since this requires some care. The size $|w|$ of a word w is its length $|w|$. The size of a pattern $\boldsymbol{p} = w_1^* \ldots w_n^*$ is $|\boldsymbol{p}| = \sum_{i=1}^n |w_i|$. Defining the size of a grammar requires a bit of care. The seemingly natural choice would be to define the size of a grammar $G = (\mathcal{X}, \Sigma, \mathcal{P}, S)$ as $|\mathcal{X}| + |\mathcal{P}|$. However, recall that the grammar G_t for a thread t is constructed in two steps: in a first step, a grammar G'_t is constructed that matches the behaviour of the thread is defined; in a second step, loop productions of the form $X \to aX$ are added for every variable X and every channel a that does not appear in t. In pathological cases the number of these productions could be much larger than the number of "true" productions, artificially increasing the size of the grammar. For this reason, when a grammar has productions $X \to aX$ for every variable X and some terminal a, we define that all those productions count as one single production for determining the size. We denote the size of a grammar G, so defined, by $|G|$.

4.1 nDMP is NP-hard

We show[4] that nDMP 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.

THEOREM 1. *The following problem is NP-hard:*
Instance: *Regular grammars $G_1, \ldots, G_g$ in program normal form.*
Question: *Is $\bigcap_{i=1}^g L(G_i) \cap L(\boldsymbol{p}) \neq \varnothing$ for the pattern $\boldsymbol{p} = a^*$?*

PROOF: The proof is by reduction from 3-CNF-SAT. Let Ψ be a propositional formula with n 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 $\bigcap_{i=1}^m L(G_i) \neq \varnothing$ iff Ψ is satisfiable.

We need some preliminaries. Assign to each variable v a prime number n_v, and assign to each clause c the number n_c obtained by multiplying the primes of the three variables occurring in c. (This requires to construct n primes in time $p(n)$ for some polynomial p. It is well-known that the i-th prime number p_i satisfies $p_i < i \ln i + i \ln \ln i$, and so one can compute n primes by applying a primality test to each number from 1 to $n \ln n + n \ln \ln n$. Notice that the primality test can take exponential time, because the size of the number k is $\mathcal{O}(\ln k)$.) 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, or 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 $k \equiv 0 \mod 5$, i.e., 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. It is easy to see that ϕ satisfies c iff n_ϕ is a c-witness.

Now, for each clause 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 $0 \leq k \leq n_c - 1$, where $\oplus_c$ is addition modulo n_c, and a further production $k \to \epsilon$ for each c-witness $k \leq n_c - 1$. We have $L(G_c) = \{a^k \mid k \text{ is a } c\text{-witness}\}$, and so an assignment ϕ satisfies c iff $a^{n_\phi} \in L(G_c)$. So $\bigcap_{i=1}^m L(G_{c_i}) \neq \varnothing$ iff Ψ is satisfiable, and so $\bigcap_{i=1}^m L(G_{c_i}) \cap L(\boldsymbol{p}) \neq \varnothing$ holds for $\boldsymbol{p} = a^*$ iff Ψ is satisfiable. □

4.2 nDMP is in NP

We show that nDMP is in NP. The direct approach would be to show that if $\bigcap_{i=1}^g L(G_i) \cap L(\boldsymbol{p}) \neq \varnothing$, then there is a witness $w \in \bigcap_{i=1}^g L(G_i) \cap L(\boldsymbol{p})$ 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 Lemma 2). So we proceed differently, in two steps: first we polynomially reduce nDMP to a problem about Parikh images of context-free grammars, and then we show that this problem is in NP.

The *Parikh image* of a word $w \in \Sigma^*$ is the multiset $\Pi(w) : \Sigma \mapsto \mathbb{N}$ that assigns to each $a \in \Sigma$ the number of occurrences of a in w. The Parikh image of a language L, denoted by $\Pi(L)$, is the set of Parikh images of its words. We consider the following problem:

DEFINITION 2. ***Non Disjointness of Parikh Images (nDPK)***

[3] Since our multithreaded programs are dataless, the local variable g_i is actually encoded into the nodes of the program as explained in a previous remark.

[4] The proof is due to Mikołaj Bojańczyk.

Instance: *Context-free grammars $G_1, \ldots, G_g$ in program normal form.*
Question: *Is $\bigcap_{i=1}^{g} \Pi(L(G_i)) \neq \varnothing$?*

The reduction from nDMP to nDPK relies on a classical result by Ginsburg and Spanier [11]: Given context-free languages $L_1, \ldots, L_g$ and a pattern $\boldsymbol{p} = w_1^* \ldots w_n^*$ over an alphabet Σ, there exist context-free languages $L'_1, \ldots, L'_g$ such that

$$\bigcap_{i=1}^{g} L_i \cap L(\boldsymbol{p}) \neq \varnothing \quad \text{iff} \quad \bigcap_{i=1}^{g} \Pi(L'_i) \neq \varnothing \qquad (1)$$

The proof can be easily sketched: take a new alphabet $\widetilde{\Sigma} = \{a_1, \ldots, a_n\}$, and consider the homomorphism $h\colon \widetilde{\Sigma} \to \Sigma^*$ given by $h(a_i) = w_i$ for every $1 \leq i \leq n$. Since context-free languages are closed under intersection with regular languages and under inverse homomorphism, the language $L'_i = h^{-1}(L_i \cap L(\boldsymbol{p})) \cap L(a_1^* \ldots a_n^*)$ is context-free and satisfies property (1). Moreover, using the constructions underlying these closure properties we can easily construct from a grammar G_i for L_i a grammar G'_i for L'_i in polynomial time.

However, for the complexity analysis in Sect. 5.3 we also need to establish a relation between the number of procedure variables of G_i and G'_i. For this reason we provide our own direct construction.

4.2.1 A polynomial time reduction from nDMP to nDPK

The following lemma contains the main properties of our construction.

LEMMA 1. *Given $\boldsymbol{p} = w_1^* \ldots w_n^*$ over Σ, an alphabet $\widetilde{\Sigma} = \{a_1, \ldots, a_n\}$, a homomorphism $h\colon \widetilde{\Sigma} \to \Sigma^*$, and a grammar G in program normal form, we can compute in polynomial time a grammar G^f over $\widetilde{\Sigma}$ in program normal such that:*

- *$L(G^f) = h^{-1}(L(G) \cap L(\boldsymbol{p})) \cap L(a_1^* \ldots a_n^*)$;*
- *If pr is the number of procedure variables in G, then G^f has $\mathcal{O}(pa^2 \cdot pr)$ procedure variables where pa is the size of $\boldsymbol{p}$.*

PROOF: (Sketch.) Let $G^{\boldsymbol{p}}$ be a regular grammar with $\mathcal{O}(pa)$ variables such that $L(G^{\boldsymbol{p}}) = L(\boldsymbol{p})$ (this grammar clearly exists). The variables of G^f are triples $[q_1 X q_2]$, where X is a variable of G and q_1, q_2 are variables of $G^{\boldsymbol{p}}$ such that q_2 is reachable from q_1. The productions are chosen to satisfy that $[q_1 X q_2] \Rightarrow^* w$ holds in G^f iff there exists $u \in \Sigma^*$ such that (1) $w \in h^{-1}(u) \cap L(a_1^* \ldots a_n^*)$, (2) $X \Rightarrow^* u$ holds in G, and (3) $q_1 \Rightarrow^* u \cdot q_2$ in $G^{\boldsymbol{p}}$. This is achieved in two steps. First, a grammar $G^{\bowtie}$ is constructed that satisfies $[q_1 X q_2] \Rightarrow^* u$ iff conditions (2) and (3) above hold. The construction is similar to the triple construction used to transform a pushdown automaton into an equivalent context-free grammar. In the second step we adjust the terminals in the productions of $G^{\bowtie}$: the productions used to generate the letters of the words $w_1, \ldots, w_n$ are modified so that they generate no terminal at all, with the exception of those productions generating the last letter of one the words $w_1, \ldots, w_n$, say, the word w_i: these that are modified so that they generate the letter $a_i = h^{-1}(w_i)$ instead. For the number of process variables, notice that by the above construction each procedure variable in G yields $\mathcal{O}(pa^2)$ procedure variables in G^f, hence if pr is the number of procedure variables in G we find that the number of procedure variables in G^f is $\mathcal{O}(pa^2 \cdot pr)$.

A detailed proof of this lemma is given in an appendix. ☐

4.2.2 nDPK is in NP

The proof relies on results of [11, 23, 31], showing that Parikh images of context-free languages are semilinear sets, 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 $P = \{p_1, \ldots, p_m\} \subseteq \mathbb{N}^k$, we denote by $L(c; P)$ the subset of $\mathbb{N}^k$ defined as follows

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

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:

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

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 $[\![\phi]\!]$ is nonempty. Satisfiability of existential Presburger formulas is an NP-complete problem (see e.g. [31, Lemma 5]).

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 iff it is existential Presburger definable iff it is semilinear.

We use the following result of [31, Th. 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 [7] characterizes $\Pi(L(G))$ as the set of all multisets $\mathbf{m} \in \mathbb{M}[\mathcal{P}]$ that are solution of a certain system of linear equations, and for which a certain derived graph is connected. Then, [31, Th. 4] shows that this set of multisets is Presburger definable by explicitly constructing an existential Presburger formula in linear time in the size of G.

THEOREM 2. *nDMP is in NP.*

PROOF: By Lemma 1 it suffices to show that nDPK is in NP. Let $G_1, \ldots, G_g$ be an instance of nDPK, and let ϕ_{G_i} be the existential Presburger formula of [31, Th. 4] defining $\Pi(L(G_i))$. Let $\Psi = \phi_{G_1} \wedge \ldots \wedge \phi_{G_g}$. We have $[\![\Psi]\!] = \bigcap_{i=1}^{g} \Pi(L(G_i))$, and so Ψ is satisfiable iff $\bigcap_{i=1}^{g} \Pi(L(G_i)) \neq \varnothing$. 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. [31]), the result follows. ☐

5. Multiparameter analysis

From a verification point of view, it is important to analyze whether nDMP remains NP-complete or becomes polynomial for programs in which one or more of the following parameters is fixed: the number of threads, the maximal size of a procedure, the maximum number of procedures per thread, and the size of the pattern. In the formal model, these parameters correspond to the number of grammars g, the maximal size of a grammar sg, the maximal number of procedure variables in each grammar pr, and the size of the pattern pa. Since each parameter can be fixed or not, there are in principle 16 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 g and pr are fixed but sg and pa are not, is denoted by nDMP$(\widehat{g}, sg, \widehat{pr}, pa)$. Section 5.1 consider the cases in which the size of a grammar sg is *fixed*. Sections 5.2 and 5.3 deal with the more involved cases in which sg is not fixed, i.e., threads can have arbitrary size.

5.1 Fixed-sized grammars

In this section we assume that sg is fixed. Recall that the size of a grammar in program normal form is equal to the number

of variables plus the number of productions, but when for some terminal a the grammar contains a production $X \to aX$ for every variable X, then all those productions count together as one.

Observe that fixing the size sg of a grammar immediately fixes pr (the number of procedure variables of a grammar cannot be larger than its size). This leaves four cases, corresponding to the four combinations for fixed/nonfixed g and pa.

We first observe that if on top of sg and pr we fix at least another parameter (viz. g or pa), then each instance of nDMP can be reduced to one out of a constant number of nDPK instances, and so the problem can be trivially solved in polynomial time. So the only non-trivial case is nDMP($g, \widehat{sg}, \widehat{pr}, pa$), which corresponds to small but arbitrarily many threads, and an arbitrary pattern. This case remains NP-complete.

THEOREM 3. *The following problem is NP-hard:*
Instance: *Regular grammars $G_1, \dots, G_g$ in program normal form of fixed size, and a pattern $\boldsymbol{p}$.*
Question: *Is $\bigcap_{i=1}^{g} L(G_i) \cap L(\boldsymbol{p}) \neq \varnothing$?*

PROOF: (Sketch.) By reduction from 3-CNF-SAT. Let Ψ be a propositional formula with n variables $x_1, \dots, x_n$ and m clauses $c_1, \dots, c_m$. We define for each clause c_i a regular grammar G_i such that $\bigcap_{i=1}^{m} L(G_i) \cap L(\boldsymbol{p}) \neq \varnothing$ iff Ψ is satisfiable. Let $c_i = \ell_{i_1} \vee \ell_{i_2} \vee \ell_{i_3}$ where $1 \leq i_1 < i_2 < i_3 \leq n$ and $\ell_i \in \{x_i, \overline{x}_i\}$. We define G_i as a regular grammar for the language $\Sigma^*\ell_{i_1}\Sigma^*\ell_{i_2}\Sigma^*\ell_{i_3}\Sigma^*$, where $\Sigma = \{x_k, \overline{x}_k \mid k \notin \{i_1, i_2, i_3\}\}$. Observe that we can easily give a regular grammar G_i with four variables and four productions, plus productions of the form $X \to aX$, which are not counted in the size of the grammar. It is now easy to see that $\bigcap_{i=1}^{m} L(G_i)$ is the set of words $\ell_1 \dots \ell_n$ that correspond to a satisfying assignment of Ψ, and so by taking $\boldsymbol{p} = x_1^*(\overline{x}_1)^* \dots x_n^*(\overline{x}_n)^*$ we are done. □

5.2 Grammars of arbitrary size: NP-hard cases

Since sg is not fixed, there are three parameters, namely g, pr, and pa, that can still be fixed or not. We show that if *at least one* of these three parameters is *not* fixed, then nDMP remains NP-complete. In Section 5.3 we complete the analysis by proving that if *all three* parameters are fixed, then nDMP becomes polynomial.

We have already dealt with one case: Theorem 1 shows that nDMP($g, sg, \widehat{pr}, \widehat{pa}$) is NP-complete (in the theorem the grammars are regular, and so $pr = 0$, and the pattern is always a^*, and so $pa = 1$). This leaves two cases: nDMP($\widehat{g}, sg, pr, \widehat{pa}$), and nDMP($\widehat{g}, sg, \widehat{pr}, pa$). For nDMP($\widehat{g}, sg, pr, \widehat{pa}$) we show that nDMP remains NP-complete for two grammars and fixed pattern a^* by a reduction from the 0-1 Knapsack problem.

DEFINITION 3 (0–1 Knapsack Problem).
Instance: *(1) A set of objects $\{o_1, \dots, o_m\}$ and their associated weights $\{w_1, \dots, w_m\}$, which are positive integer given in binary. (2) A positive integer W given in binary.*
Question: *Is there a subset $S \subseteq \{o_1, \dots, o_m\}$ such that the total weight of S is equal to W ?*

THEOREM 4. *The following problem is NP-hard:*
Instance: *Two context-free grammars G_1, G_2 in program normal form over the alphabet $\{a\}$.*
Question: *Is $L(G_1) \cap L(G_2) \cap L(\boldsymbol{p}) \neq \varnothing$ for pattern $\boldsymbol{p} = a^*$?*

PROOF: 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 let n be the maximum 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 (2) through (7) shown below. Intuitively, a derivation of G nondeterministically selects a subset of objects as follows. The object o_i is selected by applying the production $S_i \to S_i^{(n)}$ (2), and omitted by applying $S_i \to S_{i+1}$ (3). If o_i has been selected, then the derivation outputs a^{w_i} through the variable $S_i^{(n)}$ using the productions in (4) and (5), and then comes back to S_{i+1} using production (6). Formally, we have $S_i^{(n)} \Rightarrow^* a^{w_i} \cdot S_{i+1}$. Indeed, observe that $w_i = \sum_{j=0}^{n} j\text{th bit of } w_i \times 2^j$, and the productions of (4)-(5) follow the binary encoding of w_i: if the j-th 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 (7) make use of a well-known encoding to ensure $L_{A_k}(G) = \left\{a^{2^k}\right\}$ for every $0 \leq k \leq n$. Finally, the axiom of G is S_1.

$$\left\{S_i \to S_i^{(n)} : 1 \leq i \leq m\right\} \tag{2}$$

$$\{S_i \to S_{i+1} : 1 \leq i \leq m-1\} \cup \{S_m \to \varepsilon\} \tag{3}$$

$$\left\{ S_i^{(k)} \to A_k \cdot S_i^{(k-1)} : \begin{array}{c} 1 \leq i \leq m \\ 1 \leq k \leq n \\ \text{bit } k \text{ of } w_i \text{ is } 1 \end{array} \right\} \tag{4}$$

$$\left\{ S_i^{(k)} \to S_i^{(k-1)} : \begin{array}{c} 1 \leq i \leq m \\ 1 \leq k \leq n \\ \text{bit } k \text{ of } w_i \text{ is } 0 \end{array} \right\} \tag{5}$$

$$\left\{S_i^{(0)} \to S_{i+1} : 1 \leq i \leq m-1\right\} \cup \left\{S_m^{(0)} \to \varepsilon\right\} \tag{6}$$

$$\{A_k \to A_{k-1}A_{k-1} : 1 \leq k \leq n\} \cup \{A_0 \to aZ\} \cup \{Z \to \varepsilon\} \tag{7}$$

We now turn to W, and define the grammar G_W by:

$$\left\{ W^{(k)} \to A_k \cdot W^{(k-1)} : \begin{array}{c} 1 \leq k \leq n \\ \text{bit } k \text{ of } W \text{ is } 1 \end{array} \right\} \tag{8}$$

$$\left\{ W^{(k)} \to W^{(k-1)} : \begin{array}{c} 1 \leq k \leq n \\ \text{bit } k \text{ of } W \text{ is } 0 \end{array} \right\} \cup \left\{W^{(0)} \to \varepsilon\right\} \tag{9}$$

where $W^{(n)}$ is the axiom. From the reasoning above we find that $L(G) = \left\{a^W\right\}$.

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

For nDMP($\widehat{g}, sg, \widehat{pr}, pa$), we show that nDMP remains NP-complete for three grammars, each of them with at most one procedure variable. The proof is by reduction from the bounded Post Correspondence Problem [10].

DEFINITION 4 (Bounded Post Correspondence Problem).
Instance. *Two sequences $a = (a_1, \dots, a_n)$ and $b = (b_1, \dots, b_n)$ of words over an alphabet Σ, and a positive integer $K \leq n$.*
Question: *Is there a non-empty sequence $i_1, \dots, i_k$ of $k \leq K$ (not necessarily distinct) positive integers, each between 1 and n, such that $a_{i_1}a_{i_2} \dots a_{i_k} = b_{i_1}b_{i_2} \dots b_{i_k}$.*

THEOREM 5. *The following problem is NP-hard:*
Instance: *Two context-free grammars G_1, G_2 in program normal form, each of them with 1 procedure variable, a regular grammar R in program normal form, and a pattern $\boldsymbol{p}$.*
Question: *Is $L(G_1) \cap L(G_2) \cap L(R) \cap L(\boldsymbol{p}) \neq \varnothing$?*

PROOF:(Sketch.) Let a, b, K be an instance of the bounded Post Correspondence Problem. Define $\Gamma = \{1, \dots, n\}$ and assume it is disjoint from Σ. We construct the context-free grammars $G_1 = (\{X\}, \Sigma \cup \Gamma, \mathcal{P}_1, X)$, where

$$\mathcal{P}_1 = \{X \to a_i \cdot X \cdot i \mid i \in \Gamma\} \cup \{X \to \varepsilon\} \ ,$$

$G_2 = (\{Y\}, \Sigma \cup \Gamma, \mathcal{P}_2, Y)$, where

$$\mathcal{P}_2 = \{Y \to b_i \cdot Y \cdot i \mid i \in \Gamma\} \cup \{Y \to \varepsilon\} \ ,$$

the regular grammar R such that $L(R) = \Sigma^* \cdot \bigcup_{i=1}^{K} \Gamma^i$, and the pattern $\boldsymbol{p} = (a_1^* \dots a_n^*)^K (1^* \dots n^*)^K$. Observe that, since $K \le n$, 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 $X \to a_i \cdot X \cdot i$ by productions $X \to a_i \cdot X_i', X_i' \to X \cdot X_i'', X_i'' \to i \cdot Z, Z \to \varepsilon$, 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 $L(G_1) \cap L(G_2) \cap L(R) \cap L(\boldsymbol{p}) \neq \varnothing$ iff the bounded PCP instance is positive. □

Notice that in this reduction, neither the number of words in $\boldsymbol{p}$ nor their length is fixed. By mean of a more involved reduction it is possible to show NP-hardness with a single word only (but arbitrarily long). This reduction is presented next.

5.2.1 A finer analysis

We have defined the size 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 the maximal length of a pattern. Since the reduction of Theorem 5 requires a pattern with a large number of words ($2n$ in the worst case), we study whether nDMP stays NP-complete if on top of the number of grammars g and the number pr of procedures also the number of words n in the pattern $\boldsymbol{p} = w_1^* \dots w_n^*$ is fixed, but not their length. We show that nDMP remains NP-hard by reduction to the 0-1 Knapsack problem of Def. 3.

Consider the reduction from 0-1 Knapsack shown in Th. 4. It does not yield a grammar with a fixed number of procedure variables because of the sets (7), (4), and (8) of productions.

To solve this problem we first construct a grammar $G^\sharp$ with a fixed number of procedure variables that can still be used to encode big numbers, albeit by means of a more complicated encoding. Fix a number $n \ge 1$ and an alphabet $\Sigma = \{a_0, a_1, \dots, a_n\} \cup \{b_1, \dots, b_n\}$, and let $w = a_n b_n \cdots a_1 b_1 a_0$. We encode the number $k \le 2^n$ by the word w^k. The grammar $G^\sharp = (\mathcal{X}^\sharp, \Sigma, \mathcal{P}^\sharp, X)$ has variables $\mathcal{X}^\sharp = \{X\} \cup \{A_1, \dots, A_n\}$ (X is the only procedure variable), and productions $\mathcal{P}^\sharp$ given by the union of the sets (10) to (14):

$$\{X \to A_n\} \tag{10}$$
$$\{X \to b_k A_{k-1} \mid 1 \le k \le n\} \tag{11}$$
$$\{A_k \to a_k X A_{k-1} \mid 1 \le k \le n\} \tag{12}$$
$$\{A_k \to a_j b_j A_k \mid n \ge j, k \ge 0 \wedge j > k\} \tag{13}$$
$$\{A_0 \to a_0\} \tag{14}$$

$G^\sharp$ can also be obtained as follows. We first apply the construction of Sect. 3.2 to the program shown in Figure 2. This returns a context-free grammar in program normal form. Second, some productions are merged for better readability.

Consider the pattern $\boldsymbol{p} = w^*$. Our first lemma shows that the language $L_{A_k}(G^\sharp) \cap L(\boldsymbol{p})$ consists of a unique word given by 2^k repetitions of w.

LEMMA 2. *$L_{A_k}(G^\sharp) \cap L(\boldsymbol{p}) = \left\{ w^{2^k} \right\}$ for every $0 \le k \le n$.*

PROOF: The proof is by induction on k.
$\mathbf{k = 0}$. The only word which can be derived from A_0 and follows $\boldsymbol{p}$ is given by $A_0 (\Rightarrow^{(13)})^* a_n b_n \dots a_1 b_1 A_0 \Rightarrow^{(14)} a_n b_n \dots a_1 b_1 a_0 = w^{2^0}$.

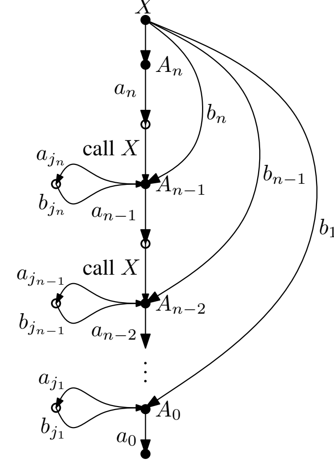

Figure 2. The abstracted program defining $G^\flat$. For every $i \in \{1, \dots, n\}$ we have $j_i \in \{i, \dots, n\}$ as in (13).

$\mathbf{k > 0}$. We distinguish two cases: $k < n$ and $k = n$. For $k < n$, consider the following partial leftmost derivation:

$$\begin{array}{lcl} A_k & (\Rightarrow^{(13)})^* & a_n b_n \dots a_{k+1} b_{k+1} A_k \\ & \Rightarrow^{(12)} & a_n b_n \dots a_{k+1} b_{k+1} a_k X A_{k-1} \\ & \Rightarrow^{(11)} & a_n b_n \dots a_{k+1} b_{k+1} a_k b_k A_{k-1} A_{k-1} \\ & (\Rightarrow^{(12)} \Rightarrow^{(11)})^* & a_n b_n \cdots a_1 b_1 A_0 A_0 A_1 \dots A_{k-1} \\ & \Rightarrow^{(14)} & a_n b_n \cdots a_1 b_1 a_0 A_0 A_1 \dots A_{k-1} \\ & = & w \cdot A_0 A_1 \dots A_{k-1} \end{array}$$

For the case $k = n$, consider

$$\begin{array}{lcl} A_n & (\Rightarrow^{(12)} \Rightarrow^{(11)})^* & a_n b_n \dots a_1 b_1 A_0 A_0 A_1 \dots A_{n-1} \\ & \Rightarrow^{(14)} & w \cdot A_0 A_1 \dots A_{n-1} \ . \end{array}$$

We only need these two partial derivations, because every leftmost derivation that does not start like one of the two does not generates a word of $L(\boldsymbol{p})$ either. To conclude, we apply the induction hypothesis on $A_0 A_1 \dots A_{k-1}$ to show that $A_k \Rightarrow^* w \cdot w^{2^0} \cdot w^{2^1} \cdots w^{2^{k-1}}$, 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 nDMP in polynomial time. If in the reduction of Th. 4 the set (7) is replaced by the set $\mathcal{P}^\sharp$, we get $L(G) \cap L(G_W) \cap L(\boldsymbol{p}) \neq \varnothing$ iff there exists $S \subseteq \{o_1, \dots, o_m\}$ such that S's weight is W. However, this reduction is not yet adequate because the variables $\{A_1, \dots, A_n\}$ are procedures variables (see the sets (4) and (8) of productions). To fix this problem, we need a second lemma:

LEMMA 3. *(1) $L(G^\sharp) \cap L(\boldsymbol{p}) = \left\{ w^{2^n} \right\}$.*
(2) $\left(\{a_n b_n \dots a_{k+1}\} \cdot L(G^\sharp)\right) \cap L(\boldsymbol{p}) = \left\{ w^{2^k} \right\}$ for all $1 \le k \le n-1$.

PROOF: (1) Any derivation of $G^\sharp$ that generates a word of $L(\boldsymbol{p})$ must use the production (10) first, so that $X \Rightarrow^{(10)} A_n$. Applying Lem. 2 we get $L(G^\sharp) \cap L(\boldsymbol{p}) = L_{A_n}(G) \cap L(\boldsymbol{p}) = \left\{ w^{2^n} \right\}$.
(2) Any derivation of $G^\sharp$ generating a word u such that $a_n b_n \dots a_{k+1} u$ belongs to $L(\boldsymbol{p})$ must start with $X \Rightarrow^{(11)} b_{k+1} \cdot A_k$. As shown in the proof of Lem. 2, the derivation must continue with $X \Rightarrow^{(11)} b_{k+1} \cdot A_k \Rightarrow^* w \cdot A_0 A_1 \cdots A_{k-1}$, and so finally lead to w^{2^k}. □

We the help of this lemma we can now proceed as follows. Recall that we have already replaced set (7) in the reduction of

Th. 4 by $\mathcal{P}^\sharp$. Now we replace the set (4) by

$$\left\{ S_i^{(k)} \to a_n b_n \cdots a_{k+1} \cdot X \cdot S_i^{(k-1)} : \begin{array}{c} 1 \le i \le m \\ 1 \le k \le n \\ \text{bit } k \text{ of } W \text{ is } 1 \end{array} \right\}$$

and the set (8) by

$$\left\{ W^{(k)} \to a_n b_n \cdots a_{k+1} \cdot X \cdot W^{(k-1)} : \begin{array}{c} 1 \le k \le n \\ \text{bit } k \text{ of } W \text{ is } 1 \end{array} \right\}$$

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

THEOREM 6. *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 1 procedure variable, and a pattern $\boldsymbol{p} = w^*$ consisting of a single word $w \in \Sigma^*$.*
Question: *Is $L(G_1) \cap L(G_2) \cap L(\boldsymbol{p}) \neq \varnothing$?*

PROOF: The proof is by reduction to 0-1 Knapsack. We construct $G_1^{\bowtie}, G_2^{\bowtie}$ and $\boldsymbol{p}$ as above. The proof of correctness for the reduction essentially follows the one of Th. 4 where the result of Lem. 3 is used when needed. We thus obtain that $L(G_1^{\bowtie}) \cap L(G_2^{\bowtie}) \cap L(\boldsymbol{p}) \neq \varnothing$ iff 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 of $G_i^{\bowtie}$ where $i \in \{1,2\}$ and (3) $\boldsymbol{p} = w^*$ is computable in polynomial time. Note that $G_i^{\bowtie}$ are 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. □

5.3 Grammars of arbitrary size: a polynomial case

We present the most involved result of the paper, a polynomial algorithm for nDMP($\widehat{g}, sg, \widehat{pr}, \widehat{pa}$). Notice that the reduction of nDMP to satisfiability of existential Presburger formulas of Th. 2 does not help, because it yields formulas of arbitrary size that, to the best of our knowledge, do not fall immediately into any polynomial class described in the literature (see e.g. [12, 22, 27]). However, using some recent results of [19, 29] we show how to compute in polynomial time an equisatisfiable formula that belongs to the polynomial class of [27].

As a first step we observe that, because of the reduction from nDMP to nDPK shown in Section 4.2, it suffices to provide a polynomial algorithm for nDPK, and in fact only for the instances of nDPK with a fixed number g of grammars over an alphabet of fixed size al, and a fixed number of procedure variables pr', i.e., a polynomial procedure for nDPK($\widehat{g}, sg, \widehat{al}, \widehat{pr'}$). Indeed, Lem. 1 shows that (1) al, the size of the alphabet in the reduced nDPK instance, is fixed, because pa is fixed in nDMP, and (2) that pr', the number of procedure variables in the reduced nDPK instance, is fixed since pa and pr are fixed.

Let $G_1, \ldots, G_g$ be an instance of nDPK($\widehat{g}, sg, \widehat{al}, \widehat{pr}$). The polynomial algorithm proceeds in two steps: first, for each $i \in \{1, \ldots, g\}$ the algorithm computes a regular grammar (or non-deterministic automaton) A_i such that $\Pi(L(A_i)) = \Pi(L(G_i))$; then, the algorithm checks if $\bigcap_{i=1}^{g} \Pi(L(A_i)) = \varnothing$. The difficulty consists of showing that both steps can be carried out in polynomial time. For this we prove two facts. First, if $G_i = (\mathcal{X}, \Sigma, \mathcal{P}, S)$, then the algorithm constructs A_i in $\mathcal{O}(|G_i|^{f(pr)})$ time and space for some function f. Second, given automata $A_1, \ldots, A_g$ over an alphabet of size al, the algorithm performs the check $\bigcap_{i=1}^{g} \Pi(L(A_i)) = \varnothing$ in $\mathcal{O}\big((|A_1| + \cdots + |A_g|)^{h(g,al)}\big)$ time for some function h. Since g, pr, and al have fixed values, so do $f(pr)$ and $h(g, al)$.

Step 1. We show that given a context-free grammar G in program normal form with pr variables, we can construct a regular grammar A_G satisfying $\Pi(L(A_G)) = \Pi(L(G))$ in $\mathcal{O}(|G_i|^{f(pr)})$ time and space (for some function f). For this we strengthen a recent result of [8], which shows that such a grammar can be constructed in $\mathcal{O}(|G|^{f(v)})$ time and space, where v is the *total* number of variables of G. We start by defining the grammar A_G.

DEFINITION 5. *Let $G = (\mathcal{X}, \Sigma, \mathcal{P}, S)$ be a context-free grammar in program normal form, and let pr be the number of procedure variables of G. We define the regular grammar $A_G = (\mathcal{Q}, \Sigma, \delta, q)$ as follows:*

- *$\mathcal{Q}$ is the set of all multisets $\mathbf{m} \in \mathbb{M}[\mathcal{X}]$ of at most $(pr + 2)$ elements, and q, the axiom, is given by $[\![S]\!]$;*
- *$\delta = \{\emptyset \to \epsilon\} \cup \delta'$, where δ' contains a production $\mathbf{m} \to \alpha \cdot \mathbf{m}'$ iff $\mathcal{P}$ contains a production $X \to \alpha\beta$, such that $\alpha \in \Sigma^*$, $\beta \in \mathcal{X}^*$, and $\mathbf{m}' \oplus [\![X]\!] = \mathbf{m} \oplus \Pi(\beta)$.*

Observe that $|\mathcal{Q}| = \mathcal{O}(|\mathcal{X}|^{pr+2})$ and $|\delta| \le |\mathcal{Q}|^2 \cdot al$. We set out to prove the following result (see Theorem 7 in the next page) by means of several lemmas:

> Let $G = (\mathcal{X}, \Sigma, \mathcal{P}, S)$ be a context-free grammar in program normal form. The regular grammar A_G of Def. 5 satisfies $\Pi(L(G)) = \Pi(L(A_G))$.

We first introduce some new notation. Given $L_1, L_2 \subseteq \Sigma^*$, we write $L_1 =_\Pi L_2$, respectively $L_1 \subseteq_\Pi L_2$, to denote that the Parikh image of L_1 is equal to, respectively included in, the Parikh image of L_2. Also, given $w, w' \in \Sigma^*$, we abbreviate $\{w\} =_\Pi \{w'\}$ to $w =_\pi w'$. Using this notation we can rewrite our proof goal $\Pi(L(G)) = \Pi(L(A_G))$ as $L(A_G) =_\Pi L(G)$.

The proof is a modification of the one given in [8]. The inclusion $L(A_G) \subseteq_\Pi L(G)$ is proved in [8, Prop 2.1]. Establishing $L(G) \subseteq_\Pi L(A_G)$ is done through the chain of inclusions $L(G) \subseteq_\Pi L^{(pr+2)}(G) \subseteq_\Pi L(A_G)$, where $L^{(i)}(G)$ is the index-i approximation of $L(G)$, defined as follows.

DEFINITION 6. *A derivation $S = \alpha_0 \Rightarrow \cdots \Rightarrow \alpha_m$ of $G = (\mathcal{X}, \Sigma, \mathcal{P}, S)$ has index k if for every $i \in \{0, \ldots, m\}$ at most k symbols of α_i are variables. The set of words derivable through derivations of index k is denoted by $L^{(k)}(G)$.*

The inclusion $L^{(pr+2)}(G) \subseteq_\Pi L(A_G)$ is proved in [8, Lem. 2.4]. To prove $L(G) \subseteq_\Pi L^{(pr+2)}(G)$ we need a few preliminaries.

DEFINITION 7. *Let $G = (\mathcal{X}, \Sigma, \mathcal{P}, S)$ be a context-free grammar in program normal form. We inductively define the set Tr of finite labelled trees as follows:*

- *if $(X, \varepsilon) \in \mathcal{P}$ then the tree t labelled by production (X, ε) and consisting of one single node is a tree of Tr, and its yield $\Delta(t)$ is equal to ε;*
- *if $(X, a \cdot Y) \in \mathcal{P}$, then the tree t labelled by $(X, a \cdot Y)$ and having as only child a tree $t' \in Tr$ labelled by some $(Y, \alpha) \in \mathcal{P}$ is a tree of Tr, and $\Delta(t) = a \cdot \Delta(t')$;*
- *if $(X, Y) \in \mathcal{P}$, then the tree t labelled by (X, Y) and having as only child a tree $t' \in Tr$ labelled by some $(Y, \alpha) \in \mathcal{P}$ is a tree of Tr, and $\Delta(t) = \Delta(t')$;*
- *if $(X, Z \cdot Y) \in \mathcal{P}$, then the tree t labelled by $(X, Z \cdot Y)$ and having two children labelled by some (Z, α_1) (left) and (Y, α_2) (right) is also a tree of Tr, and $\Delta(t) = \Delta(t_1) \cdot \Delta(t_2)$.*

A tree $t \in Tr$ is a derivation tree *if it is labelled by a production $(S, \alpha) \in \mathcal{P}$ for some α. The set of all derivation trees of G is denoted by T_G. The* yield *$\Delta(T)$ of a countable set $T \subseteq Tr$ of trees is defined by $\Delta(T) = \bigcup_{t \in T} \Delta(t)$. In the following, we mean* derivation tree *whenever we say* tree.

LEMMA 4 (Easy). *Let $G = (\mathcal{X}, \Sigma, \mathcal{P}, S)$ be a context-free grammar in program normal form. Then $L(G) = \Delta(T_G)$.*

By Lem. 4, proving $L(G) \subseteq_\Pi L^{(pr+2)}(G)$ reduces to proving $\Delta(T_G) \subseteq_\Pi L^{(pr+2)}(G)$. We now introduce the notion of dimension of a tree.

DEFINITION 8. *The* dimension $d(t)$ *of a tree t is inductively defined as follows:*

1. *If t has no children, then $d(t) = 0$;*
2. *If t has exactly one child t_1, then $d(t) = d(t_1)$;*
3. *If t has exactly two children t_1 and t_2, then*

$$d(t) = \begin{cases} d(t_1) + 1 & \text{if } d(t_1) = d(t_2) \\ \max(d(t_1), d(t_2)) & \text{if } d(t_1) \neq d(t_2). \end{cases}$$

The set of all derivation trees of dimension k for grammar G is denoted by T_G^k.

The next lemma goes along the lines of [8, Lem. 2.1], but with many small changes.

LEMMA 5. *Let $G = (\mathcal{X}, \Sigma, \mathcal{P}, S)$ be a context-free grammar in program normal form with pr procedure variables. Then $\Delta(T_G) \subseteq_\Pi \bigcup_{i=0}^{pr+1} \Delta(T_G^i)$.*

PROOF: In this proof we write $t = t_1 \cdot t_2$ to denote that t_1 is a derivation tree except that exactly one leaf ℓ is labelled by a production of the form (A, α) with $\alpha \neq \varepsilon$; t_2 is a derivation tree labelled (A, α') for some α'; and the tree t is obtained from t_1 and t_2 by replacing the leaf $\ell = (A, \alpha)$ of t_1 by t_2.

We want to prove that for every tree $t \in T_G$, there exists a tree t' such that $\Delta(t) =_\Pi \Delta(t')$ and $d(t') \leq pr + 1$. Let a tree t be *compact* if $d(t) \leq L(t)$, where $L(t) = 1 + L'(t)$ and $L'(t)$ is the number of distinct procedure variables in t. We find that $L'(t) \leq pr$ for every $t \in T_G$, hence $L(t) \leq pr + 1$. To establish the above result, it suffices to show that for every tree t, there exists a compact tree t' such that $\Delta(t) =_\Pi \Delta(t')$.

The proof is by induction on the number of nodes of t. In the base case, t has just one node labelled (S, ε), so $d(t) = 0 < 1 \leq L(t)$, hence t is compact, and we are done. In the following, assume that t has more than one node and $d(t) > L(t)$ holds. If t has exactly one child t_1 then $d(t) = d(t_1) > L(t)$. Since t_1 has one node less than t, induction hypothesis shows that t_1 can be made compact, i.e. $d(t_1) \leq L(t_1)$. Next we conclude from the definition of L and the structure of t that $L(t_1) \leq L(t)$, also that $d(t) = d(t_1)$ and finally that $d(t) \leq L(t)$ and we are done. Let us turn to the case where t has two children t_1 and t_2. We assume w.l.o.g. that $d(t) \geq d(t_1) \geq d(t_2)$. Finally, by the induction hypothesis, we can further assume that t_1 and t_2 are compact, i.e. $d(t_i) \leq L(t_i)$ for $i = 1, 2$.

From the definition of dimension and L, t_1 is a subtree of t, and $d(t) > L(t)$ we find that: $L(t) + 1 \leq d(t) \leq d(t_1) + 1 \leq L(t_1) + 1 \leq L(t) + 1$. We conclude from above that $d(t_1) = L(t)$ and $d(t) = d(t_1) + 1$, hence that $d(t_1) = d(t_2)$ by definition of dimension and finally that $d(t_1) = d(t_2) = L(t) = L(t_1) = L(t_2)$ since t_1, t_2 are compact subtrees of t. We now prove the following claim: there is a path in t_2 from the root to a leaf such that two nodes are labelled by (Z, α) and (Z, α') where Z is a procedure variable.

Our proof is by contradiction. Observe that for derivation tree t with child t' such that $d(t) > d(t')$, the definition of dimension and program normal form shows that t is labelled by $(X, Z \cdot Y)$ for some variables X, Z and Y. If $d(t) = k$ then the rooted path that goes down through the left child whenever possible has at least k nodes with label of the form (Z, α) where Z is a procedure variable. Finally since $d(t_2) = L(t_2) = L'(t_2) + 1$ where $L'(t_2)$ is the number of distinct procedure variables in t_2, we find that two nodes are labelled (Z, α) and (Z, α') where Z is a procedure variable, hence a contradiction.

So t_2 can be factored into $t_2^a \cdot (t_2^b \cdot t_2^c)$ such that t_2^b and t_2^c have their root labelled (Z, α) and (Z, α') where Z is a procedure variable.

As $L(t) = L(t_1) = L(t_2)$, we also find a node of t_1 labelled (Z, α) where Z is a procedure variable which allows us to write $t_1 = t_1^a \cdot t_1^b$ where t_1^b has its root labelled (Z, α) where Z is a procedure variable.

Now we cut out the middle part t_2^b of t_2, and insert it between the two parts t_1^a and t_1^b of t_1, so that we get $t_1' = t_1^a \cdot (t_2^b \cdot t_1^b)$ and $t_2' = t_2^a \cdot t_2^c$. We then have $L(t_1') = L(t_1) = L(t_2) \geq L(t_2')$. By induction, t_1' and t_2' can be made compact, so $d(t_1') \leq L(t_1') = d(t_1) = d(t_2) \geq L(t_2') \geq d(t_2')$. Consider the tree t' obtained from t by replacing t_1 by t_1' and t_2 by t_2'. Clearly, $\Delta(t) =_\Pi \Delta(t')$. If $d(t_1') < d(t_1)$ or $d(t_2') < d(t_2)$, then $d(t') \leq d(t) - 1 = L(t) = L(t')$ by definition of dimension, and we are done because t' is compact. Otherwise, we have $d(t_1') = d(t_2') = L(t') = L(t_1') = L(t_2')$. So we can iterate the above procedure and insert a part of t_2' into t_1'. This procedure terminates, because the transfer of nodes from the second child to the first cannot proceed forever. □

By this lemma, proving $\Delta(T_G) \subseteq_\Pi L^{(pr+2)}(G)$ reduces to showing $\Delta(T_G) \subseteq_\Pi \bigcup_{i=0}^{pr+1} \Delta(T_G^i) \subseteq_\Pi L^{(pr+2)}(G)$. To conclude the proof we show $\bigcup_{i=0}^{pr+1} \Delta(T_G^i) \subseteq L^{(pr+2)}(G)$.

LEMMA 6. *For every $k \geq 0$: $\Delta(T_G^k) \subseteq L^{(k+1)}(G)$.*

PROOF: Let t be a derivation tree of dimension k. The proof is by induction on the structure of t.

Base. t consists of a node labelled (S, ε), hence $k = 0$ and $S \Rightarrow \varepsilon$ is of index 1.

Step. W.l.o.g. t has two children t_1 and t_2 such that $d(t) \geq d(t_1) \geq d(t_2)$. By the definition of dimension we have $d(t_2) \leq k - 1$. Let A, A_1 and A_2 be the roots of t, t_1 and t_2, respectively. Then $A \to A_1 A_2$ is a production of the grammar. By induction hypothesis, there are derivations $A_1 \Rightarrow^* w_1$ of index $k + 1$ and $A_2 \Rightarrow^* w_2$ of index k. So there is a derivation $A \Rightarrow A_1 A_2 \Rightarrow^* A_1 w_2 \Rightarrow^* w_1 w_2$ of index $k + 1$. □

Collecting the results above, we get:

THEOREM 7. *For every context-free grammar G in program normal form with pr procedure variables, $\Pi(L(A_G)) = \Pi(L(G))$.*

PROOF: In [8, Prop 2.1], $L(A) \subseteq_\Pi L(G)$ has been proved. For the reverse inclusion:

$$\begin{aligned} L(G) &= \Delta(T_G) && \text{Lem. 4} \\ &\subseteq_\Pi \textstyle\bigcup_{i=0}^{pr+1} \Delta(T_G^i) && \text{Lem. 5} \\ &\subseteq L^{(pr+2)}(G) && \text{Lem. 6} \\ &\subseteq_\Pi L(A_G) && \text{[8, Lem. 2.4]} \end{aligned}$$

□

COROLLARY 1. *Given a context-free grammar G with pr procedure variables, we can construct in $\mathcal{O}(|G|^{(pr+2)})$ time a regular grammar A_G such that $\Pi(L(G)) = \Pi(L(A_G))$.*

PROOF: Follows immediately from the fact that the number of variables of the regular grammar A_G of Def.5 for a context-free grammar G with n variables and pr procedure variables is $\mathcal{O}(n^{pr+2})$. □

Step 2. Given regular grammars $A_1, \ldots, A_g$ over an alphabet of size al, we show that $\bigcap_{i=1}^{g} \Pi(L(A_i)) = \varnothing$ can be checked in time $\mathcal{O}((|A_1| + \cdots + |A_g|)^{h(g,al)})$ for a function h.

It is well known that for every regular grammar A, the set $\Pi(L(A))$ is semilinear. It has been recently proved that $\Pi(L(A))$ is "small".

THEOREM 8. **[29, Th. 4.1]** *Let A be a regular grammar with n variables over alphabet Σ of size al. There exists a representation of $\Pi(L(A)) \subseteq \mathbb{N}^{al}$ as a union of linear sets $\bigcup_{j=1}^{m} L(c_j; P_j)$, where m is polynomial in n and exponential in al, the maximum entry of each c_j is polynomial in n and exponential in al, the number of periods in each P_j is at most al, and the maximum entry of each period is at most n. Furthermore, this is computable in time polynomial in n and exponential in al.*

This theorem suggests the following procedure to check $\bigcap_{i=1}^{g} \Pi(L(A_i)) = \varnothing$ for fixed g. First, compute for each A_i a representation of $\Pi(L(A_i))$ as given above. This is done in polynomial time in the number of variables of A_i since Σ is of fixed size. Then, for each tuple $\langle L(c_1; P_1), \ldots, L(c_g; P_g)\rangle$, where $L(c_i; P_i)$ is a linear set of $\Pi(L(A_i))$, check if $\cap_{i=1}^{g} L(c_i; P_i) = \varnothing$. Since g and al are fixed, the number of tuples is polynomial, and so in order to obtain a polynomial procedure we just need to prove that $\cap_{i=1}^{g} L(c_i; P_i) = \varnothing$ can be checked in polynomial time. For this we first reduce the problem to solving a system of linear equations over the natural numbers.

DEFINITION 9. *Let $t = \langle L(c_1; P_1), \ldots, L(c_g; P_g)\rangle$ be a tuple of linear sets of dimension al, where $P_i = \left\{p_i^{(1)}, \ldots, p_i^{(j_i)}\right\}$. The existential Presburger formula Φ_t is given by*

$$\exists x_1^{(1)}, \ldots, x_1^{(j_1)}, \ldots, x_g^{(1)}, \ldots, x_g^{(j_g)} : \bigwedge_{\substack{1 \le id \le g-1 \\ 1 \le \ell \le al}} \phi(id, id+1, \ell)$$

where $\phi(id, id', \ell)$ denotes the formula

$$c_{id}(a_\ell) + \sum_{i=1}^{j_{id}} x_{id}^{(i)} * p_{id}^{(i)}(a_\ell) = c_{id'}(a_\ell) + \sum_{i=1}^{j_{id'}} x_{id'}^{(i)} * p_{id'}^{(i)}(a_\ell) \ .$$

In the above definition, the subformula $\phi(id, id', \ell)$ has the following interpretation: $\phi(id, id', \ell)$ is satisfiable iff there exist $v \in L(c_{id}; P_{id})$ and $v' \in L(c_{id'}; P_{id'})$ such that $v(\ell) = v'(\ell)$. Therefore $\bigwedge_{1 \le \ell \le al} \phi(id, id', \ell)$ is satisfiable iff $L(c_{id}; P_{id})$ and $L(c_{id'}; P_{id'})$ have a common vector, i.e., iff $L(c_{id}; P_{id}) \cap L(c_{id'}; P_{id'}) \neq \varnothing$. So $\bigwedge_{1 \le id \le g-1} \bigwedge_{1 \le \ell \le al} \phi(id, id+1, \ell)$ is satisfiable iff $\cap_{i=1}^{g} L(c_i; P_i) \neq \varnothing$. Hence the following result.

LEMMA 7. *Let $t = \langle L(c_1; P_1), \ldots, L(c_g; P_g)\rangle$ be a tuple of linear sets. We have $\cap_{i=1}^{g} L(c_i; P_i) \neq \varnothing$ iff Φ_t is satisfiable.*

Assume now that the maximum entries and number of periods of the linear sets in the tuple t of Def. 9 are as given in Th. 8. An inspection of the formula Φ_t in Def. 9 shows that in this case the number of variables of Φ_t is at most $g * al$, and so that Φ_t is an existential Presburger formula with $g * al$ quantifiers and no free variables. Since g and al are fixed parameters, $g * al$ is also fixed. It follows that the satisfiability of Φ_t can be determined in polynomial time by means of the Lenstra-Scarpellini's algorithm [22, 27] (see also [12]).

This concludes the proof that given regular grammars $A_1, \ldots, A_g$ over an alphabet of size al, whether $\bigcap_{i=1}^{g} \Pi(L(A_i)) \neq \varnothing$ holds or not can be determined in polynomial time.

6. Conclusions

We have studied the complexity of pattern-based verification, an approach to the verification of multithreaded programs essentially introduced by Kahlon in [15]. The approach asks the programmer to supply a pattern, a regular expression of the form $w_1^* \ldots w_n^*$ over the alphabet of channels (and possibly other program instructions). The verification tool then analyzes whether the program has some execution that uses the channels conforming to the pattern.

The expressivity of pattern-based verification was first investigated in [9], where it was shown that context bounding, the technique introduced by Qadeer and Rehof in [25] and implemented in CHESS, SPIN, SLAM, jMoped, and other tools [1, 5, 20, 28, 30], is a special case of pattern-based verification. In this paper we provide a further analysis and give a explicit reduction.

We have reduced the pattern-based verification problem to nDMP, the problem of deciding whether the intersection of a given set of context-free grammars and a pattern is nonempty. Putting together classical results by Ginsburg and Spanier [11] about bounded context-free languages; the characterization of the Parikh images of context-free languages given in [7]; the encoding of this characterization into existential Presburger arithmetic presented in [31]; and the fact that existential Presburger arithmetic reduces to solving a system of linear Diophantine equations (well-known to be in NP [32]) we have shown that nDMP is NP-complete. Since context bounding is also NP-complete, the additional expressivity of pattern-based verification does not come at an extra cost in terms of asymptotic complexity.

We have conducted a multiparameter analysis of nDMP on the number of threads, the maximum number of procedures per thread, the maximal size of a procedure, and the size of the pattern. By requiring the value of a parameter to be fixed or not, we get 16 cases. We have shown that all *except one* are either trivially polynomial or still NP-complete. The analysis of the remaining case (all parameters fixed except the maximal size of a procedure) is the main technical contribution of the paper. Using a novel constructive proof of Parikh's theorem and recent results about the Parikh images of nondeterministic automata [19, 29], we have shown that this case is polynomial. Given the high complexity of automatic verification of multithreaded procedural programs (unless strong constraints like absence of communication between threads or restriction to local properties of a thread are imposed) we think that this is a remarkable result.

Two comments about our model are in order. First, while we have only considered abstracted programs (i.e., we assume that all program paths with (1) correct nesting of procedure calls and returns, and (2) correct synchronization over channels, are feasible), our approach can also be applied to boolean programs at the price of an increase in the size of the procedures and the number of procedures per thread. Notice that context bounding and other techniques face the same problem. Second, we have opted for a communication model based on rendez-vous à la CSP. The reason is convenience: the connection between the verification problem and the emptiness problem for the intersection of context-free grammars is easier to describe in this model. Our approach can also be applied to other communication mechanisms by suitably choosing the alphabet of the patterns.

Related work. The automatic verification of safety properties for multithreaded programs with possibly recursive procedures has been intensively studied in the last 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 multi-pushdown systems, and systems with acyclic communication structure (also satisfying some additional conditions) [2, 3, 14–17]. Several recent papers study the automatic verification of parametric programs with an arbitrary number of procedures [18] and with dynamic creation of procedures [4, 6], two features that we have not considered in this paper. From a complexity point of view, pattern-based ver-

ification 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 [17].

Acknowledgement. Many thanks to Mikołaj Bojańczyk for providing the proof of Theorem 1, to Anthony Widjaja To for suggesting a more direct polynomial time algorithm using [22, 27], to Michael Luttenberger and Andrey Rybalchenko for helpful discussions, and finally to the anonymous reviewers for their helpful comments and to Tomas Poch for pointing typos in the manuscript.

References

[1] CHESS: Find and reproduce heisenbugs in concurrent programs. URL http://research.microsoft.com/en-us/projects/CHESS/.

[2] M. F. Atig, B. Bollig, and P. Habermehl. Emptiness of multi-pushdown automata is 2EXPTIME-complete. In *DLT '08: Proc. 12th Int. Conf. on Developments in Language Theory*, volume 5257 of *LNCS*, pages 121–133. Springer, 2008.

[3] M. F. Atig, A. Bouajjani, and T. Touili. On the reachability analysis of acyclic networks of pushdown systems. In *CONCUR '08: Proc. 19th Int. Conf. on Concurrency Theory*, volume 5201 of *LNCS*, pages 356–371. Springer, 2008.

[4] M. F. Atig, A. Bouajjani, and S. Qadeer. Context-bounded analysis for concurrent programs with dynamic creation of threads. In *TACAS '09: Proc. 15th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 5505 of *LNCS*, pages 107–123. Springer, 2009.

[5] T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL '02: Proc. 29th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 1–3. ACM Press, 2002.

[6] A. Bouajjani, M. Müller-Olm, and T. Touili. Regular symbolic analysis of dynamic networks of pushdown systems. In *CONCUR '05: Proc. 16th Int. Conf. on Concurrency Theory*, volume 3653 of *LNCS*, pages 473–487. Springer, 2005.

[7] J. Esparza. Petri nets, commutative context-free grammars, and basic parallel processes. *Fundamenta Informaticae*, 31:13–26, 1997.

[8] J. Esparza, P. Ganty, S. Kiefer, and M. Luttenberger. Parikh's theorem: A simple and direct construction. *CoRR*, 1006.3825, 2010.

[9] P. Ganty, B. Monmege, and R. Majumdar. Bounded underapproximations. In *CAV '10: Proc. 20th Int. Conf. on Computer Aided Verification*, volume 6174 of *LNCS*, pages 600–614. Springer, 2010.

[10] M. R. Garey and D. S. Johnson. *Computers and Intractability, A Guide to the Theory of NP-Completeness*. New York, 1979.

[11] S. Ginsburg and E. H. Spanier. Semigroups, presburger formulas, and languages. *Pacific Journal of Mathematics*, 16(2):285–296, 1966.

[12] E. Graedel. Subclasses of presburger arithmetic and the polynomial-time hierarchy. *Theoretical Computer Science*, 56(3):289 – 301, 1988.

[13] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1st edition, 1979.

[14] V. Kahlon. Boundedness vs. unboundedness of lock chains: Characterizing decidability of pairwise cfl-reachability for threads communicating via locks. In *LICS '09: Proc. 24th Annual IEEE Symp. on Logic in Computer Science*, pages 27–36. IEEE Computer Society, 2009.

[15] V. Kahlon. Tractable dataflow analysis for concurrent programs via bounded languages. Patent WO/2009/094439, July 2009.

[16] V. Kahlon and A. Gupta. On the analysis of interacting pushdown systems. In *POPL '03: Proc. 30th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 303–314. ACM, 2007.

[17] V. Kahlon, F. Ivancic, and A. Gupta. Reasoning about threads communicating via locks. In *CAV '05: Proc. 17th Int. Conf. on Computer Aided Verification*, volume 3576 of *LNCS*, pages 505–518. Springer, 2005.

[18] A. Kaiser, D. Kroening, and T. Wahl. Dynamic cutoff detection in parameterized concurrent programs. In *CAV '10: Proc. 20th Int. Conf. on Computer Aided Verification*, volume 6174 of *LNCS*. Springer, 2010.

[19] E. Kopczyński and A. W. To. Parikh images of grammars: Complexity and applications. In *LICS '10: Proc. 25th Annual IEEE Symp. on Logic in Computer Science*. IEEE, 2010.

[20] A. Lal and T. Reps. Reducing concurrent analysis under a context bound to sequential analysis. In *CAV '08: Proc. 20th Int. Conf. on Computer Aided Verification*, volume 5128 of *LNCS*, pages 37–51. Springer, 2008.

[21] A. Lal, T. Touili, N. Kidd, and T. W. Reps. Interprocedural analysis of concurrent programs under a context bound. In *TACAS '08: Proc. 14th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *LNCS*, pages 282–298. Springer, 2008.

[22] J. Lenstra, H. W. Integer programming with a fixed number of variables. *Mathematics of Operations Research*, 8(4):538–548, 1983.

[23] R. J. Parikh. On context-free languages. *Journal of the ACM*, 13(4): 570–581, 1966.

[24] S. Qadeer. The case for context-bounded verification of concurrent programs. In *SPIN '08: Proc. of 15th Int. Model Checking Software Workshop*, volume 5156 of *LNCS*, pages 3–6. Springer, 2008.

[25] S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *TACAS '05: Proc. 11th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 3440 of *LNCS*, pages 93–107. Springer, 2005.

[26] G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM TOPLAS*, 22(2):416–430, 2000.

[27] B. Scarpellini. Complexity of subcases of presburger arithmetic. *Trans. of the American Mathematical Society*, 284(1):203–218, 1984.

[28] D. Suwimonteerabuth, J. Esparza, and S. Schwoon. Symbolic context-bounded analysis of multithreaded java programs. In *SPIN '08: Proc. of 15th Int. Model Checking Software Workshop*, volume 5156 of *LNCS*, pages 270–287. Springer, 2008.

[29] A. W. To. Parikh images of regular languages: Complexity and applications. *CoRR*, 1002.1464, 2010.

[30] S. La Torre, G. Parlato, and P. Madhusudan. Reducing context-bounded concurrent reachability to sequential reachability. In *CAV'09: Proc. 21st Int. Conf. on Computer Aided Verification*, volume 5643 of *LNCS*, pages 477–492. Springer, 2009.

[31] K. N. Verma, H. Seidl, and T. Schwentick. On the complexity of equational horn clauses. In *CADE '00: 20th Int. Conf. on Automated Deduction*, volume 1831 of *LNCS*, pages 337–352. Springer, 2005.

[32] J. von zur Gathen and M. Sieveking. A bound on solutions of linear integer equalities and inequalities. *Proc. of the American Mathematical Society*, 72:155–158, 1978.

A. Reduction of nDMP to nDPK

A.1 Construction of the grammar G^f

Let $\boldsymbol{p} = w_1^* \dots w_n^*$, and let $w_i = b_1^{(i)} \dots b_{j_i}^{(i)}$ for every $1 \leq i \leq n$. Let $G^{\boldsymbol{p}} = (\mathcal{X}^{\boldsymbol{p}}, \Sigma, \delta^{\boldsymbol{p}}, q_1^{(1)})$ be the regular grammar where

$$\begin{aligned}
\mathcal{X}^{\boldsymbol{p}} &= \left\{ q_r^{(s)} \mid 1 \leq s \leq n \wedge 1 \leq r \leq j_s \right\} \\
\delta^{\boldsymbol{p}} &= \left\{ q_i^{(s)} \to b_i^{(s)} q_{i+1}^{(s)} \mid 1 \leq s \leq n \wedge 1 \leq i < j_s \right\} \cup \\
&\quad \left\{ q_{j_s}^{(s)} \to b_{j_s}^{(s)} q_1^{(s')} \mid 1 \leq s \leq s' \leq n \right\} \cup \\
&\quad \left\{ q_1^{(s)} \to \varepsilon \mid 1 \leq s \leq n \right\} .
\end{aligned}$$

It is routine to check that $\bigcup_{i=1}^{n} L_{q_1^{(i)}}(G^{\boldsymbol{p}}) = L(w_1^* \dots w_n^*)$.

Given $G^{\boldsymbol{p}}$ and a grammar $G = (\mathcal{X}, \Sigma, \mathcal{P}, S)$ in program normal form our goal is to define a grammar for the language

$$h^{-1}(L(G) \cap L(\boldsymbol{p})) \cap L(a_1^* \ldots a_n^*) \ .$$

We first define $G^{\bowtie} = (\mathcal{X}^{\bowtie}, \Sigma, \mathcal{P}^{\bowtie}, X_0)$ which, as we prove later, satisfies $L(G^{\bowtie}) = L(G) \cap L(\boldsymbol{p})$:

- $\mathcal{X}^{\bowtie} = \{X_0\} \cup \left\{[q_r^{(s)} X q_y^{(x)}] \mid X \in \mathcal{X}, q_r^{(s)}, q_y^{(x)} \in \mathcal{X}^{\boldsymbol{p}}, s \leq x\right\}$
- $\mathcal{P}^{\bowtie}$ is the set containing for every $1 \leq s \leq x \leq n$ a production $X_0 \to [q_1^{(s)} S q_1^{(x)}]$ and:
  - for every production $X \to \varepsilon \in \mathcal{P}$ of G and for every $1 \leq s \leq n, 1 \leq r \leq j_s$ a production
  $$[q_r^{(s)} X q_r^{(s)}] \to \varepsilon \ ; \quad (15)$$
  - for every production $X \to Y \in \mathcal{P}$ of G and for every $1 \leq s \leq u \leq n, 1 \leq r \leq j_s, 1 \leq v \leq j_u$ a production
  $$[q_r^{(s)} X q_v^{(u)}] \to [q_r^{(s)} Y q_v^{(u)}] \ ; \quad (16)$$
  - for every production $X \to \gamma \cdot Y \in \mathcal{P}$ of G with $\gamma \in \Sigma$ and for every $1 \leq s \leq u \leq n, 1 \leq r \leq j_s, 1 \leq v \leq j_u$ productions
  $$[q_r^{(s)} X q_v^{(u)}] \to \gamma \cdot [q_{r+1}^{(s)} Y q_v^{(u)}]$$
  $$\text{if } r < j_s \text{ and } \gamma = b_r^{(s)}; \text{ and} \quad (17)$$
  $$[q_{j_s}^{(s)} X q_v^{(u)}] \to \gamma \cdot [q_1^{(s')} Y q_v^{(u)}] \in \mathcal{P}^{\bowtie}$$
  $$\text{if } \gamma = b_{j_s}^{(s)} \text{ and } s \leq s' \leq u \quad (18)$$
  - for every production $X \to ZY \in \mathcal{P}$ of G and for every $1 \leq s \leq u \leq x \leq n, 1 \leq r \leq j_s, 1 \leq v \leq j_u, 1 \leq y \leq j_x$ a production
  $$[q_r^{(s)} X q_y^{(x)}] \to [q_r^{(s)} Z q_v^{(u)}][q_v^{(u)} Y q_y^{(x)}] \ . \quad (19)$$

In what follows, we use $X \underset{G}{\Rightarrow} \alpha$ to indicate that the derivation is carried out using the productions of the grammar G.

LEMMA 8. *Let $w \in \Sigma^*$. We have $[q_r^{(s)} X q_v^{(u)}] \underset{G^{\bowtie}}{\Rightarrow}^* w$ iff $q_r^{(s)} \underset{G^{\boldsymbol{p}}}{\Rightarrow}^* w \cdot q_v^{(u)}$ and $X \underset{G}{\Rightarrow}^* w$.*

PROOF: The proof for the only if direction is by induction on the length of the derivation of $[q_r^{(s)} X q_v^{(u)}] \Rightarrow^* w$.

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

$\mathbf{i > 1}$. We do a case analysis according to the definition of $G^{\bowtie}$.

- $[q_r^{(s)} X q_v^{(u)}] \Rightarrow [q_r^{(s)} Y q_v^{(u)}] \Rightarrow^* w$. It is trivially solved using the induction hypothesis.
- $[q_r^{(s)} X q_v^{(u)}] \Rightarrow \gamma \cdot [q_{r+1}^{(s)} Y q_v^{(u)}] \Rightarrow^* \gamma w'$. The definition of $G^{\bowtie}$ shows that $\gamma = b_r^{(s)}$ for some $r < j_s$. The production $q_r^{(s)} \to b_r^{(s)} \cdot q_{r+1}^{(s)} \in \delta^{\boldsymbol{p}}$ and induction hypothesis show that $q_r^{(s)} \Rightarrow b_r^{(s)} \cdot q_{r+1}^{(s)} \Rightarrow^* b_r^{(s)} w' \cdot q_v^{(u)}$. Also the production $X \to \gamma \cdot Y \in \mathcal{P}$ and induction hypothesis show that $X \Rightarrow \gamma \cdot Y \Rightarrow^* \gamma \cdot w'$ and we are done since $\gamma = b_r^{(s)}$.
- $[q_{j_s}^{(s)} X q_v^{(u)}] \Rightarrow b_{j_s}^{(s)} \cdot [q_1^{(s')} Y q_v^{(u)}] \Rightarrow^* b_{j_s}^{(s)} \cdot w'$. The production $q_{j_s}^{(s)} \to b_{j_s}^{(s)} \cdot q_1^{(s')} \in \delta^{\boldsymbol{p}}$ and induction hypothesis show that $q_{j_s}^{(s)} \Rightarrow b_{j_s}^{(s)} \cdot q_1^{(s')} \Rightarrow^* b_{j_s}^{(s)} w' \cdot q_v^{(u)}$. Also the production $X \to \gamma \cdot Y \in \mathcal{P}$ where $\gamma = b_{j_s}^{(s)}$ and the induction hypothesis show that $X \Rightarrow \gamma \cdot Y \Rightarrow^* \gamma \cdot w'$ and we are done.
- $[q_r^{(s)} X q_y^{(x)}] \Rightarrow [q_r^{(s)} Z q_v^{(u)}][q_v^{(u)} Y q_y^{(x)}] \Rightarrow^* w_1 \cdot [q_v^{(u)} Y q_y^{(x)}] \Rightarrow^* w_1 w_2 = w$. By induction hypothesis, we have $q_r^{(s)} \Rightarrow^* w_1 \cdot q_v^{(u)}$ and $Z \Rightarrow^* w_1$. Also $q_v^{(u)} \Rightarrow^* w_2 \cdot q_y^{(x)}$ and $Y \Rightarrow^* w_2$. Hence we find that $q_r^{(s)} \Rightarrow^* w_1 w_2 \cdot q_y^{(x)}$ and $X \Rightarrow^* w_1 w_2$ since $X \to Z \cdot Y \in \mathcal{P}$ which concludes this case since $w = w_1 w_2$.

Using a similar induction on the length of $X \Rightarrow^* w$, the "if" direction is easily proved. □

We now obtain a grammar G^f by slightly modifying $G^{\bowtie}$. We change the productions of (17) and (18) respectively to

$$[q_r^{(s)} X q_v^{(u)}] \to [q_{r+1}^{(s)} Y q_v^{(u)}] \text{ and } [q_{j_s}^{(s)} X q_v^{(u)}] \to a_s \cdot [q_1^{(s')} Y q_v^{(u)}]$$

where $a_s \in \widetilde{\Sigma}$. Because of this change, G^f has alphabet $\widetilde{\Sigma}$. Also, it is routine to check that this change amounts to applying the inverse homomorphism h^{-1} and taking the intersection with $L(a_1^* \ldots a_n^*)$.

A.2 Proof of Lemma. 1

Given $\boldsymbol{p} = w_1^* \ldots w_n^*$ and a grammar G in program normal form, we have that G^f satisfies each of the following properties:

- $L(G^f) \subseteq \widetilde{\Sigma}^*$ where $\widetilde{\Sigma} = \{a_1, \ldots, a_n\}$;
- $L(G^f) = h^{-1}(L(G) \cap L(\boldsymbol{p})) \cap L(a_1^* \ldots a_n^*)$;
- G^f is in program normal form;
- G^f is computable in polynomial time;
- If pr is the number of procedure variables in G, then G^f has $\mathcal{O}(pa^2 \cdot pr)$ procedure variables where pa is the size of $\boldsymbol{p}$.

PROOF: The first item is obvious from the definition of G^f. Let $G^f = (\mathcal{X}^f, \widetilde{\Sigma}, \mathcal{P}^f, X_0)$. If follows directly from Lem. 8 that for every $w \in \Sigma^*$, $X_0 \underset{G^{\bowtie}}{\Rightarrow}^* w$ iff $q_1^{(s)} \underset{G^{\boldsymbol{p}}}{\Rightarrow}^* w \cdot q_1^{(x)}$ where $1 \leq s \leq x \leq n$ and $S \underset{G}{\Rightarrow}^* w$. The productions $\left\{q_1^{(s)} \to \varepsilon \mid 1 \leq s \leq n\right\} \subseteq \delta^{\boldsymbol{p}}$ of $G^{\boldsymbol{p}}$ shows that the equivalence can be rewritten as follows: $X_0 \underset{G^{\bowtie}}{\Rightarrow}^* w$ iff $q_1^{(s)} \underset{G^{\boldsymbol{p}}}{\Rightarrow}^* w$ where $1 \leq s \leq n$ and $S \underset{G}{\Rightarrow}^* w$. We conclude from the definition of $G^{\boldsymbol{p}}$ that $L(\boldsymbol{p}) = \bigcup_{i=1}^n L_{q_1^{(i)}}(G^{\boldsymbol{p}})$, hence that $L(G^{\bowtie}) = L(G) \cap L(\boldsymbol{p})$, and finally that $L(G^f) = h^{-1}(L(G^{\bowtie})) \cap L(a_1^* \ldots a_n^*)$.

The second item is immediate from the definition of G^f and that G is in program normal form. The third item is clear from the definition of $G^{\bowtie}$ and G^f. For the fourth item, we have that since pa is the size of $\boldsymbol{p}$, pa is also the size of $\mathcal{X}^{\boldsymbol{p}}$ by definition of $G^{\boldsymbol{p}}$. Also (19) shows that each procedure variable in G yields $\mathcal{O}(pa^2)$ procedure variables in G, hence if pr is the number of procedure variables in G we find that the number of procedure variables in $G^{\bowtie}$, hence G^f is $\mathcal{O}(pa^2 \cdot pr)$. □