Solving String Constraints: The Case for Constraint Programming

Jun He[1,2], Pierre Flener[1], Justin Pearson[1], and Wei Ming Zhang[2]

[1] Uppsala University, Department of Information Technology, Uppsala, Sweden
[2] National University of Defense Technology,
School of Information System and Management, Changsha, Hunan, China
{Jun.He,Pierre.Flener,Justin.Pearson}@it.uu.se, wmzhang@nudt.edu.cn

Abstract. We improve an existing propagator for the context-free grammar constraint and demonstrate experimentally the practicality of the resulting propagator. The underlying technique could be applied to other existing propagators for this constraint. We argue that constraint programming solvers are more suitable than existing solvers for verification tools that have to solve string constraints, as they have a rich tradition of constraints for membership in formal languages.

1 Introduction

For constraint programming (CP) languages, user-level extensibility has been an important goal for over a decade. Global constraints for formal languages are promising for this purpose. The Regular constraint [16] requires a sequence of decision variables to belong to a regular language, specified by a deterministic finite automaton (DFA) or a regular expression; the Automaton constraint [2] takes a DFA with counters. The CFG constraint [17,20] requires a sequence of decision variables to belong to a context-free language, specified by a context-free grammar (CFG). For many applications, the length n of a sequence constrained to belong to some formal language is known in advance. Since every fixed-size language is finite and hence regular, the need for a CFG constraint in such applications depends on the grammar and the complexities of the propagators. It takes $O(n\,|A|)$ time to achieve generalised arc consistency (GAC) for a Regular constraint with an automaton A, but $O(n^3\,|G|)$ time for a CFG constraint with a grammar G. In [12], the authors introduce a reformulation of a grammar into an automaton for a fixed length n, and show that this reformulation is preferable if the resulting automaton is not huge. However, their reformulation itself needs a CFG propagator to achieve domain consistency at the root of the search tree so that the resulting automaton is smaller. In [7], the authors introduce a forklift scheduling problem, where there is no tractable reformulation of a grammar into an automaton as the size of the resulting automaton is exponential in n. Hence, a CFG propagator is necessary in this case. To the best of our knowledge, no CP solver includes the CFG constraint.

In the analysis, testing, and verification of string-manipulating programs, constraints on sequences (strings) of decision variables arise. Kieżun *et al.* [14] argue

C. Schulte (Ed.): CP 2013, LNCS 8124, pp. 381–397, 2013.

that custom string solvers should not be designed any more, for sustainability reasons, since powerful off-the-shelf solvers are available: their tool, Hampi, translates a Regular or CFG constraint on a fixed-size string into bit-vector constraints so as to solve them using the SMT solver STP [6], much more efficiently than three custom tools and even up to three orders of magnitude faster than the SAT-based CFGAnalyzer tool [1]. The solver Kaluza [19] handles constraints over multiple string variables, unlike the restriction of Hampi to one such variable, and it also generates bit-vector constraints that are passed to STP. Fu *et al.* [5] argue that it is important to model regular replacement operations, which are not supported by Hampi and Kaluza, and introduce the custom string solver Sushi, which models string constraints via automata instead of a bit-vector encoding. So the question arises whether the formal language constraints of CP are competitive with Hampi, Kaluza, and Sushi.

In this paper, we revisit the CFG constraint and make the following *contributions*:

- We improve the CFG propagator of [11], which improves the one of [20], by exploiting an idea of [14] for reformulating a grammar into a regular expression for a fixed string length. We conjecture that this idea also applies to the CFG propagators of [7,13,17,18]. (Section 3)
- We implement our CFG propagator for the Gecode [8] open-source CP solver, and demonstrate experimentally its practicality. (Sections 4.1 to 4.3)
- We show that the CP solver Gecode with our CFG propagator (or even its ancestor [11]) systematically beats Hampi and Kaluza, by up to four orders of magnitude, on Hampi's benchmark (Section 4.3). We show that Gecode with the built-in Regular propagator systematically beats Kaluza and Sushi, by a factor up to 130, on Sushi's benchmark (Section 4.4).

2 Background

We first give some background material on grammars (e.g., see [10]).

2.1 Context-Free Grammars

A *CFG* is a tuple $\langle \Sigma, N, P, S \rangle$, where Σ is the alphabet and any value $v \in \Sigma$ is called a terminal, N is the finite set of non-terminals, $P \subseteq N \times (\Sigma \cup N)^*$ is the finite set of productions, and $S \in N$ is the start non-terminal. A CFG is said to be in *Chomsky normal form* (CNF) iff $P \subseteq N \times \left(\Sigma \cup N^2\right)$. Every CFG can be converted into an equivalent grammar in CNF.

Example 1. Consider the CFG $G_B = \langle \Sigma, N, P, S \rangle$, where $\Sigma = \{\ell, r\}$, $N = \{S\}$, and $P = \{S \to \ell r,\ S \to SS,\ S \to \ell S r\}$. It defines a language of correctly bracketed expressions (e.g., $\ell r \ell r$ and $\ell \ell r r$), with 'ℓ' denoting the left bracket and 'r' the right one. Its CNF is $G'_B = \langle \Sigma, N', P', S \rangle$, where $N' = \{L, M, R, S\}$ and $P' = \{S \to LR,\ S \to SS,\ S \to MR,\ M \to LS,\ L \to \ell,\ R \to r\}$.

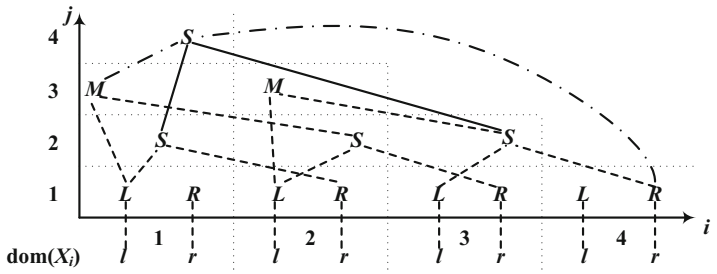

Fig. 1. The CYK-based propagator parses a sequence $\langle X_1, \ldots, X_4 \rangle$ of $n = 4$ decision variables with the same domain $\{\ell, r\}$ under the CFG G'_B of Example 1

The Cocke-Younger-Kasami (CYK) algorithm is a parser for CFGs in CNF. We describe it for a sequence of decision variables instead of values. Given a CFG $\langle \Sigma, N, P, S \rangle$ in CNF and a sequence $\langle X_1, \ldots, X_n \rangle$ of n decision variables, the CYK parser computes a table V, where $V_{i,j}$ (with $1 \leq j \leq n$ and $1 \leq i \leq n+1-j$) is the set of non-terminals (or at most the start non-terminal S for $i = 1$ and $j = n$) that can be parsed using a sequence of j values in the domains of X_i to X_{i+j-1} respectively, using dynamic programming:

$$V_{i,j} = \begin{cases} \{W \mid (W \to b) \in P \ \wedge \ b \in \mathrm{dom}\,(X_i)\} & \text{if } j = 1 \\ \bigcup_{k=1}^{j-1} \left\{ W \;\middle|\; \begin{array}{c} (W \to YZ) \in P \ \wedge \ (j < n \ \vee \ W = S) \\ \wedge \ Y \in V_{i,k} \ \wedge \ Z \in V_{i+k,j-k} \end{array} \right\} & \text{otherwise} \end{cases}$$

For example, Figure 1 gives the CYK table V when parsing a sequence X of 4 decision variables with the same domain $\{\ell, r\}$ under the grammar G'_B of Example 1. We have $V_{1,1} = \{L, R\}$ and $V_{1,4} = \{S\}$. Note that we use $\mathrm{dom}\,(X_i)$ to denote the domain of the decision variable X_i.

Given a word $w \in \Sigma^n$, let w_i (with $1 \leq i \leq n$) denote the letter at position i of w. If all decision variables X_i have $\mathrm{dom}\,(X_i) = \{w_i\}$, then w is *accepted* by G iff $V_{1,n} = \{S\}$.

2.2 The CFG Constraint

The CFG constraint is defined as $\mathrm{CFG}(X, G)$, where X is a sequence of decision variables and G is a grammar. An assignment w to X is a solution iff w is a word accepted by G.

Given a CFG $G = \langle \Sigma, N, P, S \rangle$ in CNF and a sequence X of n variables, let $|G| = \sum_{p \in P} |p|$ be the size of G, and $|p|$ the number of (non-)terminals in the production p. The propagator of [11] achieves GAC for the $\mathrm{CFG}(X, G)$ constraint in $O\left(n^3 |G|\right)$ time with $O\left(n^2 |G|\right)$ space, which is better than the propositional satisfiability (SAT) based propagator of [18], which decomposes and achieves GAC for the CFG constraint in $O\left(n^3 |G|\right)$ time and space. More recently, another SAT-based propagator is introduced in [7], which works similarly to the propagator of [11] and outperforms the propagator of [18].

In this paper, we use the propagator of [11] *as an example* to show how to improve a CFG propagator. We conjecture that the same idea can be used to improve the propagators of [7,13,17,18].

To describe elegantly the propagator of [11] and ours (given in Section 3), we first introduce a novel concept. Informally, given a non-terminal W in $V_{i,j}$ of the CYK table, a *low support* for this W, namely $(W \to YZ,\ k)$, denotes that two non-terminals lower down in V, namely Y in $V_{i,k}$ and Z in $V_{i+k,j-k}$, support the existence of W in $V_{i,j}$; and this low support corresponds to two *high support*s, namely $(W \to YZ,\ j)$ of Y in $V_{i,k}$ and Z in $V_{i+k,j-k}$. Formally:

Definition 1 (Support). *For any* $1 < j \leq n$, $1 \leq i \leq n+1-j$, *and non-terminal* W *in* $V_{i,j}$ *of the CYK table, the set* $\overline{\mathrm{LS}}_{i,j}(W) = \{(W \to YZ,\ k) \mid (W \to YZ) \in P \ \wedge\ 0 < k < j\}$ *is called the* candidate low-support set *for* W *in* $V_{i,j}$. *The set* $\mathrm{LS}_{i,j}(W) = \{(W \to YZ,\ k) \in \overline{\mathrm{LS}}_{i,j}(W) \mid Y \in V_{i,k} \ \wedge\ Z \in V_{i+k,j-k}\}$ *is called the* low-support set *for* W *in* $V_{i,j}$. *For* $j = 1$ *and any* $1 \leq i \leq n$ *and non-terminal* W *in* $V_{i,1}$, *we define* $\overline{\mathrm{LS}}_{i,1}(W) = \{(W \to b) \in P\}$ *and* $\mathrm{LS}_{i,1}(W) = \{(W \to b) \in \overline{\mathrm{LS}}_{i,1}(W) \mid b \in \mathrm{dom}(X_i)\}$.

For any $1 \leq j < n$, $1 \leq i \leq n+1-j$, *and non-terminal* W *in* $V_{i,j}$ *of the CYK table, the set* $\overline{\mathrm{HS}}_{i,j}(W) = \{(Y \to QZ,\ k) \mid (Y \to QZ) \in P \wedge (W = Q \ \vee\ W = Z) \wedge j < k \leq n\}$ *is called the* candidate high-support set *of* W *in* $V_{i,j}$. *The set* $\mathrm{HS}_{i,j}(W) = \{(Y \to QZ,\ k) \in \overline{\mathrm{HS}}_{i,j}(W) \mid (W = Q \wedge Y \in V_{i,k} \wedge Z \in V_{i+j,k-j}) \vee (W = Z \wedge Y \in V_{i-j,k} \wedge Q \in V_{i-j,k-j})\}$ *is called the* high-support set *of* W *in* $V_{i,j}$. *For any* $1 \leq i \leq n$ *and value* b *in* $\mathrm{dom}(X_i)$, *we define* $\overline{\mathrm{HS}}_i(b) = \{(W \to b) \in P\}$ *and* $\mathrm{HS}_i(b) = \{(W \to b) \in \overline{\mathrm{HS}}_i(b) \mid b \in \mathrm{dom}(X_i)\}$. □

For example, in the CYK table V of Figure 1, $\overline{\mathrm{LS}}_{1,4}(S) = \{S \to LR, S \to SS, S \to MR\} \times \{1,2,3\}$ has 9 candidate low supports; only 2 thereof are low supports for non-terminal S in $V_{1,4}$, namely $(S \to SS,\ 2)$ (depicted by the solid arcs), and $(S \to MR,\ 3)$ (depicted by the dash-dotted arcs). The low support $(S \to SS,\ 2)$ for S in $V_{1,4}$ denotes that it is supported by S in $V_{1,2}$ and $V_{3,2}$, hence the low support corresponds to 2 high supports, namely $(S \to SS,\ 4)$ of S in $V_{1,2}$ and $V_{3,2}$.

The propagator of [11] achieves GAC for the $\mathrm{CFG}(X, G)$ constraint as follows: (1) The CYK parser computes the table V. (2) A bottom-up process finds the *first* low support in every $\overline{\mathrm{LS}}_{i,j}(W)$. A top-down process finds the *first* high support in every $\overline{\mathrm{HS}}_{i,j}(W)$. All non-terminals W with no support are removed from V. (3) The *first* high support in every $\overline{\mathrm{HS}}_i(b)$ is found, and all values b in any $\mathrm{dom}(X_i)$ with no high support are removed from $\mathrm{dom}(X_i)$. When a support is found in steps 2 and 3, its position in the candidate support set is recorded. When a support is lost as the domains shrink, the next support is to be found starting *after* the previous support in the candidate support set. The propagator is incremental, and explores all candidate supports at most once.

3 An Improved Propagator

Inspired by [14], we present, verify, and analyse an improved version of the propagator of [11] for the CFG constraint.

3.1 Motivation and Theoretical Foundation

There are two dependent opportunities for improving the propagator of [11].

Encoding the Support Sets Space-Efficiently. The propagator of [11] explores *all* candidate supports once in the worst case, hence its time complexity is bounded by $\left|\overline{\mathrm{LS}}\right| + \left|\overline{\mathrm{HS}}\right| = \sum_{j=1}^{n} \sum_{i=1}^{n+1-j} \sum_{W \in V_{i,j}} \left|\overline{\mathrm{LS}}_{i,j}(W)\right| + \left|\overline{\mathrm{HS}}_{i,j}(W)\right| = O\left(n^3 |G|\right)$. If we can make the propagator run on the small support sets instead of the large candidate support sets, then the propagator probably runs faster. Consider that $\overline{\mathrm{LS}}_{i,j}(W) \supseteq \mathrm{LS}_{i,j}(W)$ and $\overline{\mathrm{HS}}_{i,j}(W) \supseteq \mathrm{HS}_{i,j}(W)$ (from Definition 1), and that the gaps may be huge. For example in Figure 1, $\mathrm{LS}_{1,4}(S) = \{(S \to SS,\ 2), (S \to MR,\ 3)\}$ is of size 2, while $\overline{\mathrm{LS}}_{1,4}(S) = \{S \to LR, S \to SS, S \to MR\} \times \{1, 2, 3\}$ is of size 9; $\mathrm{HS}_{2,1}(R) = \{(S \to LR,\ 2)\}$ is of size 1, while $\overline{\mathrm{HS}}_{2,1}(R) = \{S \to LR,\ S \to MR\} \times \{2, 3, 4\}$ is of size 6. However, the challenge is to avoid having to pay with space what we save in time.

Given a CFG $G = \langle \Sigma, N, P, S \rangle$ in CNF and n decision variables, Kadıoğlu and Sellmann [11] claim that storing all support sets takes $O\left(n^3 |G|\right)$ space, which is expensive. Their propagator thus runs on the large *candidate* support sets, which can be encoded very space-efficiently. Two sets $\mathrm{Out}(W) = \{(W \to YZ) \in P\}$ and $\mathrm{In}(W) = \{(Y \to QZ) \in P \mid W = Q \vee W = Z\}$ are computed for any $W \in N$, so that $\overline{\mathrm{LS}}_{i,j}(W) = \mathrm{Out}(W) \times \{1, \ldots, j-1\}$ and $\overline{\mathrm{HS}}_{i,j}(W) = \mathrm{In}(W) \times \{j+1, \ldots, n\}$. For any j, the sets $\{1, \ldots, j-1\}$ and $\{j+1, \ldots, n\}$ need not be stored. Hence encoding all candidate support sets only takes $O(|G|)$ space by storing all $\mathrm{Out}(W)$ and $\mathrm{In}(W)$. As it takes $O\left(n^2 |G|\right)$ space to store the CYK table V, the overall space complexity is $O\left(n^2 |G|\right)$.

However, we *can* decrease the space requirement for encoding all low-support sets and a superset of all high-support sets (given in Theorem 2 below) from $O\left(n^3 |G|\right)$ to $O\left(n^2 |G|\right)$, which is the *same* as the one needed to store the CYK table V, by using an idea of [14] for reformulating a grammar into a regular expression for a fixed string length n. In that reformulation, a regular expression is obtained by using the *same* domains: $\mathrm{dom}(X_i) = \Sigma$ for *all* $1 \leq i \leq n$. A regular expression $E_{1,j}$ for the sub-sequence $\langle X_1, \ldots, X_j \rangle$ is computed and stored as a template for every $1 \leq j \leq n$, and then the regular expression $E_{i,j}$ for the sub-sequence $\langle X_i, \ldots, X_{i+j-1} \rangle$ turns out to be equal to $E_{1,j}$ for every $1 < i \leq n+1-j$. Similarly, in Figure 1, we find that $V_{i,j} = V_{1,j}$ and every non-terminal in $V_{i,j}$ has the the same low supports as in $V_{1,j}$. For example, $V_{3,2} = V_{2,2} = V_{1,2} = \{S\}$ and $\mathrm{LS}_{3,2}(S) = \mathrm{LS}_{2,2}(S) = \mathrm{LS}_{1,2}(S) = \{(S \to LR,\ 1)\}$. Based on this observation, we give the following theorem (we show in Section 3.2 how to lift the same-domain restriction):

Theorem 1. *Given a CFG $G = \langle \Sigma, N, P, S \rangle$ in CNF and a sequence $\langle X_1, \ldots, X_n \rangle$ of n decision variables, if all X_i have the same domain, then for any $1 \leq j \leq n$ and $1 < i \leq n+1-j$:*

1. $V_{i,j} = V_{1,j}$
2. $\forall W \in V_{i,j} :\ \mathrm{LS}_{i,j}(W) = \mathrm{LS}_{1,j}(W)$

Proof: We prove claim 1 by complete induction on j.

(Base: $j = 1$) For any non-terminal W, we have $W \in V_{i,1}$ iff there exists a production $(W \to b) \in P$ such that $b \in \mathrm{dom}(X_i)$. As $\mathrm{dom}(X_i) = \mathrm{dom}(X_1)$, we have $W \in V_{i,1}$ iff $W \in V_{1,1}$.

(Step: $1 < j \leq n$) For any $1 \leq j' < j$, the induction hypothesis is $V_{i,j'} = V_{1,j'}$ for any $1 < i$. We want to prove $V_{i,j} = V_{1,j}$ for any $1 < i \leq n + 1 - j$. For any non-terminal W, we have $W \in V_{i,j}$ iff there exists a production $(W \to YZ) \in P$ and $1 \leq k < j$ such that $Y \in V_{i,k}$ and $Z \in V_{i+k,j-k}$. As $V_{i,k} = V_{1,k}$ and $V_{i+k,j-k} = V_{1,j-k} = V_{1+k,j-k}$, we have $W \in V_{i,j}$ iff $W \in V_{1,j}$.

Using this, claim 2 follows from Definition 1. □

The next theorem enables a space-efficient encoding of the support sets (again, we show in Section 3.2 how to lift the same-domain restriction).

Theorem 2. *Given a CFG $G = \langle \Sigma, N, P, S \rangle$ in CNF and a sequence $\langle X_1, \ldots, X_n \rangle$ of n decision variables, if all X_i have the same domain, then it takes $O\left(n^2 |G|\right)$ space to encode the CYK table V and all support sets.*

Proof: For any $1 \leq j \leq n$ and $1 < i \leq n + 1 - j$:

By Theorem 1, we have $V_{i,j} = V_{1,j}$. Hence we obtain the whole CYK table V by storing all $V_{1,j}$ in $\sum_{j=1}^{n} |V_{1,j}| = O(n|N|) = O(n|G|)$ space, as $|G| = \sum_{p \in P} |p| > |N|$.

By Theorem 1, we have $\mathrm{LS}_{i,j}(W) = \mathrm{LS}_{1,j}(W)$. Hence we obtain all low supports by storing all $\mathrm{LS}_{1,j}(W)$ in $\sum_{j=1}^{n} \sum_{W \in V_{1,j}} |\mathrm{LS}_{1,j}(W)| \leq \sum_{j=1}^{n} |P \times \{k \mid 1 \leq k < j\}|$ $= \sum_{j=1}^{n} O(n|G|) = O\left(n^2 |G|\right)$ space, as $|G| = \sum_{p \in P} |p| > |P|$ and each low support takes constant space.

Considering the high-support set $\mathrm{HS}_{i,j}(W)$, it takes $O\left(n^3 |G|\right)$ space to store all $\mathrm{HS}_{i,j}(W)$ as $\mathrm{HS}_{i,j}(W) = \mathrm{HS}_{1,j}(W)$ is not true for all $1 \leq j \leq n$ and $i > 1$. For example in Figure 1, we have $\mathrm{HS}_{2,1}(R) = \{(S \to LR,\ 2)\}$, while $\mathrm{HS}_{1,1}(R) = \emptyset$. To save space, we compute the set $\mathrm{HS}'_{i,j}(W) = \bigcup_{k=1}^{n+1-j} \mathrm{HS}_{k,j}(W)$ instead of $\mathrm{HS}_{i,j}(W)$, as we can encode $\mathrm{HS}'_{i,j}(W)$ efficiently. Note that we still have $\mathrm{HS}'_{i,j}(W) \subseteq \overline{\mathrm{HS}}_{i,j}(W)$ as $\overline{\mathrm{HS}}_{i,j}(W) = \overline{\mathrm{HS}}_{1,j}(W)$ (its formulation in Definition 1 is independent of i) and $\mathrm{HS}'_{i,j}(W) = \bigcup_{k=1}^{n+1-j} \mathrm{HS}_{k,j}(W) \subseteq \bigcup_{k=1}^{n+1-j} \overline{\mathrm{HS}}_{k,j}(W) = \overline{\mathrm{HS}}_{1,j}(W)$. Hence we obtain all $\mathrm{HS}'_{i,j}(W)$ by computing and storing all $\mathrm{HS}'_{1,j}(W)$ in $O\left(n^2 |G|\right)$ space, as $\mathrm{HS}'_{i,j}(W) = \mathrm{HS}'_{1,j}(W)$ and $\sum_{j=1}^{n} \sum_{W \in V_{1,j}} \left|\mathrm{HS}'_{1,j}(W)\right| \leq$ $2 \sum_{j=1}^{n} \sum_{W \in V_{1,j}} |\mathrm{LS}_{1,j}(W)| = O\left(n^2 |G|\right)$ (the definition of $\mathrm{HS}'_{i,j}(W)$ is independent of i and one low support corresponds to at most two high supports).

Hence we can encode the CYK table V, all $\mathrm{LS}_{i,j}(W)$, and all $\mathrm{HS}'_{i,j}(W)$ in $O\left(n^2 |G|\right)$ space. □

Using Theorem 2, it *is* practical to make the propagator run on $\mathrm{LS}_{i,j}(W)$ and $\mathrm{HS}'_{i,j}(W)$, which are subsets of the candidate support sets, with $O\left(n^2 |G|\right)$ space. Although Theorem 2 requires all $\mathrm{dom}(X_i)$ to be the *same*, this is not an obstacle in practice, as shown in Section 3.2 below. Note that $|\mathrm{LS}| + \left|\mathrm{HS}'\right|$ and $\left|\overline{\mathrm{LS}}\right| + \left|\overline{\mathrm{HS}}\right|$ are asymptotically the same (as shown in Section 3.3 below), hence we cannot improve the propagator of [11] asymptotically.

Counting the Supports. For each non-terminal W in the CYK table, the propagator of [11], which is based on the arc-consistency (AC) algorithm AC-6 [3], decides whether W has low and high supports by exhibiting two actual supports (one low and one high). However, this is not necessary. We can simply *count* the supports for W as in AC-4 [15], and then just decrease the counter by one when a support is lost. Although Bessière [3] shows that AC-4 is worse than AC-6 for *binary* CSPs given *extensionally* because initialising the counters is expensive, in our case initialisation is much cheaper because we have $|\mathrm{LS}_{i,j}(W)| = |\mathrm{LS}_{1,j}(W)|$ initially when using our efficient encoding of the support sets. However, by using counters, we do not need complex data structures and operations to trace which non-terminal in the CYK table is currently supporting and supported by which non-terminal(s), as in [11]. Indeed, our experiments (omitted for space reasons, see Appendix C of [9]) show that counting with our efficient encoding of the support sets works better (up to 12 times) than using *only* the latter, which already works better (up to 20 times) than the propagator of [11].

3.2 Description and Proof of Our Propagator

Consider a CFG $G = \langle \Sigma, N, P, S \rangle$ in CNF and a sequence $X = \langle X_1, \ldots, X_n \rangle$ of n decision variables. We introduce a propagator for the $\mathrm{CFG}(X, G)$ constraint using the AC-4 framework, which computes all supports and counts them when *posting* the constraint (see Algorithm 1), and then *only* decreases the support counters during propagation (see Algorithm 2), without changing the support sets. Hence, to satisfy the condition of Theorem 2, we *only* need to make all decision variables *temporarily* take the same domain when *posting* the constraint. Our propagator has *no* limitation on the initial domains of the decision variables, as we will show how our propagator lifts the temporary restriction at no asymptotic overhead.

Let $C^{\mathrm{LS}}_{i,j}(W)$ (or $C^{\mathrm{HS}}_{i,j}(W)$) denote the number of low (or high) supports for (or of) a non-terminal W in $V_{i,j}$ of the CYK table during propagation. Similarly, let $C^{\mathrm{LS}}_{i}(b)$ (or $C^{\mathrm{HS}}_{i}(b)$) denote the number of low (or high) supports for (or of) a terminal b in $\mathrm{dom}(X_i)$. Note that every (non-)terminal has two counters and there is no sharing of counters between any two (non-)terminals, as the counters will be changed *independently* during propagation. Using Theorem 2, Algorithm 1 posts the $\mathrm{CFG}(X, G)$ constraint, encodes the CYK table

and support sets, counts the supports, and achieves GAC. Given all propagator state variables, which are also shared by Algorithm 2, initialised so that $V_{1,j} = \mathrm{LS}_{1,j}(W) = \mathrm{HS}'_{1,j}(W) = \emptyset$ and $C^{\mathrm{HS}}_{i,j}(W) = 0$ (lines 2 to 4), Algorithm 1 works as follows. First, it constructs a virtual domain $\mathrm{Dom}' = \bigcup_{i=1}^{n} \mathrm{dom}(X_i)$ (line 5), and uses it to post the $\mathrm{CFG}(X, G)$ constraint, hence the condition of Theorem 2 is satisfied as all domains are now the same. Using the virtual domain may introduce extra solutions, and we show in the last step how to avoid this. Second, it uses a bottom-up process (lines 6 to 17) based on the CYK parser to compute all $V_{1,j}$, $\mathrm{LS}_{1,j}(W)$, $\mathrm{HS}'_{1,j}(W)$, and $C^{\mathrm{LS}}_{i,j}(W)$. Note that we only need to compute $V_{1,j}$ by Theorem 1, and any reference to $V_{i,j}$ is replaced by $V_{1,j}$. The same holds for $\mathrm{LS}_{i,j}(W)$, and $\mathrm{HS}'_{i,j}(W)$ (by its definition independently of i in Theorem 2). If the start non-terminal S is not in $V_{1,n}$, then it fails (line 18; no word from the current domains is accepted by G, hence no solution exists). Third, it uses a top-down process (lines 19 to 25) to compute all $C^{\mathrm{HS}}_{i,j}(W)$. Fourth, it removes all values with no high support from the domains (lines 26 to 28). Finally, it constructs a set Δ of all variable-value pairs that are not in the domains of X but in the virtual domain (line 29), and calls the function filterFromUpdate (in Algorithm 2, discussed next) to re-establish GAC after removing all such variable-value pairs (line 30). Hence the side effect of using the virtual domain is lifted; we show in Section 3.3 that calling the function filterFromUpdate does not increase the asymptotic complexity of Algorithm 1.

Given a set Δ of all recently filtered variable-value pairs by other propagators or a branching of the search tree, the function filterFromUpdate in Algorithm 2 incrementally re-establishes GAC for the $\mathrm{CFG}(X, G)$ constraint as follows. First, it creates two arrays Q_{LS} and Q_{HS} of initially empty queues (line 2), with $Q_{\mathrm{LS}}[j]$ (or $Q_{\mathrm{HS}}[j]$) storing all non-terminals W in the j-th row of the CYK table with no low (or high) supports due to the domain changes Δ. Second, it iterates over all removed values in Δ, decreasing the counter $C^{\mathrm{LS}}_{i,1}(W)$ for all non-terminals W in the bottom row supported by a removed value, and adding all W with no low support to the queue $Q_{\mathrm{LS}}[1]$ (lines 3 to 7). Third, a bottom-up process (lines 8 to 11) calls the procedure rmNoLS handling all W in the queue $Q_{\mathrm{LS}}[j]$. Given a non-terminal W with no low support, rmNoLS iterates over each high support of W, decreasing the three counters related with this lost high support, and enqueuing $Q_{\mathrm{LS}}[j]$ (or $Q_{\mathrm{HS}}[j]$) whenever a low (or high) support counter is zero (lines 22 to 33). Fourth, a top-down process (lines 12 to 14) calls the procedure rmNoHS (omitted for space reasons, see Appendix C of [9]), which works similarly to rmNoLS, handling all W in the queue $Q_{\mathrm{HS}}[j]$. Finally, it removes inconsistent values (with no high support) from the domains of X (lines 15 to 20), and reaches a fixpoint (line 21). Note that Algorithm 2 is a direct usage of the AC-4 framework. Once Algorithm 1 initialises the support sets and counters correctly, the correctness of Algorithm 2 is guaranteed by the AC-4 framework.

Theorem 3. *Our propagator achieves GAC for* $\mathrm{CFG}(X, G)$.

Proof: A value is removed by our propagator from the domains of X iff it has no high supports, as with the propagator of [11]. Hence the two propagators are equivalent. The result follows from Theorem 2 on page 132 of [11]. □

Algorithm 1. An improved propagator for the $\mathrm{CFG}(X, G)$ constraint, where $X = \langle X_1, \ldots, X_n \rangle$ is a sequence of n decision variables and $G = \langle \Sigma, N, P, S \rangle$ is a CFG in CNF

1: **function** post($\mathrm{CFG}(X, G)$)
2: **for all** $W \in N$ **and** $j \leftarrow 1$ **to** n **do**
3: $V_{1,j} \leftarrow \mathrm{LS}_{1,j}(W) \leftarrow \mathrm{HS}'_{1,j}(W) \leftarrow \emptyset$
4: **for all** $i \leftarrow 1$ **to** $n + 1 - j$ **do** $C^{\mathrm{HS}}_{i,j}(W) \leftarrow 0$
5: $\mathrm{Dom}' \leftarrow \bigcup_{i=1}^{n} \mathrm{dom}(X_i)$
6: $V_{1,1} \leftarrow \{W \mid (W \to b) \in P \ \wedge \ b \in \mathrm{Dom}'\}$
7: $\mathrm{LS}_{1,1}(W) \leftarrow \{W \to b \mid (W \to b) \in P \ \wedge \ b \in \mathrm{Dom}'\}$
8: $\mathrm{HS}'_1(b) \leftarrow \{W \to b \mid (W \to b) \in P \ \wedge \ b \in \mathrm{Dom}'\}$
9: **for all** $j \leftarrow 2$ **to** n **do**
10: **for all** $(W \to YZ) \in P$ **and** $k \leftarrow 1$ **to** $j - 1$ **do**
11: **if** $Y \in V_{1,k} \wedge Z \in V_{1,j-k} \wedge (j < n \vee W = S)$ **then**
12: $V_{1,j} \leftarrow V_{1,j} \cup \{W\}$
13: $\mathrm{LS}_{1,j}(W) \leftarrow \mathrm{LS}_{1,j}(W) \cup \{(W \to YZ,\ k)\}$
14: $\mathrm{HS}'_{1,k}(Y) \leftarrow \mathrm{HS}'_{1,k}(Y) \cup \{(W \to YZ,\ j)\}$
15: $\mathrm{HS}'_{1,j-k}(Z) \leftarrow \mathrm{HS}'_{1,j-k}(Z) \cup \{(W \to YZ,\ j)\}$
16: **for all** $j \leftarrow 1$ **to** n **and** $W \in V_{1,j}$ **do**
17: **for all** $i \leftarrow 1$ **to** $n + 1 - j$ **do** $C^{\mathrm{LS}}_{i,j}(W) \leftarrow |\mathrm{LS}_{1,j}(W)|$
18: **if** $S \notin V_{1,n}$ **then return** *failed*
19: **for all** $j \leftarrow n$ **to** $2,\ W \in V_{1,j},$ **and** $i \leftarrow 1$ **to** $n + 1 - j$ **do**
20: **if** $C^{\mathrm{HS}}_{i,j}(W) > 0 \vee j = n$ **then**
21: **for all** $(W \to YZ,\ k) \in \mathrm{LS}_{1,j}(W)$ **do**
22: $C^{\mathrm{HS}}_{i,k}(Y)\ {+}{+};\ C^{\mathrm{HS}}_{i+k,j-k}(Z)\ {+}{+}$
23: **for all** $W \in V_{1,1}$ **and** $i \leftarrow 1$ **to** n **do**
24: **if** $C^{\mathrm{HS}}_{i,1}(W) > 0$ **then**
25: **for all** $(W \to b) \in \mathrm{LS}_{1,1}(W)$ **do** $C^{\mathrm{HS}}_{i}(b)\ {+}{+}$
26: **for all** $i \leftarrow 1$ **to** n **and** $b \in \mathrm{dom}(X_i)$ **do**
27: **if** $C^{\mathrm{HS}}_{i}(b) = 0$ **then** $\mathrm{dom}(X_i) \leftarrow \mathrm{dom}(X_i) \setminus \{b\}$
28: **if** $\mathrm{dom}(X_i) = \emptyset$ **then return** *failed*
29: $\Delta \leftarrow \{(X_i, b) \mid X_i \in X \wedge b \in \mathrm{Dom}' \setminus \mathrm{dom}(X_i)\}$
30: **return** filterFromUpdate($\mathrm{CFG}(X, G), \Delta$)

3.3 Complexity Analysis

We first investigate the worst-case *time* complexity of our propagator for the $\mathrm{CFG}(X, G)$ constraint. In Algorithm 1, the time complexity of lines 2 to 29 is dominated by lines 19 to 25, which explore at most all low-support sets $\mathrm{LS}_{i,j}(W)$ (referenced as $\mathrm{LS}_{1,j}(W)$) once in $\sum_{j=1}^{n} \sum_{i=1}^{n+1-j} \sum_{W \in V_{1,j}} |\mathrm{LS}_{1,j}(W)| < n \sum_{j=1}^{n} \sum_{W \in V_{1,j}} |\mathrm{LS}_{1,j}(W)| = O\left(n^3 |G|\right)$ time, by Theorem 2; line 30 calls the function filterFromUpdate in Algorithm 2, which explores once all $\mathrm{LS}_{i,j}(W)$

Algorithm 2. Given a set Δ of domain changes, the function filterFromUpdate incrementally re-establishes GAC for the $\text{CFG}(X, G)$ constraint on a sequence $X = \langle X_1, \ldots, X_n \rangle$ of n decision variables.

```
 1: function filterFromUpdate(CFG(X, G), Δ)
 2: for all j ← 1 to n do Q_LS[j] ← [ ]; Q_HS[j] ← [ ]
 3: for all (X_i, b) ∈ Δ do
 4:    C_i^HS (b) ← 0
 5:    for all (W → b) ∈ HS'_1 (b) do
 6:       if C_{i,1}^LS (W) > 0 then
 7:          if −−C_{i,1}^LS (W) = 0 then Q_LS[1].enqueue((W, i))
 8: for all j ← 1 to n do
 9:    while Q_LS[j] ≠ [ ] do
10:       if j = n then return failed as S_{1,n} has no low support
11:       (W, i) ← Q_LS[j].dequeue(); rmNoLS(W, i, j, Q_LS, Q_HS)
12: for all j ← n − 1 to 2 do
13:    while Q_HS[j] ≠ [ ] do
14:       (W, i) ← Q_HS[j].dequeue(); rmNoHS(W, i, j, Q_LS, Q_HS)
15: while Q_HS[1] ≠ [ ] do
16:    (W, i) ← Q_HS[1].dequeue()
17:    for all (W → b) ∈ LS_{1,1} (W) do
18:       if C_i^HS (b) > 0 then
19:          if −−C_i^HS (b) = 0 then dom (X_i) ← dom (X_i) \ {b}
20:          if dom (X_i) = ∅ then return failed
21: return at-fixpoint
22: procedure rmNoLS(W, i, j, Q_LS, Q_HS)
23: if C_{i,j}^HS (W) > 0 then
24:    for all (F → YZ, k) ∈ HS'_{1,j} (W) do
25:       if W = Y ∧ F ∈ V_{i,k} ∧ Z ∈ V_{i+j,k−j} then
26:          (i_F, j_F, B, i_B, j_B) ← (i, k, Z, i + j, k − j)
27:       else if W = Z ∧ F ∈ V_{i−j,k} ∧ Y ∈ V_{i−j,k−j} then
28:          (i_F, j_F, B, i_B, j_B) ← (i − j, k, Y, i − j, k − j)
29:       else skip lines 30 to 33
30:       if C_{i_F,j_F}^LS (F) > 0 ∧ C_{i_B,j_B}^HS (B) > 0 ∧ C_{i,j}^HS (W) > 0 then
31:          if −−C_{i_F,j_F}^LS (F) = 0 then Q_LS[j_F].enqueue((F, i_F))
32:          if −−C_{i_B,j_B}^HS (B) = 0 then Q_HS[j_B].enqueue((B, i_B))
33:          if −−C_{i,j}^HS (W) = 0 then Q_HS[j].enqueue((W, i)); return
```

and $\text{HS}'_{i,j}(W)$ in the worst case, hence takes $\sum_{j=1}^{n} \sum_{i=1}^{n+1-j} \sum_{W \in V_{i,j}} |\text{LS}_{1,j}(W)| + \left|\text{HS}'_{1,j}(W)\right| = O\left(n^3 |G|\right)$ time, for similar reasons. Hence there is no asymptotic overhead by line 30, and the overall time complexity is $O\left(n^3 |G|\right)$.

Consider now the worst-case *space* complexity of our propagator. By Theorem 2, encoding the CYK table V, all $\text{LS}_{i,j}(W)$, and all $\text{HS}'_{i,j}(W)$ takes $O\left(n^2 |G|\right)$ space. There are $\sum_{j=1}^{n} \sum_{i=1}^{n+1-j} |V_{i,j}| = \sum_{j=1}^{n} \sum_{i=1}^{n+1-j} |V_{1,j}| = O\left(n^2 |N|\right) =$

$O\left(n^2\,|G|\right)$ non-terminals in V, hence storing the support counters for all non-terminals takes $O(n^2\,|G|)$ space. There are $n\,|\Sigma|$ terminals in the domains, hence storing the support counters for all terminals takes $O(n\,|G|)$ space. The two arrays Q_{LS} and Q_{HS} of queues contain at most all non-terminals in V, hence take $O(n^2\,|G|)$ space. The overall space complexity is thus $O\left(n^2\,|G|\right)$.

Although our propagator has the same *worst*-case time and space complexity as the one of [11], which is probably optimal anyway, our experiments below show that our propagator systematically beats it in practice (by up to two orders of magnitude), which might be confirmed by an *average*-case complexity analysis.

4 Experimental Evaluation

We now demonstrate the speed-up of our CFG propagator over its ancestor [11]. We implemented our propagator and the one of [11] in Gecode [8]. Katsirelos *et al.* [12] show how to reformulate a CFG into a DFA for a fixed length, as propagation for the Regular constraint is much cheaper than for CFG. This reformulation needs a propagator for the CFG constraint to shrink the initial domains of all decision variables to achieve GAC for all constraints at the root of the search tree, so that the obtained DFA is smaller. Hence this reformulation also benefits from a more efficient propagator for the CFG constraint.

Note that Sections 4.3 and 4.4 demonstrate that CP outperforms some state-of-the-art solvers from the verification literature by orders of magnitude on their own benchmarks. Our experimental results show that those benchmarks are trivial, *but these benchmarks were not known to be trivial* before this paper, and we have neither discarded any non-trivial benchmarks (of Hampi and Sushi) nor included the benchmarks that were in the meantime known to be trivial.

We use the Gecode built-in Regular propagator. We ran the experiments of Sections 4.1, 4.2, and 4.3 under Gecode 3.7.3, Hampi 20120213, and Ubuntu Linux 11.10 on 1.8 GHz Intel Core 2 Duo with 3GB RAM; and we ran the experiment of Section 4.4 under Gecode 3.7.3, Kaluza, Sushi 2.0, and Ubuntu Linux 10.04 with 1GB RAM in Oracle VirtualBox 4.2.4 (recommended by the Sushi developers) on the same hardware. As our chosen search heuristics do not randomise, all instances of Sections 4.1, 4.2, and 4.3 were run once. However, for Section 4.4, we ran each instance 10 times and recorded the average runtime, as the performance of the virtual machine might vary significantly.

4.1 A Shift Scheduling Problem

Demassey *et al.* [4] introduce a real-life shift scheduling problem for staff in a retail store. Let w be the number of workers, p the number of periods of the scheduling horizon, and a the number of work activities. The aim is to construct a $w \times p$ matrix of values in $[1, \ldots, a+3]$ (there are 3 non-work activities, namely break, lunch, and rest) to satisfy work regulation constraints, which can be modelled with a CFG constraint for each worker over the p periods and some global cardinality constraints (GCC).

Katsirelos *et al.* [12] model this problem as an optimisation problem, so that the reformulation of the grammar into a DFA takes only a tiny part of the runtime; they show that this optimisation problem is extremely difficult for CP-based CFG and REGULAR propagators. We are here, like [11], primarily interested in the first solution to the satisfaction version of this problem. We use the search heuristic of [11], namely selecting the second-largest value from the first decision variable with the minimum domain size in the last period with unassigned variables. HAMPI cannot handle multiple variables, while HAMPI, KALUZA, and SUSHI cannot model GCC, so we do not compare with them.

Table 1 gives our results: each row gives the instance, the search tree size, the DFA size after the reformulation of [12] of CFG into REGULAR, and the runtimes of four methods in seconds, namely our propagator (denoted by G++), the one of [11] (denoted by G), and the reformulation, using the two CFG propagators respectively (denoted by $\text{DFA}_{\text{G++}}$ and DFA_{G}). We find that G++ always works much better (up to 18 times) than G; $\text{DFA}_{\text{G++}}$ always works much better (up to 10 times) than DFA_{G}, as the reformulation of [12] itself needs a CFG propagator to shrink the initial domains at the root of the search tree (the reformulation, which is *instance-dependent*, is here taken on-line and takes about 85% of the total runtime) and as G++ works better than G; overall, G++ wins on 15 instances, and $\text{DFA}_{\text{G++}}$ wins on the other 2 instances. When solving for all or best solutions, $\text{DFA}_{\text{G++}}$ gradually takes over as the best method, as predicted by [12], but G++ continues to dominate G, and $\text{DFA}_{\text{G++}}$ decreasingly dominates DFA_{G}, as instances get harder.

4.2 A Forklift Scheduling Problem

Gange and Stuckey [7] introduce a forklift scheduling problem. Let s be the number of stations, i the number of items, and n the length of the scheduling horizon. There is a unique forklift and a shipping list giving the initial and final stations of each item. The aim is to construct an array of n actions, where an action can move the forklift from a station to any other station with a cost of 3, load an item from the current station onto the top of the forklift tray with a cost of 1, unload the item from the top of the forklift tray at the current station with a cost of 1, or do nothing with a cost of 0, so that the shipping list is accomplished with a minimised cost under forklift behaviour constraints, which can be modelled with one CFG constraint and i REGULAR constraints. We use the first-fail search heuristic, namely selecting the smallest value from the first decision variable with the minimum domain size, to solve this optimisation problem. Since HAMPI, KALUZA, and SUSHI cannot solve optimisation problems, we do not compare with them.

Table 2 gives our results over the instances solvable in one CPU hour: each row specifies the instance and gives the runtimes of two methods in seconds, namely our propagator (denoted by G++) and the one of [11] (denoted by G). We find that G++ always works better (up to 5 times) than G. The reformulation of [12] of the CFG constraint into the REGULAR constraint is not suitable for this problem, as the resulting automaton is of size exponential in n.

Table 1. Runtimes for the shift scheduling problem

benchmark ($p = 96$)			search tree size			DFA	runtimes of four methods in seconds			
instance	a	w	#nodes	#propagations	#fails	$\|A\|$	G++	$\text{DFA}_{\text{G++}}$	DFA_{G}	G
1_1	1	1	11	438	1	446	**0.24**	0.49	4.26	3.93
1_2	1	3	133	2123	33	998	**0.90**	3.78	15.38	12.87
1_3	1	4	349	5790	137	998	**1.68**	4.10	19.48	19.49
1_4	1	5	95	1836	7	814	**1.18**	2.41	21.99	20.53
1_5	1	4	71	1332	3	722	**0.92**	1.75	16.95	16.32
1_6	1	5	76	1567	3	722	**1.17**	2.01	21.16	20.17
1_7	1	6	3623	56635	1773	814	7.87	**2.97**	25.56	47.48
1_8	1	2	57	1005	10	998	**0.52**	3.59	10.76	8.47
1_9	1	1	19	460	1	630	**0.22**	0.80	4.41	3.94
1_10	1	7	12699	209988	6305	814	23.31	**4.02**	30.14	100.95
2_1	2	2	46	1414	8	984	**0.93**	1.69	16.76	15.97
2_5	2	4	83	2208	20	1209	**1.02**	3.15	18.51	16.41
2_6	2	5	89	1801	12	1207	**1.35**	2.94	23.03	21.57
2_7	2	6	258	5847	104	944	**1.97**	2.63	32.22	32.03
2_8	2	2	1046	28691	500	1774	**2.86**	7.75	23.09	24.09
2_9	2	1	35	1249	8	1460	**0.63**	4.11	14.21	11.03
2_10	2	7	4690	100007	2302	1506	**7.64**	7.82	43.24	53.90

Table 2. Runtimes for the forklift scheduling problem

instance			runtimes in seconds		instance			runtimes in seconds	
s	i	n	G++	G	s	i	n	G++	G
3	4	15	**4.35**	20.02	3	4	16	**22.64**	103.75
3	4	17	**20.98**	100.48	3	4	18	**76.77**	382.31
3	4	19	**72.66**	338.69	3	4	20	**197.98**	1013.78
3	5	16	**67.54**	297.55	3	5	17	**81.67**	368.65
3	5	18	**200.91**	1058.17	3	6	18	**1134.58**	5008.90
4	5	17	**388.92**	1631.94	4	5	18	**819.82**	3876.87

4.3 Intersection of Two Context-Free Languages

HAMPI [14] selects a subset of 100 CFG pairs (from the benchmark of CFGAnalyzer [1]), where a string of length $1 \leq n \leq 50$ accepted by both CFGs in each pair is to be found (8 instances are satisfiable and 92 are unsatisfiable; disjointness of two context-free languages is undecidable). The CFGs of this benchmark have 10 to 600 productions in CNF and up to 18 alphabet symbols. This problem can also be solved using tools from automata theory. On this benchmark, HAMPI beats CFGAnalyzer by a large margin. HAMPI also beats other ad hoc solvers on other benchmarks, which are too easy (HAMPI solves them in one second), hence any improvements might be subject to runtime measurement errors.

Instead of running each CFG pair 50 times with the n-th run to find a string of length n accepted by both CFGs, we search once, namely for the first solution string of length *up to* 50 for each pair. Given a CFG $G = \langle \Sigma, N, P, S \rangle$, we create a new CFG $G' = \langle \Sigma', N', P', S' \rangle$ with $\Sigma' = \Sigma \cup \{\#\}$ (let $\# \notin \Sigma$ denote a dummy symbol), $N' = N \cup \{S'\}$, and $P' = P \cup \{S' \to S \mid S'\#\}$. If a string s' of length

n is accepted by G', then the string s obtained by removing all '#' at the end of s' has a length up to n and is accepted by G.

Given a CFG pair (G_1, G_2), our model is $\text{CFG}(X, G'_1) \wedge \text{CFG}(X, G'_2)$, where X is a sequence of n decision variables with $\text{dom}(X_i) = \Sigma'_1 \cup \Sigma'_2$. Our search heuristic is to select the first value from the last unassigned variable. Figure 2 gives the runtimes of HAMPI and the two CFG propagators for the 55 instances where HAMPI takes at least one second. Each '$\times$' (or '+') denotes the comparison between our propagator (or the one of [11]) and HAMPI; each 'Δ' denotes the solving time of the bit-vector solver STP. For all 100 instances, the two propagators always work much better (up to 9000 times) than HAMPI, and even always work much better than STP when the fixed-sizing of the grammar into a regular expression and the transformation into bit-vector constraints are taken off-line; our propagator always works much better (up to 250 times) than the one of [11]. As 97 instances turn out to be solvable at the root of the search tree, the reformulation of [12] of the CFG constraint into the REGULAR constraint has similar results; for the other 3 instances, our CFG propagator is 3 to 5 orders of magnitude faster (details omitted for space reasons, see Appendix C of [9]). The two CFG propagators always beat HAMPI for all $n < 50$ (up to 380 times even with $n = 10$), and whether run on the CFG pair (G'_1, G'_2) or the original pair (G_1, G_2). We get similar speed-ups (details omitted for space reasons, see Appendix C of [9]) over 99% of the CFG pairs even with the first-fail search heuristic. Note that KALUZA uses HAMPI's functionality to solve the CFG constraint, hence KALUZA has the same performance as HAMPI on this benchmark.

4.4 Solving String Equations

Fu *et al.* [5] introduce just one benchmark of 5 string equations with a parameter $1 \leq n \leq 37$ to demonstrate the practicality of their string solver SUSHI against KALUZA. SUSHI handles string variables of *unbounded* length. Like KALUZA, we expect a user-given parameter $\overline{n}$ and look for the first solution string of up to $\overline{n}$ symbols. Unlike KALUZA, which tries all lengths until $\overline{n}$, we allow strings to end with dummy symbols '#' (as in Section 4.3) and add length constraints. For a sequence $X = \langle X_1, \ldots, X_{\overline{n}} \rangle$, let decision variable n_X with $\text{dom}(n_X) = \{0, \ldots, \overline{n}\}$ denote the index of the right-most non-dummy symbol in X. The length constraint is $\forall 1 \leq i \leq \overline{n} : X_i = \# \Leftrightarrow n_X < i$. String concatenation $X = Y + Z$ is modelled as $n_X = n_Y + n_Z \wedge \langle X_1, \ldots, X_{n_X} \rangle = \langle Y_1, \ldots, Y_{n_Y}, Z_1, \ldots, Z_{n_Z} \rangle$ with reification constraints. Regular language membership $X \in L(R)$, where $L(R)$ denotes the language accepted by the regular expression R, is modelled as $\text{REGULAR}(X, R\#^*)$. We use the first-fail search heuristic. Table 3 gives the runtimes of GECODE, SUSHI, and KALUZA for equations 1 to 3 with the *hardest* setting $n = 37$ and the KALUZA models (for a fair comparison). As KALUZA solves the equations for some $n \leq \overline{n} < 3n$, we *pessimistically* set $\overline{n} = 4n$ for GECODE, and GECODE *still* beats SUSHI and KALUZA, by up to 130 times. GECODE solves our better models than the KALUZA ones of equations 4 and 5 within 0.10 seconds, beating SUSHI and KALUZA by up to 3000 times.

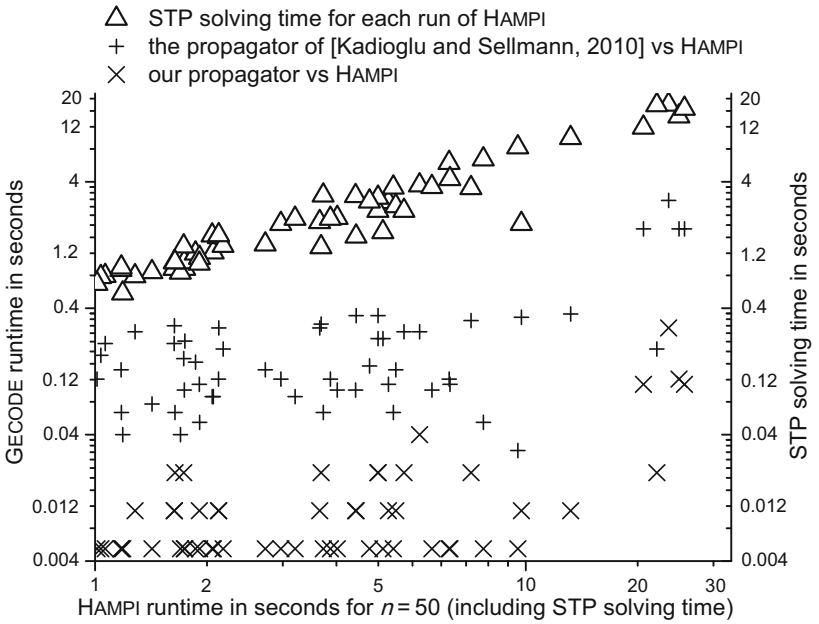

Fig. 2. Runtimes for the CFG-intersection problem

Table 3. Runtimes (in seconds) for solving string equations

	eq1: 3 string variables			eq2: 2 string variables			eq3: 4 string variables		
n	GECODE	SUSHI	KALUZA	GECODE	SUSHI	KALUZA	GECODE	SUSHI	KALUZA
37	**0.15**	1.34	10.40	**0.05**	1.82	3.94	**0.07**	2.52	5.71

5 Conclusion

We argue that CP solvers are more suitable than existing solvers for verification tools that solve string constraints. Indeed, CP has a rich tradition of constraints for membership in formal languages: their propagators run directly on descriptions, such as automata and grammars, of these languages. Apparently tricky features, such as string equality or multiple string variables (with shared characters), pose no problem to CP. Future work includes designing propagators for string constraints over strings of (un)bounded length.

Acknowledgements. The first three authors are supported by grants 2007-6445 and 2011-6133 of the Swedish Research Council (VR), and Jun He is also supported by grant 2008-611010 of China Scholarship Council and the National University of Defence Technology of China. Many thanks to Xiang Fu, Serdar Kadıoğlu, George Katsirelos, Adam Kieżun, Prateek Saxena, and Guido Tack for useful discussions during the preparation of this work.

References

1. Axelsson, R., Heljanko, K., Lange, M.: Analyzing context-free grammars using an incremental SAT solver. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008, Part II. LNCS, vol. 5126, pp. 410–422. Springer, Heidelberg (2008)
2. Beldiceanu, N., Carlsson, M., Petit, T.: Deriving filtering algorithms from constraint checkers. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 107–122. Springer, Heidelberg (2004)
3. Bessière, C.: Arc-consistency and arc-consistency again. Artificial Intelligence 65(1), 179–190 (1994)
4. Demassey, S., Pesant, G., Rousseau, L.-M.: A cost-regular based hybrid column generation approach. Constraints 11(4), 315–333 (2006)
5. Fu, X., Powell, M.C., Bantegui, M., Li, C.-C.: Simple linear string constraints. Formal Aspects of Computing (2012), Published on-line in January 2012 and available from `http://dx.doi.org/10.1007/s00165-011-0214-3`. SUSHI is available from `http://people.hofstra.edu/Xiang_Fu/XiangFu/projects/SAFELI/SUSHI.php`
6. Ganesh, V., Dill, D.L.: A decision procedure for bit-vectors and arrays. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 519–531. Springer, Heidelberg (2007), STP is available from `https://sites.google.com/site/stpfastprover/`
7. Gange, G., Stuckey, P.J.: Explaining propagators for s-DNNF circuits. In: Beldiceanu, N., Jussien, N., Pinson, É. (eds.) CPAIOR 2012. LNCS, vol. 7298, pp. 195–210. Springer, Heidelberg (2012)
8. Gecode Team. Gecode: A generic constraint development environment (2006), `http://www.gecode.org/`
9. He, J.: Constraints for Membership in Formal Languages under Systematic Search and Stochastic Local Search. PhD thesis, Uppsala University, Sweden (2013), `http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-196347`
10. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation. Addison Wesley, New York (1979)
11. Kadioğlu, S., Sellmann, M.: Grammar constraints. Constraints 15(1), 117–144 (2008); An early version is published in the Proceedings of the 23rd AAAI Conference on Artificial Intelligence in 2008
12. Katsirelos, G., Narodytska, N., Walsh, T.: Reformulating global grammar constraints. In: van Hoeve, W.-J., Hooker, J.N. (eds.) CPAIOR 2009. LNCS, vol. 5547, pp. 132–147. Springer, Heidelberg (2009)
13. Katsirelos, G., Narodytska, N., Walsh, T.: The weighted GRAMMAR constraint. Annals of Operations Research 184(1), 179–207 (2011), An early version is published in the Proceedings of the 5th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems in 2008
14. Kieżun, A., Ganesh, V., Guo, P.J., Hooimeijer, P., Ernst, M.D.: HAMPI: A solver for string constraints. In: Proceedings of the 18th International Symposium on Software Testing and Analysis, Chicago, USA, July 2009, pp. 105–116. ACM Press (2009), HAMPI is available from `http://people.csail.mit.edu/akiezun/hampi/`
15. Mohr, R., Henderson, T.C.: Arc and path consistency revisited. Artificial Intelligence 28(2), 225–233 (1986)
16. Pesant, G.: A regular language membership constraint for finite sequences of variables. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 482–495. Springer, Heidelberg (2004)

17. Quimper, C.-G., Walsh, T.: Global grammar constraints. In: Benhamou, F. (ed.) CP 2006. LNCS, vol. 4204, pp. 751–755. Springer, Heidelberg (2006)
18. Quimper, C.-G., Walsh, T.: Decomposing global grammar constraints. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 590–604. Springer, Heidelberg (2007)
19. Saxena, P., Akhawe, D., Hanna, S., Mao, F., McCamant, S., Song, D.: A symbolic execution framework for javascript. In: Proceedings of the 31st IEEE Symposium on Security and Privacy, California, USA, pp. 513–528. IEEE Press (May 2010), Kaluza is available from `http://webblaze.cs.berkeley.edu/2010/kaluza/`
20. Sellmann, M.: The theory of grammar constraints. In: Benhamou, F. (ed.) CP 2006. LNCS, vol. 4204, pp. 530–544. Springer, Heidelberg (2006)