Sharpening Constraint Programming approaches for Bit-Vector Theory*

Zakaria Chihani, Bruno Marre, François Bobot, Sébastien Bardin
CEA, LIST, Software Security Lab, Gif-sur-Yvette, France
(first.last@cea.fr)

Abstract

We address the challenge of developing efficient Constraint Programming-based approaches for solving formulas over the quantifier-free fragment of the theory of bitvectors (BV), which is of paramount importance in software verification. We propose CP(BV), a highly efficient BV resolution technique built on carefully chosen anterior results sharpened with key original features such as thorough domain combination or dedicated labeling. Extensive experimental evaluations demonstrate that CP(BV) is much more efficient than previous similar attempts from the CP community, that it is indeed able to solve the majority of the standard verification benchmarks for bitvectors, and that it already complements the standard SMT approaches on several crucial (and industry-relevant) aspects, notably in terms of scalability w.r.t. bit-width, theory combination or intricate mix of non-linear arithmetic and bitwise operators. This work paves the way toward building competitive CP-based verification-oriented solvers.

1 Introduction

Context. Not so long ago, program verification was such an ambitious goal that even brilliant minds decided it was "bound to fail" [37]. At the time, the authors concluded their controversial paper saying that if, despite all their reasons, " verification still seems an avenue worth exploring, so be it". And so it was. Today, software verification is a well established field of research, and industrial adoption has been achieved is some key areas, such as safety-critical systems.

Since the early 2000's, there is a significant trend in the research community toward reducing verification problems to satisfiability problems of first-order logical formulas over well-chosen theories (e.g. bitvectors, arrays or floating-point arithmetic), leveraging the advances of modern powerful SAT and SMT solvers [30, 43, 38, 6]. Besides weakest-precondition calculi dating back to the 1970's [19], most major recent verification approaches follow this idea [15, 25, 29, 35].

*Work partially funded by ANR under grants ANR-14-CE28-0020 and ANR-12-INSE-0002. The CP solver COLIBRI is generously sponsored by IRSN, the French nuclear safety agency.

The problem. While SMT and SAT are the *de facto* standard in verification, a few teams explore how Constraint Programming (CP) techniques can be used in that setting [33, 16, 26, 42, 27]. Indeed, CP could in principle improve over some of the well-known weaknesses of SMT approaches, such as non-native handling of finite domains theories (encoded in the Boolean part of the formula, losing the high-level structure) or very restricted theory combinations [39].

Yet, currently, there is no good CP-based resolution technique for (the quantifier-free fragment of) the theory of bitvectors [30], i.e. fixed-size arrays of bits equipped with standard low-level machine instructions, which is of paramount importance in verification since it allows to encode most of the basic datatypes found in any programming language.

Goal and challenge. We address the challenge of developing efficient Constraint Programming-based approaches for solving formulas over the quantifier-free fragment of the theory of bitvectors (BV). Our goal is to be able to solve many practical problems arising from verification (with the SMTCOMP challenge [1] as a benchmark) and to be at least complementary to current best SMT approaches. The very few anterior results were still quite far from these objectives [44], even if preliminary work by some of the present authors was promising on conjunctive-only formulas [1].

Proposal and contributions. We propose CP(BV), a highly efficient BV resolution technique built on carefully chosen anterior results [1, 36] sharpened with key original features such as thorough domain combination or dedicated labeling. Extensive experimental evaluations demonstrate that CP(BV) is much more efficient than previous similar attempts from the CP community, that it is indeed able to solve the majority of the standard verification benchmarks for bitvectors, and that it already complements the standard SMT approaches on several crucial (and industry-relevant) aspects, notably in terms of scalability w.r.t. bit-width, theory combination or intricate mix of non-linear arithmetic and bitwise operators. Our main contributions are the following:

- We present CP(BV), an original framework for CP-based resolution of BV problems, which built on anterior results and extend them with several key new features in terms of thorough domain combination or dedicated labeling. This results in a competitive CP-based solver, excelling in key aspects such as scalability w.r.t. bitwidth and combination of theories. A comparison of CP(BV) with previous work is presented in Table 1.

- We perform a systematic and extensive evaluation of the effect of our different CP improvements on the efficiency of our implementation. This compares the options at our disposal and justifies those we retained, establishing a firm ground onto which future improvements can be made. It also shows the advantage of our approach relative to the other CP approaches applied to BV, and that our approach is able to solve a very substantial part of problems from the SMTCOMP challenge.

- Finally, we perform an extensive comparison against the best SMT solvers for BV problems, namely: Z3 [8], Yices [20], MathSAT [14], CVC4 [4] and Boolector [11].

[1] `smtcomp.sourceforge.net`

This comparison exhibits the strenghts of CP over SMT approaches on particular instances. Specifically, our implementation surpasses several (and sometimes *all*) solvers on some examples involving large bit-vectors and/or combination with floating-point arithmetic.

Table 1: Overview of our method

	Bardin *et al* [1]	Michel *et al* [36]	CP(BV)
bitvector domain	+	++	++ [36]
arithmetic domain	++	+	++ [1]
domain combination	+	+	++
simplifications	+	x	++
BV-aware labeling	x	x	++
implemented	yes	no	yes
benchmark	conjunctive formulas ≈ 200 formulas		arbitrary formulas ≈ 30,000 formula

Discussion. Considering that our current CP(BV) approach is far from optimal compared to existing SMT solvers (implemented in Prolog, no learning), we consider this work as an important landmark toward building competitive CP-based verification-oriented solvers. Moreover, our approach clearly challenges the well-accepted belief that bitvector solving is better done through bitblasting, opening the way for a new generation of word-level solvers.

2 Motivation

The standard (SMT) approach for solving bit-vector problem, called *bit-blasting* [7], relies on a boolean encoding of the initial bitvector problem, one boolean variable being associated to each bit of a bitvector. This *low-level encoding* allows for a direct reuse of the very mature and ever-evolving tools of the SAT community, especially DPLL-style SAT solvers [38, 43]. Yet, crucial high-level structural information may be lost during bitblasting, leading to potentially poor reasoning abilities on certain kinds of problems, typically those involving many arithmetic operations [12] and large-size bitvectors. Following anterior work [1, 36], we propose a *high-level encoding* of bitvector problems, seen as Constraint Satisfaction Problems (CSP) over finite (but potentially huge) domains. Each bitvector variable of the original BV problem is now considered as a (CSP) bounded-arithmetic variable, with dedicated domains and propagators.

Now illustrating with concrete examples, we show the kind of problems where our approach can surpass existing SMT solvers. Consider the three following formulas:

$$x \times y = (x \;\&\; y) \times (x \mid y) + (x \;\&\; \overline{y}) \times (\overline{x} \;\&\; y) \tag{A}$$

$$x_1 < x_2 < \cdots < x_n < x_1 \tag{B}$$

$$(\bigwedge_{i=1}^{n-2} x_i < x_{i+1} \;\&\; x_{i+2}) \wedge (x_{n-1} < x_n \;\&\; x_1) \wedge (x_n < x_1 \;\&\; x_2) \tag{C}$$

where $\bar{\cdot}$ (resp. $\cdot \,\&\, \cdot$, $\cdot \mid \cdot$) is the bit-wise negation (resp. conjunction, disjunction), $\wedge$ is the logical conjunction, $<$ is an unsigned comparison operator, n was chosen to be 7. As an example, the SMT-LIB language [5] encoding of formula A is:

```
(assert (= (bvmul X Y) (bvadd (bvmul (bvand X Y) (bvor X Y))
(bvmul (bvand X (bvnot Y)) (bvand (bvnot X) Y)))))
```

Table 2 shows the time in seconds according to bit-vector size, both for the satisfiability proof of the *valid* formula A and the unsatisfiablity of B and C. CP(BV) is the name of our technique, and TO means that the solver was halted after a 60-second timeout.

Table 2: Comparison of performance (time in sec.) for different solvers

Formula	size(bits)	Z3	Yices	MathSAT	CVC4	Boolector	CP(BV)
A	512	TO	1.60	6.04	17.28	20.55	0.24
	1024	TO	7.25	26.72	TO	TO	0.23
	2048	TO	31.83	TO	TO	TO	0.23
B	512	0.53	0.82	1.37	0	2.75	0.26
	1024	1.75	4.89	4.23	0	7.39	0.22
	2048	5.73	16.15	22.76	0	16.81	0.21
C	512	0.15	0.85	1.55	0.76	3.15	0.25
	1024	0.33	1.25	4.53	3.49	3.81	0.22
	2048	0.70	5.55	19.57	8.82	14.73	0.25

On these examples, CP(BV) clearly surpasses SMT solvers, reporting no TO and a very low (size-independent) solving time. In light of this, we wish to emphasize the following advantages of our CP(BV) high-level encoding for bitvector solving:

- Each variable is attached to *several and complementary domain representations*, in our case: *intervals plus congruence*, *bitlist* [1] (i.e. constraints on the possible values of specific bits of the bitvector) and *global difference constraint* (delta). Each domain representation comes with its own *constraint propagation* algorithms and deals with different aspects of the formula to solve. We will call *integer domains* or *arithmetic domains* those domains dealing mainly with high-level arithmetic properties (here: intervals, congruence, deltas), and *BV domains* or *bitvector domains* those domains dealing with low-level aspects (here: bitlist).

- These domains can *freely communicate between each other*, each one refining the other through a *reduced product* (or *channeling*) mechanism [41]. For example, in formula B, adding the difference constraint to the delta domain does allow CP(BV) to conclude *unsat* directly at propagation. Having multiple domains also allows to search for a solution in the smallest of them (in terms of the cardinality of the concretization of the domain abstraction).

- In the case of formula C, a reduced product is not enough to conclude at propagation. Here, a BV constraint *itself* refines an arithmetic domain: indeed, with the simple observation that, if $a \,\&\, b = c$ then $c \leqslant a$ and $c \leqslant b$, the bit-vector part of CP(BV) not only acts on the bit-vector representation of the variables, but also "informs" the global difference constraints of a link it could not have found on its own.

3 Background

This section lays down the ground on which our research was carried out, both the theoretical foundations, anterior works and the COLIBRI CP solver [33].

3.1 BV theory

We recall that BV is the quantifier-free theory of bitvectors [30], i.e. a theory where variables are interpreted over fixed-size arrays (or vectors) of bits along with their basic operations: logical operators (conjunction " & ", disjunction "|" and xor "$\oplus$", *etc.*), modular arithmetic (addition $+$, multiplication $\times$, *etc.*) and other structural manipulations (concatenation $::$, extraction $\lfloor . \rceil_{i.j}$, etc.).

3.2 CP for Finite-Domain Arithmetic

A *Constraint Satisfaction Problem* [18] (CSP) consists in a finite set of variables ranging over some domains, together with a set of constraints over these variables – each constraint defining its own set of solutions (valuations of the variables that satisfy the constraint). Solving a CSP consists in finding a solution meeting all the constraints of the CSP, or proving that no such solution exists. We are interested here only in the case where domains are finite. *Constraint Programming* [18] (CP) consists in solving a CSP through a combination of *propagation* and *search*. Propagation consists mainly in reducing the potential domains of the CSP variables by deducing that some values cannot be part of any solution. Once no more propagation is possible, search consists in assigning a value to a variable (taken from its reduced domain) and continue the exploration, with backtrack if required.

The CP discipline gets its strenght from global constraints and capabilities for dense interreductions between domain representations, along with constraint solving machinery. We present in this section standard domains and constraints for bounded arithmetic.

Union of intervals. A simple interval $[a; d]$, where $a, d \in N$ represents the fact that a given variable can take values only between a and d. A natural extension of this notion is the *union of intervals* (*Is*). As a shortened notation, if $x \in \{a\} \uplus [b; c] \uplus \{d\}$, one writes $[x] = [a, b..c, d]$.

Congruence [31]. If the division remainder of a variable x by a divisor d is r (*i.e.,* x satisfies the equation $x\%d = r$), then $\langle x \rangle = r[d]$ is a *congruence* and represents all values that variable x can take, *e.g.,* $\langle x \rangle = 5[8]$ means $x \in \{5, 13, 21, \ldots\}$.

Global difference constraint (Delta) [23]. A global difference constraint is a set of linear constraints of the form $x - y \diamond k$ where $\diamond \in \{=, \neq, <, >, \leqslant, \geqslant\}$. Tracking such sets of constraints allows for better domain propagation and early infeasibility detection, thanks to a global view of the problem compared with the previous (local) domains.

3.3 The COLIBRI Solver for FD Arithmetic

The COLIBRI CP solver [33] was initially developped to assist CEA verification tools [9, 47, 2, 17]. COLIBRI supports bounded integers (both standard and modular arithmetic [28]), floating-points [34] and reals. Considering arithmetic, COLIBRI already provides all the domains described in Section 3.2, together with standard propagation techniques and strong interreductions between domains. Search relies mostly on a standard fail-first heuristics. COLIBRI is implemented in ECLiPSe Prolog, yielding a significant performance penalty (compared with compiled imperative languages such as C or C++) but allowing to quickly prototype new ideas.

3.4 Former CP(BV) approaches

Two papers must be credited with supplying the inspiration for this work, written by Bardin et al [1] and by Michel and Van Hentenryck [36]. Put together, these two papers had good ideas, which we adopted, unsatisfactory ideas which were disgarded, and finally ideas that were not advanced enough which we extended.

The first paper [1] introduces the bitlist domain, i.e. lists of four-valued items ranging over $\{0, 1, ?, \bot\}$ – indicating that the i^{th} bit of a bitvector must be 0, 1, any of these two values (?), or that a contradiction has been found ($\bot$) – together with its propagators for BV operators. Moreover, the authors also explain how arithmetic domains (union of intervals and congruence) can be used for BV, and describe first interreduction mechanisms between bitlists and arithmetic domains.

The second paper [36] introduces a very optimized implementation of bitlists, using two BVs $\langle {}^1x, {}^0x\rangle$ to represent the bitlist of x, where ${}^1x$ (resp. ${}^0x$) represents bits known to be set (resp. cleared) in x. The efficiency comes from the use of machine-level operations for performing domain operations, yielding constant time propagation algorithms for reasonable bitvector sizes.

Basically, we improve the technique described in [1] by: borrowing the optimized bitvector domain representation from [36] (with a few very slight improvements), significantly improving inter-domain reduction, and designing a BV-dedicated search labeling strategy.

As improving inter-domain reduction is one of our key contributions, we present hereafter the reductions between BV domains and arithmetic domains described in [1]:

With congruence: the BV domain interacts according to the longest sequence of known least significant bits. For example, a BV domain $[\![10?00?101]\!]$ of a variable b indicates that b satisfies the equation $b[8] = 5$, which therefore constrains the congruence domain using $5[8]$. Conversely a known congruence of some power of 2 fixes the least significant bits.

With *Is*: for a variable x, the union of intervals can refine the most significant bits of the BV domain by clearing bits according to the power of two that is immediately greater than the maximum extremum of the *Is*. And the BV domain influences *Is* by (only) refining the extrema through the maximum and minimum bit-vectors allowed, and by removing *singletons* that do not conform to the BV domain.

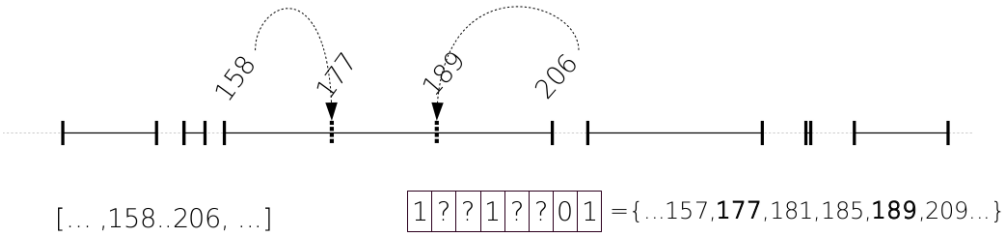

Figure 1: Enlarging the gaps in the *Is* according to the BV domain

4 Boosting CP(BV) for Efficient Handling of Bit-Vectors

In this section, we delve into the specificities of our approach, leaving complex details to a technical report[2]. We first start by presenting a significantly improved inter-domain reduction, followed by new reductions from BV constraints to other domains, then we show some simplifications and factorisations at the constraint level, and we finish by presenting our BV-dedicated labeling strategy.

4.1 Better interreduction between BV- and arithmetic- domains

We present here several significant improvements to the inter-domain reduction techniques proposed in [1] between bitlists and unions of intervals. Our implementation also borrows the inter-reduction between bitlists and congruence from [1].

***Is* to BVs.** Let m and M be respectively the minimal and the maximal value of a union of intervals. Then, the longest sequence of most-significant bits on which they "agree" can also be fixed in the bit-vector domain. For example, $m = 48$ and $M = 52$ (00110000 and 00110100 in binary) share their five most-significant bits, denoted $[\![00110???]\!]$. Therefore, a bit-vector $bl = [\![0??1???0]\!]$ can be refined into $[\![00110??0]\!]$. For comparison, the technique in [1] only reduces bl to $[\![00?1???0]\!]$.

BV to *Is*. Consider a variable b with a *Is* domain $[b] = [153, 155, 158..206, 209]$, and a bit-vector domain $(\!|b|\!) = [\![1??1??01]\!] = \{\cdots, 153, 157, 177, 181, 185, 189, 209, \cdots\}$, as illustrated in Figure 1. The inter-domain reduction from [1] can refine the extremum of the *Is* (here: nothing to do, since 153 and 209 both conforms to $(\!|b|\!)$) and removes the singletons that are not in $(\!|b|\!)$ (here: 155), yielding $[b] = [153, 158..206, 209]$. We propose to go a step further by refining each bound inside a *Is*, such that after reduction each bound of $[b]$ conforms to $(\!|b|\!)$. Here, 158 (resp. 206) is not allowed and should be replaced by its closest upper (resp. lower) value in $(\!|b|\!)$, i.e. 177 (resp. 189), yielding $[b] = [153, 177..189, 209]$.

We have designed such a correct and optimal reduction algorithm from bitlist to *Is*. Since we work on the $\langle {}^1x, {}^0x \rangle$ representation of bitlists, the algorithm relies on machine-level operations and is linear in the size of the bitvector (cf. technical report). We describe the procedure for increasing a lower bound in Alg. 1; de-

[2] `sites.google.com/site/zakchihani/cpaior`

creasing the upper bound (symmetrically) follows the same principle (cf. technical report). In order to calculate a new bound r accepted by the bit-vector domain $(\!|b|\!)$, we start by imposing on the lower bound l what we already know, *i.e.,* set what is set in ${}^1b$ and clear what is cleared in ${}^0b$ (line 1 of Alg 1). Then flag the bits that were changed by this operation, going from cleared to set and from set to cleared. To refine the lower bound, we must raise it as much as necessary but not one bit higher, *i.e.,* we should look for the smallest amount to add to the lower bound in order to make it part of the concretisation of $(\!|b|\!)$. This entails two things: a cleared bit can only become set if all bits of lower significance get cleared. For example, to increase the binary represented integer **0**10 exactly until the left-most bit gets set, we will pass by 011 and *stop* at **1**00 : going to 101 would increase more than strictly necessary. Similarly, the *smallest* increase that clears a set bit i is one where the *first cleared bit on the left* of i can be set (line 12 of Alg 1, function `left-cl-can-set-of`). For example, to clear the third most significant bit (in bold) in 01**1**011, one needs to increase to 01**1**100, 01**1**101, 01**1**110, 01**1**111 then reach 10**0**000. Doing so clears not only the target bit i but all the bits of lower significance.

Algorithm 1 Increasing the lower bound l according to $(\!|b|\!)$

```
1: r := ¹b | l & ⁰b
2: set2cl:= l & r̄
3: cl2set:= l̄ & r
4: if cl2set > set2cl then
5:     size:=log2(cl2set)
6:     mask0 := −1 ≪ size
7:     can-cl:= mask0 | ¹b
8:     r := r & can-cl
9: else
10:    size:=log2(set2cl)
11:    cl-can-set:= r̄ & ⁰b
12:    next-to-set := left-cl-can-set-of(size, cl-can-set)
13:    r :=set(r,next-to-set)
14:    mask0 := −1 ≪next-to-set
15:    can-cl:= mask0 | ¹b
16:    r := r & can-cl
17: end if
```

Drilling the *Is* according to BV. If the *Is* contains only one interval, then our technique does not improve over [1], and is only slighly superior to the channeling method of [36]. For this reason, we force the bit-vector domain to create at least one gap in the union of intervals. Consider for example a domain $bl = [\![0?10?1?]\!]$. When observing the concretisation $\{18, 19, 22, 23, 50, 51, 54, 55\}$, the largest gap is between 23 and 50, *i.e.,* 0010111 and 0110010, obtained by fixing the *most significant unknown bit (msub).* More generally, for a variable x the largest gap is created by intersecting $[x]$ with $[{}^1x{\cdot\cdot}a, b{\cdot\cdot}{}^0x]$, where a is obtained by clearing the *msub* and setting all other unknown bits, and b is obtained by setting the *msub* and clearing all other unknown bits. One can of course enforce more gaps, but there is a tradeoff between their propagation cost (as *Is*) and their benefits. In this work, using one gap was satisfactory.

4.2 BV constraints reducing arithmetic domains

Our CP(BV) approach strives to keep each of its domains as aware as possible of the other domains. We now show how non-BV domains can be reduced through BV constraints. In the following, we recall that $[x]$ denotes the union of intervals attached to variable x.

4.2.1 BV constraints on *Is*

It turns out that most BV constraints can influence unions of intervals.

Bitwise binary operands: a disjunction $x \mid y = z$ can refine $[z]$ in more than one way, but experimentation showed a notable effect only when $[x]$ and $[y]$ contain only singletons, at which case $[z]$ can be refined by the pairwise disjunction of those singletons. Similar refinements can occure through the bitwise conjunction and exclusive disjunction. For the latter, one can also refine in the same manner the *Is* of the operands, since $x \oplus y = z$ implies the same constraint for all permutation of x, y, z.

Negation: from a negation constraint $x = \overline{y}$, one can refine the *Is* of one variable from that of the other. By mapping each singleton $\{c\} \in [x]$ and interval $a..b \in [x]$ to $\{\overline{c}\}$ and $\overline{b}..\overline{a}$, we build a *Is* to populate $[y]$. The symmetric construction populates $[x]$.

Shifts: from a right shift $x \gg y = z$, which is equivalent to a natural division (*i.e.,* $x/2^y = z$), one can refine $[z]$ simply by right-shifting all elements of $[x]$ (singletons and bounds of internal intervals) by y. The left-shift constraint is treated mostly in the same way but requires extra care as it is a modular multiplication and it can overflow.

Sign-extension: when extending the sign of x by i positions to obtain z, the method consists in splitting the $[x]$ by the integers that are interpreted as negative, most significant bit is 1, and the one interpreted as positive, most significant bit is 0 and to apply the sign extention separately, disjunction with $2^i - 1 \ll i$ for the firsts and identity for the seconds.

Extractions: when extracting from the left-most to any position, it's the same as a right logical shift. The more general case is tricky. Take $\lfloor x \rceil_{i.j} = y$ to mean the extraction from bit i to j of x to obtain y (with $\|x\| > i \geqslant j \geqslant 0$), then a singleton in for an interval $x_a..x_b$,

- If $(x_b \gg j) - (x_a \gg j) > 2^{i-j}$, then the interval necessarily went through all integer coded on i bits, so the integer domain cannot be refined.
- else, if $(x_b \oplus x_a)\ \&\ 2^i = 0$, then no power of 2 was traversed, the bounds can simply be truncated and stay in that order: $(\lfloor x_a \rceil_{i.j})..(\lfloor x_b \rceil_{i.j})$
- else, 2^i was traversed, then the *Is* is $[0..(\lfloor x_b \rceil_{i.j}), (\lfloor x_a \rceil_{i.j})..(2^{(i-j)} - 1)]$

For example, using the binary representation for the integer bounds of a union of intervals, an extraction of the 3 rightmost bits of a variable whose union of intervals contains 01110..10011 would not produce the invalid interval 110..011 because its lower bound is greater than its upper bound. This falls in the third case above, and would generate the two intervals 000..011 and 110..111.

Concatenation: for a concatenation $x :: y = z$, the inner structure of $[z]$ can be refined from $[x]$ and $[y]$. Let $v^{\ll x}$ be $v \mid (x \ll \|y\|)$ and $(a{\cdot\cdot}b)^{\ll x}$ be $a^{\ll x}{\cdot\cdot}b^{\ll x}$. For example, if $[x] = [x_a{\cdot\cdot}x_b]$ and $[y] = [y_1, y_2{\cdot\cdot}y_3, y_4{\cdot\cdot}y_5]$, then $[z]$ can be refined by $[(y_1)^{\ll x_a}, (y_2{\cdot\cdot}y_3)^{\ll x_a}, (y_4^{\ll x_a}){\cdot\cdot}(y_1^{\ll x_b}), (y_2{\cdot\cdot}y_3)^{\ll x_b}, (y_4{\cdot\cdot}y_5)^{\ll x_b}]$. The algorithm is described in the technical report. One can also refine $[x]$ and $[y]$ from $[z]$.

4.2.2 BV constraints on deltas

As seen in the motivation section, keeping the deltas informed of the relationship between different variables can be an important factor for efficient reasoning.

Bitwise operations: a constraint $x \mid y = c$ implies that that $(c - y \leqslant x \leqslant c) \wedge (c - x \leqslant y \leqslant c)$. Symmetric information can be derived for conjunction. Exclusive disjunction, however, does not derive knowledge regarding deltas. The bitwise negation has limited effect and can only impose that its argument and its result be different.

Extraction: regardless of the indices, the result of an extraction is always less than or equal to its result. As a matter of fact, an extraction $\lfloor x \rceil_{i.j} = (x \% 2^i)/2^j$ and can enjoy the same propagations on the non-BV domains.

More generally: many BV constraints can be mapped to an integer counterpart and propagate on non-BV domains. For example, a concatenation $x :: y$ can have the same effect as the (overflowless) integer constraint $z = x \times 2^{\|y\|} + y$ would.

4.3 Factorizations and simplifications

In the course of solving, a constraint can be simplified or become duplicate or a subsumption of another constraint. These shortcuts can be separated in two categories.

Simplifications. Neutral and absorbing elements provide many rewriting rules which replace constraints by simpler ones. In addition to these usual simplifications one can detect more sophisticated ones, such as if $z \gg y = z$ and $y > 0$ then $z = 0$ (without restricting the value of y), and when z is x rotated i times, if the size of x is 1 or if $i \% \|x\| = 0$, then $z = x$. Furthermore, if i and $\|x\|$ are coprimes, and $x = z$, then $x = 0$ or $x = 2^{\|x\|} - 1$.

Factorizations. The more constraints are merged or reduced to simpler constraints, the closer we get to a proof. Functional factorization allows to detect instances based on equality of arguments, but some other instances can be factored as well, for example:

- from $x \oplus y = z$ and $x \oplus t = y$, we deduce that $t = z$, unifying the two variables and removing one of the constraints, now considered duplicates
- when $x \ll y_1 = z_1$ and $x \ll y_2 = z_2$ and $y_1 < y_2$, and z_1 is a constant, then one can infer the value of z_2. A similar operation can be carried out for $\gg$.

- two constraints $x \,\&\, y = 0$ and $x \mid y = 2^{\|x\|} - 1$ can be replaced by $x = \overline{y}$
- a constraint $x \,\&\, y = z$ (resp. $x \mid y = z$) is superfluous with the constraint $x = \overline{y}$ once z is deducted to be equal to 0 (resp. $2^{\|x\|} - 1$).
- the constraints $x = \overline{y}$ and $x \,\&\, z = y$ (resp. $x \mid z = y$) can both be removed once deducted that $x = 2^{\|x\|} - 1, y = z = 0$ (resp. $x = 0, y = z = 2^{\|x\|} - 1$).

4.4 BV-dedicated labeling strategies

A labeling strategy (a.k.a. search strategy) consists mainly of two heuristics: *variable selection* and *value selection*. For variable selection, we rely on the fail-first approach implemented in COLIBRI [33]. Basically, the variable to be selected is the one with the smallest domain (in terms of concretization). Adding the BV domain allows here to refine the notion of smallest. For value selection, in the event that BV is the smallest domain, our strategy is the following:

- First, we consider certain values that can simplify arithmetic constraints. In particular, we start by trying 0 (for $+, -, \times, /,\ \&\ , |, \oplus$), 1 (for $\times, /$) and $2^s - 1$ where s is the bitvector size (for $\&\ , |$);
- Second, we fix the value of several most significant and least significant unknown bits *(msub, lsub)* at once, allowing to strongly refine all domains thanks to inter-reduction, and to fix early the sign of the labeled variable (useful for signed BV operations). Currently, we fix at each labeling step one *msub* and two *lsub*, yielding 8 possible choices. We choose whether to set first or clear first in an arbitrary (but deterministic) way, using a fixed seed.

5 Experimentation

We describe in this section our implementation and experimental evaluation of CP(BV).

Implementation. We have implemented a BV support inside the COLIBRI CP solver [33] (cf. Section 3.3). Modular arithmetic domains and propagators are treated as blackbox, and we add the optimized bitlist domain and its associated propagators from [36], as well as all improvements discussed in Section 4. Building on top of COLBRI did allow us to prototype our approach very quickly, compared with starting from scratch.

Because it is written in a Prolog dialect, the software must be interpreted at each run, inducing a systematic 0.2 second starting time. This obstacle is not troubling for us because any real-world application would execute our software once and feed its queries in a chained manner through a server mode. Yet, the SMTCOMP rules impose that the software be called on command line with exactly one .smt2 file, which excludes a "server mode".

Experimental setup and caveats. We experiment CP(BV) on the 32k BV-formulas from the SMTCOMP benchmark, the leading competition of the SMT community. These formulas are mostly taken from verification-oriented industrial case-studies, and can be very large (up to several hundreds of MB). The first set of experiments (Section 5.1) has been run on the StarExec server[3] provided by SMTCOMP, they are made public [4]. The second set of experiments (Section 5.2) is run on a Intel® Core^{TM} i7-4712HQ CPU @ 2.30GHz with 16GB memory. Two points must be kept in mind.

- We fix a low time-out (60s) compared with the SMTCOMP rules (40 min), yet we argue that our results are still representative: first, such low time-outs are indeed very common in applications such as bug finding [25] or proof [19]; second, it is a common knowledge in first-order decision procedures that *"solvers either terminate instantly, or timeout"* – adding more time does not dramatically increase the number of solved formulas;
- SMT solvers such as Z3 and CVC4 are written in efficient compiled languages such as C/C++, with near-zero starting time. Hence, we have a constant-time disadvantage here – even if such a burden may not be so important in verification: since we are anyway attacking NP-hard problems, we are looking for exponential algorithmic improvements ; constant-time gains can only help marginally.

5.1 Evaluation against state-of-the-art benchmark

Absolute performance. Our implementation solved 24k formula out of 32k (75%). While it would not have permitted us to win the SMTCOMP, it is still a *significantly more thorough comparison with the SMT community than any previous CP effort on bitvectors*, demonstrating that CP can indeed be used for verification and bitvectors.

Comparing different choices of implementation. Improvements offered by our different optimizations are very dependent on the type of formulas, and these details would be diluted if regrouping all of the benchmarks. The reader is invited to consult our detailed results on the StarExec platform. Yet, as a rule of thumb, *our extensions yield a significant improvement on some families of examples, and do not incur any overhead on the other, proving their overall utility.* For example :

- On the family of formulas named `stp_samples`($\simeq$ 400 formulas), when deactivating the reductions from BV constraints to other domains (Section 4.2), the solver is unable to solve *a quarter less* formulas that it did with the full implementation. Removing the interreduction with the *Is* (Section 4.1), the loss rises to *half*;
- Solving the `spear` family suffers little from deactivating BV/*Is* inter-reductions, but *half* (200) formulas are lost without the BV constraints reducing other domains (Section 4.2);

[3] `www.starexec.org`

[4] `www.starexec.org/starexec/secure/explore/spaces.jsp?id=186070`

- On some other families, such as `pspace` and `dwp_formulas`, there is no tangible effect (neither positive nor negative) to the deactivation of improvements.

5.2 Comparison to state-of-the-art SMT solvers

We demonstrate in the following that CP(BV) is actually *complementary* to SMT approaches, especially on problems with large bitvectors or involving multi-theories. As competitors, we select the five best SMT solvers for BV theory: Z3, Yices, MathSAT, CVC4 and Boolector.

SMTCOMP: large formulas. To study the effect of our method on *scalability*, we show here the results on three categories of formulas, regrouped according to the (number of digits of the) size of the their largest bit-vectors: 3-digit (from 100 to 999), 4-digit and 5-digit, respectively having 629, 298 and 132 formulas. Results in Table 3 show on the one hand the *scalability* of CP(BV) – the larger BV sizes, the greater impact the CP approach has) – and on the other hand its *complementarity* with SMT. In particular, it shows the result of *duels* between CP(BV) and each of the SMT solvers : a formula is considered a win for a solver if it succeeds (TO = 60 seconds) while the other solver does not. We report results on a format Win/Lose (Solve), where Win and Lose are from CP(BV) point of view, and Solve indicates the number of formulas solved by the SMT solver. For example, MathSAT could solve 17 of the 132 5-digit formulas – all of which being solved by CP(BV), while CP(BV) could solve 63 formulas – 46 of which were unsolved by MathSAT. Here, *CP(BV) solves the higher number of 5-digit size formulas (equality with CVC4), and no solver but Boolector solves formulas that CP(BV) does not. On other sizes, CP(BV) solves less formulas, but it can still solve formulas that SMT solvers do not.*

Table 3: Comparing CP(BV) with five state-of-the-art solvers on large formulas

sz	#f	CP(BV) #solved	Z3 w/l (s)	Yices w/l (s)	MathSAT w/l (s)	CVC4 w/l (s)	Boolector w/l (s)
5	132	63	63/0 (0)	53/0 (10)	46/0 (17)	0/0 (63)	32/10 (41)
4	298	44	34/153 (163)	40/87 (91)	43/68 (69)	42/150 (152)	43/204 (205)
3	629	35	24/496 (507)	23/262 (274)	23/419 (431)	23/511 (523)	25/507 (517)

sz: size (#digits) - #f: # of formulas
w/l (s): #win/#lose for CP(BV), s: #formulas solved by SMT solver

SMTCOMP: hard formulas. We define *hard formulas* by separating 5 classes of difficulty, with class i regrouping the formulas on which i SMT solvers out of 5 spend more than 5 seconds. We compare CP(BV) to SMT solvers on these hard problems. Results are presented in Table 4, where we report for each class i the number of formulas from this class that CP(BV) solves quickly. Especially, the 5^{th} column (All-fail) shows that 61 formulas are solved only by CP(BV) in less than 5 seconds.

Mixing bitvectors and floats. Considering multi-theory formulas combining BV and FP arithmetics, COLIBRI has been tested on 7525 industrially-provided formulas (not

Table 4: Overall comparison on *hard examples*

Category	1-fail	2-fail	3-fail	4-fail	All-fail
#benchs	1083	382	338	1075	873
CP(BV) under 5s	139	108	10	68	61

publically available). It was able to solve 73% of them, standing half-way between Z3 / MathSAT and CVC4 (Table 5). Considering now the last SMTCOMP QF_BVFP category (Table 6), *even with the 0.2 seconds starting time, CP(BV) would have won the competition* – admittedly, there are only few formulas in this category.

Table 5: Industrial formula with bitvectors and floats

	CP(BV)	Z3	MathSAT	CVC4
#solved	5512	7225	7248	2245
ratio	73%	96%	96%	29%

Total: 7525 formulas

Table 6: SMTCOMP, QF_BVFP category

	Z3	MathSAT	CP(BV)
int_to_float_complex_2.smt2	1.04	0.13	0.25
int_to_float_simple_2.smt2	2.17	0.22	0.21
int_to_float_complex_1.smt2	0.95	0.08	0.25
int_to_float_simple_1.smt2	0.02	0.02	0.25
nan_1.smt2	0	0	0.26
incr_by_const.smt2	8.20	30.50	0.26
int_to_float_complex_3.smt2	1.89	0.44	0.25
quake3_1.smt2	TO	TO	TO

6 Related work

CP-based methods for BV. This work strongly stands upon the prior results of Bardin *et al.* [1] and Michel *et al.* [36]. Our respective contribution is already discussed at length in Sections 2 and 3. Basically, while we reuse the same general ideas, we sharpen them through a careful selection of the best aspects of each of these works and the design of new mechanisms, especially in terms of domain combination and labeling strategies. As a result, experiments in Section 5 demonstrate that our own CP(BV) approach performs much better than previous attempts. Moreover, we perform an extensive comparison with SMT solvers on the whole SMTCOMP benchmark, while these previous efforts were either limited to conjunctive formulas or remain only theoretical. The results by Michel *et al.* have been applied to a certain extent [46] as an extension of MiniSat [21], yet with no reduced product, to a limited set of BV operations and on BV sizes no larger than 64 bits. Older word-level approaches consider straightforward translations of

bit-vector problems into disjunctive or non-linear arithmetic problems [10, 22, 40, 45, 48, 49] (including bitblasting-like transformation for logical bitwise operators), and then rely on standard methods from linear integer programming or CP. Experimental results reported in [44, 1] demonstrate that such straightforward word-level encoding yield only very poor results on formulas coming from software verification problems.

SMT-based methods for BV. While state-of-the-art methods heavily rely on bit-blasting and modern DPLL-style SAT solvers [38, 43], the community is sensing the need for levels of abstraction "where structural information is not blasted to bits"[12]. Part of that need comes from the knowledge that certain areas, arithmetic for example, are not efficiently handled by bit-level reasoning tools. As a mitigation, SMT solvers typically complement optimized bitblasting [12, 13, 32] with word-level preprocessing [3, 24]. Compared to these approaches, we lack the highly-efficient learning mechanisms from DPLL. Yet, our domains and propagations yield more advanced simplifications, deeply nested with the search mechanism.

7 Conclusion

This work addresses the challenge of developing efficient Constraint Programming-based approaches for solving formulas over (the quantifier-free fragment of) the theory of bitvectors, which is of paramount importance in software verification. While the Formal Verification community relies essentially on the paradigm of SMT solving and reduction to Boolean satisfiability, we explore an alternative, high-level resolution technique through dedicated CP principles. We build on a few such anterior results and sharpen them in order to propose a highly efficient CP(BV) resolution method. We show that CP(BV) is much more efficient than the previous attempts from the CP community and that it is indeed able to solve the majority of the standard verification benchmarks for bitvectors. Moreover CP(BV) already complements the standard SMT approach on several crucial (and industry-relevant) aspects, such as scalability w.r.t. bit-width, formulas combining bitvectors with bounded integers or floating-point arithmetic, and formulas deeply combining non-linear arithmetic and bitwise operators.

Considering that our current CP(BV) approach is far from optimal compared with existing SMT solvers, we believe this work to be an important landmark toward building competitive CP-based verification-oriented solvers. Moreover, our approach clearly challenges the well-accepted belief that bitvector solving is better done through bitblasting, opening the way for a new generation of word-level solvers.

References

[1] S. Bardin, P. Herrmann, and F. Perroud. "An Alternative to SAT-Based Approaches for Bit-Vectors". In: *TACAS*. 2010.

[2] S. Bardin and P. Herrmann. "OSMOSE: Automatic Structural Testing of Executables". In: *Softw. Test. Verif. Reliab.* 21.1 (2011).

[3] C. Barret, D. Dill, and J. Levitt. "A decision procedure for bit-vector arithmetic". In: *DAC 98*. 1998.

[4] C. Barrett et al. "CVC4". In: *CAV*. 2011.

[5] C. Barrett et al. *The SMT-LIB Standard: Version 2.0*. Tech. rep. 2010.

[6] C. W. Barrett et al. "Satisfiability Modulo Theories". In: *Handbook of Satisfiability*. 2009.

[7] A. Biere et al. "Symbolic Model Checking without BDDs". In: *Tools and Algorithms for the Construction and Analysis of Systems*. 1999.

[8] N. Bjørner. "Taking Satisfiability to the Next Level with Z3". In: *Proceedings of the 6th International Joint Conference on Automated Reasoning*. 2012.

[9] B. Blanc et al. "Handling State-Machines Specifications with GATeL". In: *Electr. Notes Theor. Comput. Sci.* 264.3 (2010).

[10] R. Brinkmann and R. Drechsler. "RTL-datapath verification using integer linear programming". In: *15th Int. Conf. on VLSI Design*. 2002.

[11] R. Brummayer and A. Biere. "Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays". In: *TACAS*. 2009.

[12] R. Bruttomesso et al. "A Lazy and Layered SMT(BV) Solver for Hard Industrial Verification Problems". In: *Computer Aided Verification (CAV)*. 2007.

[13] R. E. Bryant et al. "Deciding Bit-Vector Arithmetic with Abstraction". In: *Tools and Algorithms for the Construction and Analysis of Systems TACAS*. 2007.

[14] A. Cimatti et al. "The MathSAT5 SMT Solver". In: *TACAS*. 2013.

[15] E. M. Clarke, D. Kroening, and F. Lerda. "A Tool for Checking ANSI-C Programs". In: *TACAS*. 2004.

[16] H. Collavizza, M. Rueher, and P. Hentenryck. "CPBPV: A Constraint-Programming Framework for Bounded Program Verification". In: *CP2008*. 2008.

[17] R. David et al. "BINSEC/SE: A Dynamic Symbolic Execution Toolkit for Binary-Level Analysis". In: *SANER 2016*. 2016.

[18] R. Dechter. *Constraint Processing*. Morgan Kaufmann Publishers Inc., 2003.

[19] E. W. Dijkstra. *A discipline of programming*. Vol. 1. Prentice-Hall Englewood Cliffs, 1976.

[20] B. Dutertre. "Yices 2.2". In: *Computer-Aided Verification (CAV)*. Vol. 8559. 2014.

[21] N. Eén and N. Sörensson. “An Extensible SAT-solver”. In: *Theory and Applications of Satisfiability Testing (SAT)*. 2003.

[22] F. Ferrandi, M. Rendine, and D. Sciuto. “Functional verification for SystemC descriptions using constraint solving”. In: *Design, Automation and Test in Europe*. 2002.

[23] T. Feydy, A. Schutt, and P. J. Stuckey. “Global Difference Constraint Propagation for Finite Domain Solvers”. In: *PPDP*. 2008.

[24] V. Ganesh and D. L. Dill. “A Decision Procedure for Bit-Vectors and Arrays”. In: *CAV 2007*. 2007.

[25] P. Godefroid. “Test Generation Using Symbolic Execution”. In: *FSTTCS*. 2012.

[26] A. Gotlieb. “TCAS software verification using Constraint Programming”. In: *Knowledge Engineering Review* 27.3 (2012).

[27] A. Gotlieb, B. Botella, and M. Rueher. “Automatic Test Data Generation Using Constraint Solving Techniques”. In: *ISSTA*. 1998.

[28] A. Gotlieb, M. Leconte, and B. Marre. “Constraint Solving on Modular Integers”. In: 2010.

[29] T. A. Henzinger et al. “Lazy abstraction”. In: *POPL*. 2002.

[30] D. Kroening and O. Strichman. *Decision Procedures: An Algorithmic Point of View*. 1st ed. Springer Publishing Company, Incorporated, 2008.

[31] M. Leconte and B. Berstel. “Extending a CP Solver with Congruences as Domains for Program Verification”. In: *Trends in Constraint Programming*. 2010.

[32] P. Manolios and D. Vroon. “Efficient circuit to CNF conversion”. In: *SAT 2007*.

[33] B. Marre and B. Blanc. “Test Selection Strategies for Lustre Descriptions in GaTeL”. In: *ENTCS*. Vol. 111. 2005.

[34] B. Marre and C. Michel. “Improving the Floating Point Addition and Subtraction Constraints”. In: *Principles and Practice of Constraint Programming - CP 2010*. 2010.

[35] K. L. McMillan. “Lazy Abstraction with Interpolants”. In: *CAV*. 2006.

[36] L. D. Michel and P. Van Hentenryck. “Constraint Satisfaction over Bit-Vectors”. English. In: *Principles and Practice of Constraint Programming*. Vol. 7514. 2012.

[37] R. A. D. Millo, R. J. Lipton, and A. J. Perlis. “Social Processes and Proofs of Theorems and Programs”. In: *Communications of the Association of Computing Machinery* 22.5 (1979).

[38] M. W. Moskewicz et al. “Chaff: Engineering an Efficient SAT Solver”. In: *Design Automation Conference, DAC*. 2001.

[39] G. Nelson and D. C. Oppen. “Simplification by Cooperating Decision Procedures”. In: *ACM Trans. Program. Lang. Syst.* 1.2 (1979).

[40] G. Parthasarathy et al. “An efficient finite-domain constraint solver for circuits”. In: *41th Design Automation Conf.* 2004.

[41] M. Pelleau et al. “A Constraint Solver Based on Abstract Domains”. In: *VMCAI 2013*. 2013.

[42] J. D. Scott, P. Flener, and J. Pearson. “Bounded Strings for Constraint Programming”. In: *ICTAI*. 2013.

[43] J. P. M. Silva and K. A. Sakallah. “GRASP: A Search Algorithm for Propositional Satisfiability”. In: *IEEE Trans. Computers* 48.5 (1999).

[44] A. Sülflow et al. “Evaluation of SAT like proof techniques for formal verification of word level circuits”. In: *8th IEEE Workshop on RTL and High Level Testing*. 2007.

[45] R. Vemuri and R. Kalyanaraman. “Generation of design verification tests from behavioral VHDL programs using path enumeration and constraint programming”. In: *IEEE Transactions on VLSI Systems, 3(2), pp. 201-214*. 1995.

[46] W. Wang, H. Søndergaard, and P. J. Stuckey. “A Bit-Vector Solver with Word-Level Propagation”. In: *CPAIOR*. 2016.

[47] N. Williams, B. Marre, and P. Mouy. “On-the-Fly Generation of K-Path Tests for C Functions”. In: *ASE 2004*. 2004.

[48] Z. Zeng, M. Ciesielski, and B. Rouzeyre. “Functional test generation using Constraint Logic Programming”. In: *11th Int. Conf. on Very Large Scale Integration of Systems-on-Chip*. 2001.

[49] Z. Zeng, P. Kalla, and M. Ciesielski. “LPSAT: a unified approach to RTL satisfiability”. In: *4th Conf. on Design, Automation and Test in Europe*. 2001.