Improving Bit-Blasting for Nonlinear Integer Constraints

Fuqi Jia
SKLCS, ISCAS
University of Chinese Academy of Sciences
Beijing, China
jiafq@ios.ac.cn

Rui Han
SKLCS, ISCAS
University of Chinese Academy of Sciences
Beijing, China
hanrui@ios.ac.cn

Pei Huang
Stanford University
Stanford, USA
huangpei@stanford.edu

Minghao Liu
SKLCS, ISCAS
University of Chinese Academy of Sciences
Beijing, China
Liumh@ios.ac.cn

Feifei Ma*
SKLCS, LPSCS, ISCAS
University of Chinese Academy of Sciences
Beijing, China
maff@ios.ac.cn

Jian Zhang*
SKLCS, ISCAS
University of Chinese Academy of Sciences
Beijing, China
zj@ios.ac.cn

ABSTRACT

Nonlinear integer constraints are common and difficult in the verification and analysis of software/hardware. SMT(QF_NIA) generalizes such constraints, which is a boolean combination of nonlinear integer arithmetic constraints. A classical method to solve SMT(QF_NIA) is bit-blasting, which reduces them to boolean satisfiability problems. Currently, the existing pure bit-blasting based solvers are noncompetitive with other state-of-the-art SMT solvers. The bit-blasting based methods have some problems: First, the bit-blasting method is hampered by nonlinear multiplication operations; second, it sometimes does not search in a proper search space; and third, it contains some redundancy.

In this paper, we focus on improving the efficiency of bit-blasting based method. To decide on a proper search space, we proposed an adaptive function for hard nonlinear multiplications, and heuristic strategies to analyze specific constraints. We also found that different orders in successive additions will result in bit vectors with different bit-widths. We proposed an optimal order decision algorithm to save redundancy in successive additions. We implement a solver with the proposed methods named BLAN. Experiments demonstrate that BLAN outperforms other state-of-the-art SMT solvers (APROVE, CVC5, MATHSAT, YICES2, Z3) on the satisfiable SMT(QF_NIA) instances in SMT-LIB. We provide an outlook of BLAN on solving unsatisfiable instances via combining with other solvers. Sensitivity analysis also demonstrates the effectiveness of the proposed methods.

*Corresponding Authors.

ISSTA '23, July 17–21, 2023, Seattle, WA, USA

ACM ISBN 979-8-4007-0221-1/23/07…$15.00
https://doi.org/10.1145/3597926.3598034

CCS CONCEPTS

• **Theory of computation → Logic and verification; Automated reasoning**; • **Mathematics of computing → Solvers**.

KEYWORDS

nonlinear integer constraints, satisfiability modulo theories

ACM Reference Format:
Fuqi Jia, Rui Han, Pei Huang, Minghao Liu, Feifei Ma, and Jian Zhang. 2023. Improving Bit-Blasting for Nonlinear Integer Constraints. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23), July 17–21, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597926.3598034

1 INTRODUCTION

Nonlinear arithmetic problems over integers have attracted the attention of researchers since ancient times. In the 3rd century, Greek mathematician Diophantine studied a class of equations involving only sums, products, and powers in which all the constants are integers. When the only solutions of interest are also restricted to integers, they are known as Diophantine equations. A famous particular case is Fermat's last theorem, which puzzled people for more than three hundred years until it was resolved in 1993 by Andrew Wiles. In 1900, Hilbert asked for an algorithm for determining whether an arbitrary Diophantine equation has a solution, known as Hilbert's 10th problem. Unfortunately, the impossibility was proven by Yuri Matiyasevich [41].

In this paper, we focus on a general form of nonlinear integer arithmetic: satisfiability modulo the logic fragment of nonlinear integer arithmetic theory (SMT(QF_NIA)). It is the boolean combination (using logic operators: and, or, not) of nonlinear integer arithmetic constraints, involving equations and inequalities. As they often appear in software/hardware verification and analysis, a practical algorithm for the problem is still highly desirable. For verification, many important tasks involve nonlinear integer theory, for example, non-linear loop invariant generation [45], state reachability [17, 27]; For symbolic execution, nonlinear integer theory is expensive and remains one of the main obstacles to the scalability of symbolic execution engines [2, 43]; For test case generation, especially finding targeted test input, SMT solvers play a key role

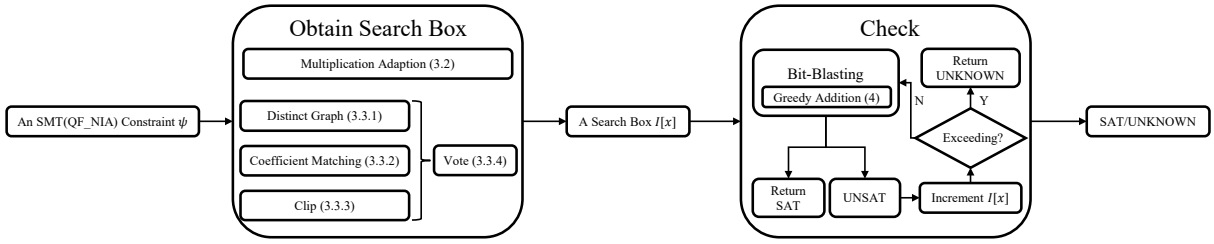


Figure 1: The schematic diagram of the architecture of BLAN. The content in brackets refers to the corresponding section.

in finding a solution for path condition which sometimes contains complicated non-linear arithmetic [21, 44]. All such tasks require an efficient solver for nonlinear integer constraints.

With the advancement of SAT solving techniques, SAT solvers are employed in the decision procedures for some SMT theories. The basic idea is to replace the arithmetic operators with circuit equivalence to obtain a propositional formula, which is then converted into Conjunctive Normal Form (CNF) and given to a SAT solver. Note that the domains of the variables in the original SMT constraint are presumed to be finite. This approach is commonly referred to as 'bit-blasting' [38]. With the help of powerful modern SAT solvers, bit-blasting can find a solution or prove that there is no solution in the given finite domains.

Technically, bit-blasting can be summarized in two categories: fixed-size or flexible-size with overflow bits. The boolean combination of constraints on fixed-size bit-vectors is also known as SMT(QF_BV) [4, 38]. It discards the overflowing bit of the resulting bit vector, which is equivalent to doing a modulo operation. Flexible-size bit-blasting [26, 50] will always store the overflow bits so that the bit-width of the result of the operation is not fixed. In this paper, we focus on the flexible-size bit-blasting.

Due to the inefficiency (the vast transformed CNF size and the propagation incompleteness) of multiplication in bit-blasting[10], there is also a doubt about whether bit-blasting is the right path to solve constraints with plenty of multiplications. Experiments have indicated that the existing pure bit-blasting based solvers like APROVE [26] are noncompetitive with other state-of-the-art SMT solvers even on satisfiable instances. Considering the implementation of storing the overflow bit, multiple additions will result in more intermediate boolean variables and clauses than that of SMT(QF_BV), especially in the case of successive addition. Whether there is redundancy is a question.

To improve the efficiency of bit-blasting based methods, deciding on a proper search space can be valuable before bit-blasting and solving. The desired "proper search space" is a set of as small as possible domains of unknowns while covering at least one solution for satisfied instances. It will lead bit-blasting to a shortcut that significantly relieves the burden on the backend SAT solver. However, obtaining such a proper search space is undecidable. Assume that there is an algorithm that always answers the best finite search space in which there must be a solution for satisfiable instances and random or empty space for unsatisfiable instances. Then invoking a bit-blasting based or enumeration based procedure will report the answer of all kinds of SMT(QF_NIA) problems, which also contradicts Matiyasevic's results [41].

In summary, there are three problems we focus on to improve the bit-blasting based method:

- How to extract a search space to alleviate the impact of multiplication?
- How to extract a search space from specific constraints?
- How to optimize bit-blasting within successive addition?

In this paper, we propose a list of methods to solve the problems. Initial Bit-Width decision reasons about the scope of search space, including Multiplication Adaptation and Vote strategies that initially determine a proper search space. Considering the inefficiency of multiplication, the Multiplication Adaptation strategy adapts bit-width with multiplications. By analyzing different kinds of constraints, we give three strategies to decide on an initial search space and Vote for a proper search space finally. Greedy Addition is a proven efficient addition algorithm to save boolean variables and clauses as well as running time in successive addition. We implement a pure bit-blasting based solver equipped with such methods named BLAN, and a schematic diagram is shown in Figure 1. We compare BLAN with the state-of-the-art SMT solvers (APROVE, CVC5, MATHSAT, YICES2, Z3) on SMT(QF_NIA) benchmarks. The results show a significant advantage in the number of solved instances and speed. On satisfiable instances, BLAN solves 6.44% to 34.81% more instances, and the solving speed is 2.02x - 11.11x faster on all solved instances. On 5212 unknown instances, BLAN finds feasible solutions for 114 new instances including 80 unique instances that can only be solved by BLAN, while other solvers can solve at most 29 instances. It also indicates that bit-blasting is still a promising approach to solving satisfiable instances. Each solver can also be improved on solving unsatisfiable instances when combined with BLAN. Particularly, MATHSAT+BLAN achieves the best result and YICES2+BLAN achieves the largest increase.

2 BACKGROUND

In this section, we introduce the syntax and semantics of the SMT logic, SMT(QF_NIA), bit-blasting, and bit-blasting based methods. We also introduce a search box to represent search space and indicate the relation between bit-width and the search box.

2.1 Syntax and Semantics of SMT(QF_NIA)

The set of the integers and the set of natural numbers are denoted as $\mathbb{Z}$ and $\mathbb{N}$, respectively. The first-order logic structure $\mathcal{S}$ of SMT(QF_NIA) consists of

- constants: $\top$ (true), $\bot$ (false), $\mathbb{Z}$;

- variables: integer variables $\mathcal{V}_{int}$, logical variables $\mathcal{V}_{bool}$;
- functions: $\mathcal{O} = \{+, -, \cdot, \div\}$;
- predicates: $\mathcal{R} = \{>, \geq, <, \leq, =, \neq\}$;
- logical connectives: $\wedge, \vee, \neg$.

The definition of polynomials is discussed as follows. Given an n-dimensional vector of integer variables $\boldsymbol{x} = \{x_1, x_2, \cdots, x_n\}$ and an n-dimensional vector of natural numbers $\boldsymbol{d}$ as the exponent vector, a *monomial* denotes a formula of the form $\boldsymbol{x}^{\boldsymbol{d}} = x_1^{d_1} \cdot x_2^{d_2} \cdot \ldots \cdot x_n^{d_n}$.

A term is of the form $c \cdot \boldsymbol{x}^{\boldsymbol{d}}$, where $c \in \mathbb{Z}$. A polynomial $f \in \mathbb{Z}[\boldsymbol{x}]$ is a finite sum of terms i.e., $f = \sum_i c_i \cdot \boldsymbol{x}^{\boldsymbol{D}_i}$ where $\boldsymbol{D}$ is a list of exponent vectors and $\boldsymbol{D}_i$ is the i-th exponent vector. A polynomial constraint is of the form $f \odot 0$, where $\odot \in \mathcal{R}$. The SMT(QF_NIA) constraint is formed by connecting polynomial constraints and boolean combinations with logical connectives. Note that when each exponent vector is a one-hot vector where only one element is 1 and the rest are 0, an SMT(QF_NIA) constraint degenerates into an SMT(QF_LIA) constraint (linear integer arithmetic).

In semantics, an assignment α is a mapping from variable symbols to the integers and boolean value, i.e., $\alpha = \alpha_{bool} \cup \alpha_{int}$, where $\alpha_{bool} : \mathcal{V}_{bool} \to \{\top, \bot\}$ where $\top$ for tautology and $\bot$ for contradiction and $\alpha_{int} : \mathcal{V}_{int} \to \mathbb{Z}$. An SMT(QF_NIA) constraint C is given the truth value $\top$, if and only if $\mathcal{S}, \alpha \models C$.

2.2 Bit-Blasting

Bit-blasting transforms integer variables with finite domains into bit vectors and arithmetic/comparison operations into propositional constraints. Typically, the bit vector is a vector of $\{\bot, \top\}$. Without loss of generality, the following content mainly focuses on signed bit vectors implemented by the two's complement encoding since such encoding is employed by most microprocessor architectures [38]. For an integer variable x, its bit vector representation with bit-width w is denoted as $\overline{x} : \langle \overline{x}_{w-1}, \cdots, \overline{x}_1, \overline{x}_0 \rangle$. Here we overline the bolded variable to distinguish it from the vector of integer variables. $\overline{x}_{w-1}$ is the most significant bit, which also denotes the sign bit. If $\overline{x}_{w-1} = \top$ then $\overline{x}$ is negative, otherwise if $\overline{x}_{w-1} = \bot$ then $\overline{x}$ is non-negative. Suppose we bit-blast all variables and operations into bit vectors. In that case, an integer arithmetic satisfiability problem is reduced into a boolean satisfiability problem, and SAT solvers can solve the transformed problem. We refer the reader to paper [50] for more details on transformation, which covers the arithmetic operations (addition, subtraction, multiplication, division) and comparison operations (greater, greater equal, equal) on bit vectors.

The value of a bit vector can be calculated as

$$val(\overline{x}) = -2^{w-1} \times Int(\overline{x}_{w-1}) + 2^{w-2} \times Int(\overline{x}_{w-2}) + \cdots + \overline{x}_0,$$

where $val(\overline{x})$ is the integer value of the bit vector $\overline{x}$ and $Int(\overline{x}_i)$ is a map function that converts a boolean value $\top$ or $\bot$ into an integer 1 or 0 respectively. For example, $val(\langle \top, \bot, \top \rangle) = -3$.

The bit-blasting used in SMT(QF_BV) is different from that of this paper, since operations on fixed-sized bit vectors are not complete on $\mathbb{Z}$, as is shown in Example 2.1.

Example 2.1. Consider a constraint $x + y = z$ and bit-blast each variable to a 2-bit bit vector $\overline{x}, \overline{y}$ where $val(\overline{x}), val(\overline{y}) \in [-2, 1]$. In SMT(QF_BV), adding them up will result in a 2-bit bit vector $\overline{z}$ where $val(\overline{z}) \in [-2, 1]$ without considering the case of overflow, i.e. $\overline{z} = (\overline{x} + \overline{y}) \ mod \ 4$, for example,

$$-2 + (-2) = \langle 1, 0 \rangle + \langle 1, 0 \rangle = \langle 0, 0 \rangle = 0.$$

It is incomplete in SMT(QF_NIA), while we can utilize 3-bit bit vector $\overline{z}$ which is sufficient for overflow situations,

$$-2 + (-2) = \langle 1, 0 \rangle + \langle 1, 0 \rangle = \langle 1, 0, 0 \rangle = -4.$$

Definition 2.2. Given a set of integer variables $\mathcal{V}_{int}$, the search box I of $\mathcal{V}_{int}$ is a function mapping variables in $\mathcal{V}_{int}$ to finite closed intervals, whose lower bound and upper bound are both integers. The search box of variable x is denoted as $I[x]$.

The search box defines the search space where the bit-blasting method searches. The bit-width of an integer variable and the search box can be converted to each other. Given a bit-width $w \in \mathbb{N}$, we can obtain a search box from a simple function,

$$RANGE(w) = [-2^{w-1}, 2^{w-1} - 1].$$

An integer variable y with a search box corresponds to a bit vector with an additional constraint. Given $I[y] = [a, b]$, where $a, b \in \mathbf{Z}$, we obtain

$$\overline{\boldsymbol{y}} = \overline{\boldsymbol{x}} + a + 2^{w-1}, val(\overline{\boldsymbol{y}}) \leq b,$$

where the bit-width of $\overline{\boldsymbol{x}}$ is $w = \lceil log_2(b - a + 1) \rceil$ corresponding to a range of $RANGE(w)$.

Example 2.3. If a variable y corresponds to a search box $I[y] = [1, 8]$, we can bit-blast the variable y to a bit vector $\overline{\boldsymbol{y}} = \overline{\boldsymbol{x}} + 5$ where the bit-width of $\overline{\boldsymbol{x}}$ is 3.

2.3 Bit-Blasting Based Method

Bit-Blasting Based Methods will invoke a procedure that bit-blasts the SMT(QF_NIA) constraint C into corresponding CNF formulae, and then solve them via an SAT solver. For simplification, $BB(C, W)$ denotes such a procedure. The bit-width setting W maps an integer variable or a bit vector to its corresponding bit-width. Note that it is not able to change W inside the process of $BB(C, W)$. We can only use another bit-width setting via restarting. So we need to determine the bit width setting W at first. Bit-Blasting Based Methods will either invoke BB one time with a sufficiently large bit-width setting or restart BB many times with an incremental bit-width setting. The latter case is the most common [3, 29] because running with a large bit-width is really difficult for the backend SAT solver.

Example 2.4. Consider a process of the bit-blasting method, $BB(\{x + 1 \geq 0\}, \{W(x) = 2\}) = \top$. Firstly it bit-blasts the variable x to a (boolean) bit vector $\overline{x} : \langle \overline{x}_1, \overline{x}_0 \rangle$ of 2 bits, where $\overline{x}_1$ is the sign bit, and adding $\overline{x}$ with one results in a new bit vector $\overline{\boldsymbol{y}} = \langle \neg\overline{\boldsymbol{x}}_0 \wedge \overline{\boldsymbol{x}}_1, \overline{\boldsymbol{x}}_0 \oplus \overline{\boldsymbol{x}}_1, \neg\overline{\boldsymbol{x}}_0 \rangle$ of 3 bits with an overflow bit. Next the constraint $\overline{\boldsymbol{y}} \geq 0$ asserts that $\overline{\boldsymbol{y}}$ is positive i.e., the sign bit $\overline{\boldsymbol{y}}_2 = \bot$. Finally we add the transformed assertion $\overline{\boldsymbol{x}}_0 \vee \neg\overline{\boldsymbol{x}}_1$ to the SAT solver and obtain $\top$. Additionally, assume the SAT solver finds a solution $\{\overline{\boldsymbol{x}}_1 = \bot, \overline{\boldsymbol{x}}_0 = \bot\}$ which also indicates a satisfiable assignment of $x + 1 \geq 0$, i.e., $x = 0$.

Beyond Example 2.4, if $BB(C, W)$ returns $\bot$, Bit-Blasting Based Method will increment the bit-widths in W, for example, $W' = \{W(x) \times 2 | x \in \mathcal{V}_{int}(C)\}$ where $\mathcal{V}_{int}(C)$ denotes the set of integer variables from the constraint C. Next, $BB(C, W')$ is invoked to

check the satisfiability of C. The procedure will last until a feasible solution is found or running out of resources.

3 INITIAL BIT-WIDTH DECISION

In this section, we concentrate on how to decide on a proper search box in different situations.

3.1 Motivation

3.1.1 Motivating Example. Consider a constraint,

$$x^2 - 16x + 100000000 \geq 0 \wedge y^2 - 4y \leq 0.$$

It has a trivial solution $\{x = 0, y = 0\}$. Z3 will set the bit-widths via the maximum constant in the constraint, so the bit-widths of $\overline{x}$ and $\overline{y}$ will be 28. Actually, Z3 (the same setting of Section 6) spends more than 1 minute solving it, because the large bit-width setting burdens the backend SAT solver. At the same time, if Z3 utilizes two 2-bit bit vectors, it will solve the constraint in 0.024 seconds.

3.1.2 Large Bit-Width vs. Small Bit-Width. A larger bit-width setting for each variable means a larger search box and a greater probability of finding an assignment. In the meantime, the backend SAT solver bears more burden because of the combinatorial explosion caused by redundant boolean variables. By contrast, a smaller bit-width setting for each variable means a smaller search box. Bit-blasting usually runs fast, but most likely, such a search box has no solutions, which will result in a time-consuming exhaustive search. For some very complex instances, if there is no solution in the search box, it is also very difficult to solve, even with a small bit-width setting. But hopefully, some constraints imply the search box of some variables. A proper bit-width setting to express such a search box will be of great help in improving efficiency.

Here we propose a list of strategies to analyze constraints and decide on an initial bit-width setting.

3.2 Multiplication Adaptation Bit-Width

Dealing with multiplication is a disadvantage for bit-blasting. A multiplication is transformed into a sequence of additions. Without loss of generality, assume that there are two bit-vectors: $\overline{x}, \overline{y}$, where $\{W(\overline{x}) = a, W(\overline{y}) = b\}, a < b, a, b \in \mathbb{N}_+$. we have the resulting bit vector $\overline{r}$ of multiplication of $\overline{x}$ and $\overline{y}$ [50],

$$\overline{r} = \overline{x} \times \overline{y} = ShiftAdd(\overline{x}, \overline{y}),$$

where the function *ShiftAdd* is a classical multiplication method consisting of two parts, shifting and adding.

$$ShiftAdd(\overline{x}, \overline{y}) = \sum_{0 \leq i < a} ShiftAdd_i(\overline{x}, \overline{y}).$$
$$ShiftAdd_i(\overline{x}, \overline{y}) = (\overline{y} \times \overline{x}_i) \ll i, 0 \leq i < a.$$

The shifting procedure will lead to bit-wise AND for single bit multiplication $(\overline{y} \times \overline{x}_i)$ and left shifting i bits $(\ll i)$. The number of newly introduced bit vectors with bit-width b is a. The adding procedure will add such bit vectors up and result in $\overline{r}$ with bit-width $a + b$. Multiplications will lead to a vast number of temporary boolean variables and clauses.

Example 3.1. Consider a simple constraint $x^2y^2z^3 \geq 0$. In practice, running with 2-bit will result in about 200 boolean variables and 700 clauses, while running with 16-bit will result in about 30000 boolean variables and 170000 clauses. Actually, although running with both settings can solve in a second, the backend SAT solver (the same setting of Section 6) runs 20 times slower in the latter setting than in the former.

We can determine the initial bit-width according to the number of multiplications. If a constraint contains few multiplications, it is effective to try a large bit-width. On the contrary, starting from a small bit-width will usually run fast. We consider such trade-off as a function to represent the relation between the initial bit-width and the number of multiplications,

$$B_{MA} = max(\beta - \lceil \alpha m \rceil, L),$$

where m is the number of multiplications, α is a positive real number factor, β is a positive integer factor, and L is a default bit-width. The function is monotonous and gives an initial bit-width of each integer variable. It is an adaptive process for alleviating the impact of multiplication on bit-blasting performance. When the number of multiplications decreases, the function gives a larger bit-width, otherwise a smaller bit-width, or the default bit-width L.

3.3 Vote for Proper Search Box

Here we list some bit-width decision strategies derived from analyzing specific kinds of constraints. To enhance stability and generalization from specific kinds of constraints, the bit-width setting vote for a bit-width from all variables, which finally helps depict a proper search box.

3.3.1 Distinct Graph. The distinction constraint $x \neq y$ asserts that x and y cannot be assigned the same value. Consider a special constraint ψ that is a conjunction of distinction constraints. We can construct a distinct graph $G(\psi)$ of ψ. $G(\psi)$ is a pair $\{V, E\}$, where $V = \{x_1, ..., x_n\}$ is the set of variables, and $E = \{(x_i, x_j) | x_i, x_j \in V\}$ is the set of distinct pairs where ψ has assertions $x_i \neq x_j$. If the distinct graph is complete, i.e., there is an edge between every two variables, it tells that any two variables have different values. The degree $deg(x)$ of a variable x is the number of arcs incident to it. We obtain a candidate bit-width to satisfy the constraint,

$$W_{DG}(x) = \lceil log_2(deg(x) + 1) \rceil.$$

Example 3.2. Consider a constraint that asserts all variables from x_1 to x_7 are all distinct,

$$(distinct\ x_1\ x_2\ x_3\ x_4\ x_5\ x_6\ x_7).$$

If we set $W(x_i) = 2$ where $1 \leq i \leq 7$ by default, it is easy to know that the SAT solver will report $\perp$, because it is impossible to assign the variables seven different values from $\{-2, -1, 0, 1\}$. Based on the heuristic, we obtain a complete distinct graph with 7 nodes where the degree of each node is 6. The bit-width setting $W(x_i) = \lceil log_2(7) \rceil = 3, 1 \leq i \leq 7$ will result in a feasible solution.

As for complex constraints, we can collect the distinctions from constraints and calculate W_{DG} as an initial candidate bit-width assignment for related variables. The assignment can help find an initial search box to satisfy such distinctions.

3.3.2 Coefficient Matching. A polynomial equation constraint is a strong constraint that can be used to eliminate variables. Consider

a special constraint $\psi = \sum_i c_i \cdot \boldsymbol{x}^{\boldsymbol{D}_i} = 0$, a single polynomial equation constraint, where $\boldsymbol{x}$ is a vector of integer variables, c_i is the coefficient of the i-th term, and $\boldsymbol{D}_i$ is an exponent vector of the i-th term. If the polynomial of constraint ψ has no constant term and the variables have no bounds, a trivial solution is $\{x = 0, x \in \mathcal{V}_{int}(\psi)\}$. Bit-blasting can find a feasible solution with a bit-width setting $\{W(x)|W(x) > 0\}$. On the contrary, if it has a constant term, the assignment $\{x = 0, x \in \mathcal{V}_{int}(\psi)\}$ cannot satisfy the constraint.

If we only consider a single variable in ψ, while assigning other variables 0 or 1, it can sometimes obtain a feasible solution easily. In the meantime, the coefficients (including constants, which is a coefficient of the 0-degree term) become critical. We can determine a bit-width setting to satisfy the constraint by matching the coefficients (coefficient of the term and constant term). For example, a constraint $3x + y + 6 = 0$ when $y = 0$ tells x a search box like $[-2, 1]$, i.e. bit-width $W(x) = 2$, so that it is able to find a solution $x = -2$. The bound of the search box is calculated via the division of the constant term and coefficient of the variable x. We obtain a candidate bit-width to satisfy the constraint,

$$W_{CM}(x) = \lceil log_2(\max_i(|c_i|)/|c_x|)\rceil + 1,$$

where c_x denotes the coefficient with the minimum absolute value of the term containing variable x.

Example 3.3. Consider a polynomial equation constraint,

$$xy + 11yz - 53 = 0.$$

If we set $\{W(x) = 2, W(y) = 2, W(z) = 2\}$ by default, it is easy to know that the SAT solver will report $\perp$ because the assignment with the largest values $x = 1, y = 1$ and $z = 1$ results a maximum value $-41 < 0$ that violates the constraint. Based on the heuristic, the bit-width setting $\{W(x) = 8, W(y) = 8, W(z) = 4\}$ will result in a feasible solution that could be $\{x = -1, y = -53, z = 0\}$.

3.3.3 Clip. Currently, bit-blasting based methods usually start from a small bit-width setting uniformly. The corresponding search box excludes the satisfiability of some nested constraints, which naturally simplifies the original constraint. The strategies (Distinct graph, Coefficient Matching) cannot realize such simplification.

Example 3.4. Consider a constraint $\psi = x^2 \geq 1 \vee y \geq 1000$ and a bit-width setting $W(x) = W(y) = 2$ where search box $I[x] = I[y] = [-2, 1]$. Any assignment of $y \in [-2, 1]$ will falsify $y \geq 1000$, but $x = 1 \in [-2, 1]$ satisfies $x^2 \geq 1$. The assignment $x = 1$ and arbitrary $y \in [-2, 1]$ also satisfies ψ.

There is a compromise between small and large bit-width settings. The small setting will simplify the original constraint but sometimes increase the number of restarts. The large setting will lead to a proper slack search box, but it sometimes stresses the backend SAT solver. Consequently, we generate final initial bit-width

$$W(x) = min(K, max\{W_{DG}(x), W_{CM}(x)\}),$$

where K is a hyper-parameter that clips bit-width that is too large.

Executing the above strategies sequentially will end up with initial bit-width assignments for each variable. If some variable misses all the strategies, We will set bit-width as the default value.

3.3.4 Vote. The bit-width setting W generated by the aforementioned heuristics can vote for a bit-width B_{VO} as follows.

$$B_{VO} = max(\{w|p(w) > \gamma\} \cup \{0\}),$$
$$p(w) = \frac{\#(W(x) = w)}{|\mathcal{V}_{int}|}, x \in \mathcal{V}_{int},$$

where $p(w)$ is the portion of the bit-width w in the total bit-width setting, $\gamma \in (0, 1)$ is a vote threshold factor, and $\#(W(x) = w)$ is the number of variables whose bit-widths are equal to w. If the set $\{w|p(w) > \gamma\}$ is empty, the candidate bit-width B_{VO} will be zero. If $B_{vote} \neq 0$, we will set all bit-widths of variables in $\mathcal{V}_{int}$ as the maximum value of B_{MA} and B_{VO}.

The Heuristics (Distinct Graph, Coefficient Matching, Clip) will concentrate on the specific constraints. Vote feeds on such heuristics and gives an overview of the setting where most variables are going to search. After Vote, we depict a proper search box.

First, heuristic strategies are generally only suitable for certain types of constraints. Sometimes counterexamples to heuristics will lead to an unaffordable bit-width for only a few variables, as the dilemma in the Motivating Example 3.1.1. Vote is able to control the effect to a certain extent. If the portion of variables choosing the same bit-width exceeds the threshold γ, running with B_{VO} will be a fast and affordable attempt.

In addition, it allows a wider search for the bit-width setting decided by the aforementioned heuristics (Distinct Graph, Coefficient Matching, and Clip). If there is not enough support for Vote, i.e. the percentage of any bit-width does not exceed the threshold γ, the bit-width setting decided by heuristics will be kept.

4 GREEDY ADDITION

In this section, we optimize bit-blasting in the case of successive addition. Addition is the core operation of bit vectors. Most arithmetic operations, like subtraction, and multiplication, are simulated by addition [26, 38, 50]. In the polynomial interpretation of SMT(QF_NIA) constraint, the case of successive addition is usual.

In the semantics of SMT(QF_NIA), addition satisfies the associative rule, but the bit-width calculation of addition does not. Without loss of generality, assume that there are three bit vectors: $\overline{x}, \overline{y}, \overline{z}$, where $\{W(\overline{x}) = a, W(\overline{y}) = b, W(\overline{z}) = c\}, 0 < a < b < c, a, b, c \in \mathbb{N}$. As shown in paper [50], we have the resulting bit vector $\overline{r}$ of addition of $\overline{x}$ and $\overline{y}$,

$$\overline{r}_i = \begin{cases} \overline{x'}_0 \oplus \overline{y}_0 \oplus \overline{t}_0, i = 0, \\ \overline{x'}_i \oplus \overline{y}_i \oplus \overline{t}_i, 1 \leq i \leq b, \end{cases}$$

where $\overline{x'}$ is the signed extension bit vector of $\overline{x}$ and $\overline{t}$ is an auxiliary bit vector to record the carry bits in the addition, i.e.,

$$\overline{x'} = sign\_ext(\overline{x}, b),$$
$$\overline{t}_{i+1} = \begin{cases} \perp, i = -1, \\ (\overline{x'}_i \wedge \overline{y}_i) \vee (\overline{x'}_i \wedge \overline{t}_i) \vee (\overline{y}_i \wedge \overline{t}_i), 0 \leq i < b. \end{cases}$$

The symbol $\oplus$ denotes XOR operations. The function *sign_ext* extends the bit-width of $\overline{x}$ from a to b by padding $b - a$ sign bits to the highest position. The bit-width of the resulting bit vector

$$W(\overline{r}) = W(\overline{x} + \overline{y}) = max(a, b) + 1 = b + 1,$$

where +1 at the end to store the overflow bit as in Example 2.1. Now consider the bit-width calculation of addition. For two different

orders of additions, the bit-width calculations are as follows:

$$\begin{aligned} W((\overline{x}+\overline{y})+\overline{z}) &= max(max(a,b)+1,c)+1 \\ &= max(b+1,c)+1 = c+1. \\ W(\overline{x}+(\overline{y}+\overline{z})) &= max(a,max(b,c)+1)+1 \\ &= max(a,c+1)+1 = c+2. \\ W((\overline{x}+\overline{y})+\overline{z}) &\neq W(\overline{x}+(\overline{y}+\overline{z})). \end{aligned}$$

So the bit-widths of the resulting bit vector of addition differ in successive addition under different orders. Note that the addition between bit vectors is a bivariate addition, and the multivariate addition will be converted into multiple bivariate additions. The trace of successive addition produces a binary tree T. Each node represents a bit vector. The parent bit vector is the result of adding two child bit vectors. Given a set of bit vectors X, $B(X,\mathcal{A})$ denotes the sum of bit-widths of all intermediate resulting bit vectors from bivariate additions of successive addition. $\mathcal{A}$ denotes the algorithm determining the variable order of bivariate additions on X. $B(X,\mathcal{A})$ is positively related to the number of boolean variables. An optimal algorithm $\mathcal{A}$ will result in the fewest boolean variables, which greatly eliminates redundancy in bit-blasting.

Algorithm 1 Greedy Addition (GA)

Input : X: a set of bit vectors.
Output: $\overline{z}$: the resulting bit vector.

1: **while** Size of X > 1 **do**
2: $\overline{s},\overline{t} \leftarrow$ the two bit-vectors with smallest bit-widths.
3: remove $\overline{s}$ and $\overline{t}$ from X.
4: $\overline{y} \leftarrow \overline{s}+\overline{t}$ and add $\overline{y}$ into X.
5: **end while**
6: $\overline{z} \leftarrow X[0]$.
7: **return** $\overline{z}$.

Theorem 4.1. *Given a set of bit vectors X, $B(X,GA)$ is minimal for a successive addition on X.*

Proof. Let bit-width be the weight of each node of the successive addition tree. $B(X,\mathcal{A})$ also denotes the sum of weights of internal nodes. Note that if the two nodes with the smallest weights are not the deepest leaves, swapping them to the deepest leaves will not increase $B(X,\mathcal{A})$.

The proof is by induction on n, the number of leaves, i.e. the size of the set X. For $n=2$, $B(X,\mathcal{A})$ are the same in arbitrary orders. Assume that for any tree created by Algorithm 1 that contains $n-1$ leaves, $B(X,GA)$ is minimal. Given a successive addition tree T built by Algorithm 1 with n leaves, $n \geq 2$, suppose that $w_1 \leq w_2 \leq \cdots \leq w_n$ where w_1 to w_n are the weights of leaves. We know that Algorithm 1 will first build the new nodes $\overline{s},\overline{t}$ with weight w_1 and w_2 as the deepest leaves. We replace the two nodes with a single node $\overline{y}=\overline{s}+\overline{t}$ with weight $W(\overline{y})=max(w_1,w_2)+1=w_2+1$, resulting in a new tree T' with $n-1$ nodes. By the induction hypothesis, $B(X\cup\{\overline{y}\}-\{\overline{s},\overline{t}\},GA)$ is minimal on T'. Returning the children to $\overline{y}$ restores tree T, which must also minimize $B(X,GA)$. □

Corollary 4.2. *Assume an addition of a set of n bit vectors with a bit-width sequence $[a_1,...,a_n]$ where $a_1 < a_2 < \cdots < a_n$. $B(X,\sigma)$ is minimal, where σ is the non-descending order of bit-width sequence.*

Example 4.3. Consider a constraint with successive addition,

$$y^2z+xy+yz+xz+x \geq 0.$$

where $\{W(\overline{x})=a, W(\overline{y})=b, W(\overline{z})=c\}, 0<a<b<c, a,b,c\in\mathbb{N}$. The left-hand side of the constraint will continuously add five terms: y^2z, xy, yz, xz, x. The corresponding bit-widths are $2b+c, a+b, b+c, a+c, a$, and obviously $a < a+b < a+c < b+c < 2b+c$. We can obtain $B(X,[y^2z,xy,yz,xz,x]) = 8b+4c+10$ while $B(X,[x,xy,xz,yz,y^2z]) = 2a+4b+3c+4$. It will save $2\times(8b+4c+10-(2a+4b+3c+4))$ boolean variables in the process of successive addition. The reason for multiplying by 2 is that there are also auxiliary variables for the carry bits. Actually, if we set $a=2, b=3, c=4$ as shown in Figure 2, running such a simple instance with GA will save 36 boolean variables and 270 clauses compared with running without GA.

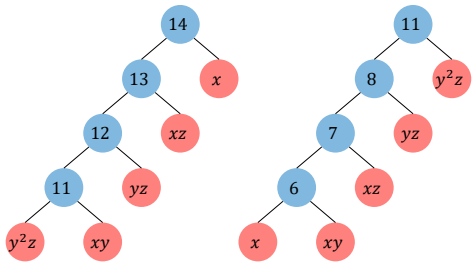


Figure 2: Results of successive addition with different orders. The internal nodes are labeled with their bit-widths. The left tree is under the order y^2z, xy, yz, xz, x, and the right tree is under the order x, xy, xz, yz, y^2z.

In practice, an SMT(QF_NIA) may contain thousands of additions and sometimes has a long continuous chain of additions. Greedy Addition will save a vast number of boolean variables, clauses as well as running time.

Theorem 4.4. *For a successive addition on X, Algorithm 1 will produce a resulting bit vector $\overline{z}$ with the smallest bit-width.*

If the successive addition is nested in multiplications, for example, $(y^2z+xy+yz+xz+x)\times(x+y+z)$, we can obtain the resulting bit vector with the smallest bit-width via Algorithm 1. From the aforementioned two theorems, Algorithm 1 is general to successive additions for optimizing bit-blasting.

5 BLAN ARCHITECTURE

In this section, we focus on the details of the architecture. First, we modify the *BB* procedure with some simplifications. Then we detail the architecture. We fine-tune the *BB* procedure as $BB(\psi', I)$, which accepts a search box I for each variable, and the Greedy Addition algorithm will also be integrated into it. Note that BLAN can solve unsatisfiable instances only if all variables are bounded in a finite domain. The situation can be distinguished by preprocessing

Algorithm 2 BLAN

Input : ψ: an SMT(QF_NIA) constraint; U: upper limit of bit-width.
Output: SAT/UNKNOWN

1: Obtain B_{MA} from Multiplication Adaptation.
2: **for** $x \in \mathcal{V}_{int}(\psi)$ **do**
3: $W(x) \leftarrow min(K, max\{W_{DG}(x), W_{CM}(x)\})$.
4: **end for**
5: Obtain B_{VO} from Vote.
6: **if** $B_{VO} \neq 0$ **then**
7: $W(x) \leftarrow max(B_{MA}, B_{VO}), x \in \mathcal{V}_{int}$.
8: **else**
9: $W(x) \leftarrow max(B_{MA}, W(x)), x \in \mathcal{V}_{int}$.
10: **end if**
11: **for** $x \in \mathcal{V}_{int}(\psi)$ **do**
12: $I[x] \leftarrow$ RANGE($W(x)$).
13: **end for**
14: **while** True **do**
15: **if** $BB(\psi, I) = \top$ **then**
16: **return** SAT.
17: **end if**
18: **for** $x \in \mathcal{V}_{int}(\psi)$ **do**
19: $I[x] \leftarrow$ RANGE($min(W(x) \times 2, U)$).
20: **end for**
21: **if** $\forall x \in \mathcal{V}_{int}(\psi).W(x) \geq U$ **then**
22: **return** UNKNOWN.
23: **end if**
24: **end while**
25: **return** UNKNOWN.

and then one can check satisfiability by invoking BB once. We will not go into details for this situation in this paper.

The architecture is shown in Algorithm 2. It is mainly divided into two parts. *(1) Obtaining Search Box* (lines 1-14). The process will obtain the search box for all variables and store them in $I[x]$. BLAN utilizes Multiplication Adaptation to obtain B_{MA} in line 1. In lines 2-4, BLAN obtains a candidate bit-width from Distinct Graph, Coefficient Matching, and Clip. BLAN utilizes Vote to obtain B_{VO} in line 5. If the Vote succeeds in lines 6-7, the bit-width will be set to the maximum value between the B_{MA} and B_{VO}. Otherwise, in lines 8-10, the bit-width will be set to the maximum value between the B_{MA} and $W(x)$ obtained from the strategies (Distinct Graph, Coefficient Matching, and Clip). In lines 11-13, BLAN finally generates the initial search box of all variables. *(2) Check* (lines 14-25). The process will finally obtain the result, SAT or UNKNOWN. If the procedure $BB(\psi, I)$ results $\top$ in lines 15-17, BLAN returns SAT. Otherwise, resulting in $\bot$ from line 15, BLAN doubly increments the search box for the variables in lines 18-20. It returns UNKNOWN if exceeding the limit in lines 21-23.

6 EMPIRICAL EVALUATION

We perform an empirical evaluation of BLAN, comparing its performance with other state-of-the-art SMT solvers. Specifically, we aim to answer the following research questions.

Table 1: The categories of benchmarks.

Category	# sat	# unknown	# unsat	Total
AProVE [30]	1663	123	623	2409
Automizer [32]	0	0	7	7
Calypto	80	0	97	177
Dartagnan [27]	7	356	11	374
Ezsmt [28]	0	8	0	8
LassoRanker [39]	10	1	127	138
LCTES	0	0	2	2
Leipzig	162	0	5	167
MCM [40]	29	156	1	186
CInteger	861	546	411	1818
ITS	9503	3975	3568	17046
SAT14	1853	1	72	1926
MathPorblems	822	46	232	1100

- RQ1: Does BLAN perform better to solve constraints compared to other state-of-the-art SMT solvers?
- RQ2: How effective are the strategies of BLAN?
- RQ3: Is BLAN sensitive to the choice of parameters?

6.1 Experimental Setup

6.1.1 Implementation. We implemented the solver named BLAN in C++ compiled by g++ with '-O3' option. It employs LIBPOlY [35], which is a library for manipulating polynomials and CADICAL [7] with its default configurations as the backend SAT solver. The source code is available at https://github.com/MRVAPOR/BLAN.

6.1.2 Parameters. The symbol U denotes the maximum bit-width, default $U = 32$. If all bit-widths are larger than or equal to U, we stop the procedure as exceeding the limit of bit-width. The default starting bit-width $L = 2$ once missed all strategies stated in this paper. We select the best set of parameters of strategies on a random 20% of the dataset (2998 instances), and details will be given in Section 6.4. The parameters of Section 3.2, $\alpha = 1/1536, \beta = 7$ result in $W(x) = max(7 - \lceil m/1536 \rceil, 2)$. The Clip Constant is $K = 16$, and the Vote threshold factor is $\gamma = 0.5$ in Section 3.3.

6.1.3 Dataset. The dataset has 11 categories totaling 25358 instances, with 14990 labeled sat, i.e., the instance is satisfiable, 5212 labeled unknown, i.e., the status of the instance is still not confirmed by SMT-LIB community, and 5156 labeled unsat, i.e., the instance is unsatisfiable. State-of-the-art solvers have difficulty in determining the status of most of unknown instances. All instances are available from SMT-LIB: http://smtlib.cs.uiowa.edu/.

Most benchmarks are industrial (95% of total instances). They focus on different kinds of tasks like state reachability [27], termination verification [39], and so on. This dataset is public, authoritative, and rich in industrial instances. We believe these instances are sufficient to explain the variety and difficulty of SMT(QF_NIA) problems and are fair to evaluate the performance of SMT solvers.

6.1.4 Subjects. Other SMT solvers participating in experiments are the latest version of state-of-the-art SMT solvers that support SMT(QF_NIA) and are top solvers in SMT-COMP [5]: APROVE [29], CVC5 1.0.2 [3], MATHSAT 5.6.8 [15], YICES 2.6.2 [23] and Z3 4.11.2 [18]. APROVE utilizes the pure bit-blasting method. Z3 [18] combines bit-blasting method and NLSAT algorithm [34]. MATHSAT

Table 2: Results of #S and #U of each solver in each category of satisfiable SMT-COMP benchmarks.

Solvers	AProVE	calypto	Dartagnan	LassoRanker	Leipzig	MCM	CInteger	ITS	SAT14	MathProblems	Total	#U
APROVE	**1663**	77	0	9	**161**	0	667	6291	0	647	9515	8
CVC5	1354	79	**7**	**10**	94	13	320	5448	1788	230	9343	0
MATHSAT	1639	79	**7**	**10**	128	13	707	7553	1770	193	12099	16
YICES2	1591	79	6	**10**	101	10	511	6783	1837	112	11040	9
Z3	1658	**80**	**7**	**10**	159	15	760	8397	**1852**	659	13597	15
Z3(B)	1630	59	0	**10**	**161**	0	678	4878	244	658	8318	0
BLAN(ours)	1662	**80**	**7**	**10**	**161**	**29**	**837**	**9243**	1845	**688**	**14562**	**422**

[15] utilizes incremental linearization method [14]. CVC5 [13] combines all known methods: bit-blasting method, NLSAT algorithm, and incremental linearization. YICES2 [33] is a non-hybrid solver that utilizes a pure NLSAT algorithm and adapts it for non-linear integers constraints . Due to the tactic language of Z3 [20], we implement an SMT solver Z3(B) which only invokes the preprocessing procedure and bit-blasting method of Z3. According to our research, only APROVE uses the pure bit-blasting method detailed in paper [50], and it is not open source.

6.1.5 Metric. There are five important indicators:

- T: the running time (seconds) of solving an instance. In the following experiments, the average running time AVG.T will usually be used.
- #S: the number of solved instances.
- #U: the number of (unique) instances that can only be solved by the current solver but cannot be solved by other solvers.
- #R: the number of restarts of solving an instance and the average version #AVG.R.
- #V: the number of boolean variables after bit-blasting.
- #C: the number of clauses after bit-blasting.

We will compare BLAN with other SMT solvers in the running time of solving an instance T and the number of solved instances #S. The number of restarts #R is defined by the number of invoking bit-blasting except for the first time. If the solver finds a solution in the initial search box, it never restarts, i.e. #R = 0. It indicates the quality of the search box, where the smaller #R, the better the quality of the search box. However, this indicator alone is incomplete because if we choose a huge search box, #R will be very small. The indicators T and #S are also necessary. A strategy is effective if it increases #S or reduces T while reducing #R. Besides, Greedy Addition is used to save redundancy so that the number of boolean variables #V and clauses #C are other necessary indicators.

6.1.6 Environment. All experiments are done with an Intel Xeon Platinum 8153 CPU (2.00GHz) and 1024G RAM within CentOS 7.7.1908. We take 20 minutes as time-bound.

6.2 RQ1: Comparison to Other SMT Solvers

In order to demonstrate the solving ability and speed of BLAN, we perform and analyze empirical evaluation on the satisfiable, unknown, respectively. We also show an outlook of the active role in solving unsatisfiable instances.

6.2.1 On Satisfiable Instances. Table 2 reports the number of solved instances of each solver in each category. We boldface the cases with the most solved instances of each category. BLAN achieves the best in most categories (Calypto, MCM, CInteger, Dartagnan, LassoRanker, ITS, MathProblems, Leipzig, and total solved instances). It also solves the most unique instances, which cannot be solved by other solvers. The SMT solvers APROVE and Z3(B) with the pure bit-blasting method are less satisfactory. BLAN is much better on Dartagan, MCM, SAT14, and CInteger, while having a good performance on AProVE, Leipzig, and MathProblems, where APROVE and Z3(B) solve well. The results show that BLAN outperforms state-of-the-art SMT solvers in solving ability.

Figure 3 shows the number of solved instances over time. Our solver solves the most number of instances on all time slices. On all solved instances by all solvers (total 3946), BLAN achieves 4.66x, 11.11x, 3.11x, 2.02x, 2.11x, and 2.22x faster than APROVE, CVC5, MATHSAT, YICES2, Z3, and Z3(B), respectively. The results show that BLAN outperforms state-of-the-art SMT solvers in speed.

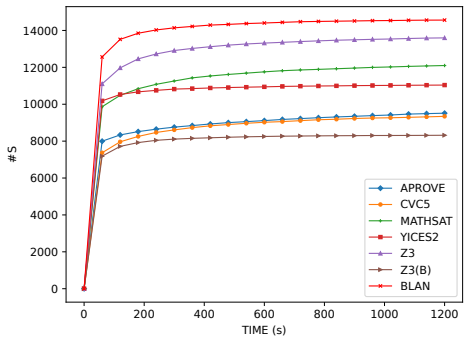


Figure 3: Results of #S of each solvers over time.

6.2.2 On Unknown Instances. There are 5212 unknown instances in SMT-LIB, which are highly challenging for state-of-the-art SMT solvers. Their satisfiability has not been officially confirmed.

When solving unknown problems, the results of sat are easy to check. We can directly generate the feasible assignment of solved satisfiable instances and bring in the SMT(QF_NIA) constraints to check whether it is consistent. The results of unsat are difficult to verify because we need a verifier to verify the proof generated by SMT solvers, which is still a subject of urgent research [19, 36, 47] and beyond the scope of this paper.

Here we list the new solved satisfiable and checked instances among all SMT solvers in Table 3. We boldface the cases with the most solved instances of each category. The instances of AProVE, LassoRanker, SAT14, and MathProblems are all unsolved.

BLAN finds 114 new and 80 unique satisfiable instances out of 5212 unknown instances whose feasible assignments have been checked by Z3. BLAN solved about five times more instances of #S and #U than other SMT solvers.

Table 3: Results of the number of (uniquely) solved and checked instances labeled unknown.

Solvers	Dartagnan	Ezsmt	MCM	CInteger	ITS	Total	#U
APROVE	0	0	0	0	29	29	17
CVC5	0	0	0	0	0	0	0
MATHSAT	**11**	**8**	1	1	0	21	3
YICES2	2	**8**	0	**2**	7	19	1
Z3	8	**8**	0	1	9	26	3
Z3(B)	0	**8**	0	0	5	13	0
BLAN(ours)	2	**8**	**27**	**2**	**75**	**114**	**80**

6.2.3 On Unsat instances. Essentially, bit-blasting is a reasoning and searching method over finite domains. BLAN can solve unsatisfiable instances only if all variables are bounded in a finite domain. Yuri Matiyasevich's result [41] also implied that a universal algorithm does not exist. But hopefully in practice, we can combine it with other methods capable of partially solving unsatisfiable instances. We already know the strong ability of BLAN to solve satisfiable instances. In this section, we demonstrate an outlook of BLAN on unsatisfiable instances via combining BLAN with other solvers, denoted as Y+BLAN. Y is the solver like CVC5, MATHSAT, YICES2, Z3, as APROVE and Z3(B) cannot handle unsatisfiable instances.

We can combine BLAN with other solvers as follows. Assume that the original SMT formula is ψ, BLAN searches in a finite space $F = \bigotimes_{x \in V} I[x]$, where V is the set of variables and $I[x] = l_x \leq x \leq u_x$. If BLAN has not found a model in F, it generates a lemma ϕ to rule out the space where

$$\phi = \bigvee_{x \in V} \neg I(x) = \bigvee_{x \in V} x < l_x \vee x > u_x.$$

We run the solver Y for 0.8 of time at first, then BLAN for 0.1 of time, and Y solves the formula $\psi \wedge \phi$ for the rest time. Y will solve most unsatisfiable instances at first. Then for the rest difficult instances, BLAN will generate a lemma to rule out the space that most likely has a solution (according to the strategies). Y will restart to reason on the original constraints combining with the lemma.

Table 4 shows the performance on unsatisfiable instances of solvers. Solvers have improved after combining with BLAN. MATHSAT+BLAN achieved the best results and YICES2+BLAN achieved the largest increase. The combination method of BLAN with other solvers, the division of the running time and etc. need to be further explored. Nevertheless, It shows the possibility of positive impact on solving unsatisfiable instances.

Table 4: Results of #S of each solver in each category of unsatisfiable SMT-COMP benchmarks.

Solvers	CVC5	MATHSAT	YICES2	Z3
Y	4222	4666	4459	4570
Y + BLAN	4238	**4685**	4657	4587

6.3 RQ2: Effectiveness of the Strategies

To better investigate the roles of strategies, we begin with a pure bit-blasting solver without strategies proposed in this paper, named BLAN(P). It starts from default bit-width 2 and restarts with $W' = \{W(x) \times 2 | x \in \mathcal{V}_{int}\}$ once failed. We compare the effects of BLAN(P) and BLAN and then illustrate the effectiveness by adding the strategy one by one to BLAN(P). MA, VO, and GA denote the strategies of Multiplication Adaptation, Vote, and Greedy Addition, respectively. Note that different from MA and VO, GA is an algorithm component of the *BB* procedure. We will conduct an experiment to show the validity at first. We perform the empirical evaluation on satisfiable instances of SMT-LIB.

Table 5: Results of the average #V and #C on all solved instances of some categories. The numbers in the table are in scientific notation multiplied by 10^4.

Solvers	($\times 10^4$)	AProVE	LassoRanker	CInteger	ITS	SAT14
BLAN(P)	#V	3.43	0.38	3.37	9.52	16.64
	#C	14.65	1.77	17.50	49.60	107.91
BLAN(P)+GA	#V	3.41	0.37	3.17	8.02	9.24
	#C	14.53	1.73	16.10	38.62	52.50

GA will save redundant boolean variables and clauses. Table 5 lists the average number of variables and clauses of BLAN(P) + GA compared with BLAN(P) on all solved instances of some categories. Other not mentioned categories are not changed. The benchmark of SAT14 has plenty of successive addition operations, so GA shows significant advantages, almost twice reducing the number of variables and clauses. Using an example to illustrate the effect, the instance SAT14/96.smt2 has chains of addition at most 107 terms, and bit-blasting (2-bit) without GA will utilize 126623 boolean variables and 849707 clauses while with GA utilizing 39970 boolean variables and 201054 clauses, almost a 4x improvement.

BLAN(P) + GA solved more instances and faster than BLAN(P) because reducing the redundancy of boolean variables and clauses accelerates the backend SAT solver. There is surplus time to call the bit-blasting procedure several times if failed in the previous bit-width setting. It is also reflected in 0.01 more restarts on average in Table 6. The results show that Greedy Addition is effective in saving redundant boolean variables and clauses and improving the efficiency of bit-blasting based method.

In Table 6, we notice that BLAN(P) + GA + MA and BLAN(P) + GA + VO reduce #AVG.R and solve more instances and run faster, which indicates that such strategies are effective. Especially, BLAN(P) + GA + MA solves 644 more instances in ITS. BLAN(P) + GA + VO solves 337 more instances in SAT14. Both of them are helpful in deciding a proper search box in specific kinds of instances.

The solver BLAN is also BLAN(P) + GA + MA + VO. As shown in Table 6, it inherits and further enhances the power of GA, MA, and VO. BLAN decreases the #AVG.R 0.87, which almost means that BLAN invokes *BB* procedure one less time than BLAN(P) on all instances. At the same time, BLAN solves 1089 more instances and reduces 107.7 seconds on average. It indicates that BLAN can find a better search box and such combination of the proposed methods is effective.

Table 6: Results of improvement of #AVG.R, #S and AVG.T of solvers with different settings. The symbol Δ(#AVG.R), Δ(#S) and Δ(AVG.T) denote the difference between solvers and BLAN(P) on #AVG.R, #S and AVG.T, respectively.

Solvers	Δ(#AVG.R) ↓	Δ(#S) ↑	Δ(AVG.T) ↓
BLAN(P) + GA	0.01	182	-39.43
BLAN(P) + GA + MA	-0.78	645	-63.01
BLAN(P) + GA + VO	-0.27	437	-65.23
BLAN	-0.87	1089	-107.7

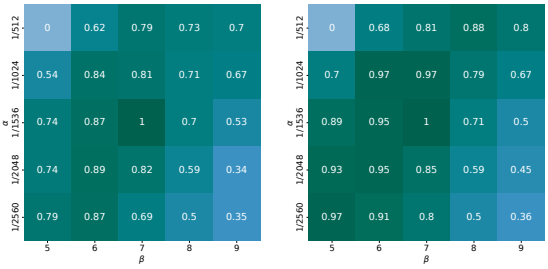
6.4 RQ3: Sensitivity Analysis

The satisfiable dataset was randomly split into two sets of 20% and 80%: one is *training* set for parameter selection (20%, totaling 2998 instances), and the other is *testing* set to verify its generalization (80%, totaling 11992 instances). The best parameters on *training* set have shown the best overall performance as in Section 6.2 and Section 6.3. In this section, we study the effect of some parameters on the performance of BLAN, especially the generalization of the parameters of BLAN. The symbols α, β, K, and γ are parameters of Multiplication Adaptation Bit-Width, Clip, and Vote, respectively. DG and CM denote the strategies of Distinct Graph and Coefficient Matching, respectively.

6.4.1 Impact of α and β. We fixed the other parameters $K = 16, \gamma = 0.5$ and test different (α, β) settings. Figure 4 show the performance on *training* and *testing* sets. The parameters vary as $\alpha \in \{\frac{1}{512}, \frac{1}{1024}, \frac{1}{1536}, \frac{1}{2048}, \frac{1}{2560}\}$ and $\beta \in \{5, 6, 7, 8, 9\}$. The pair $(\frac{1}{1536}, 7)$ shows the best performance both on *training* and *testing* sets. The results (#S) of experiments with different (α, β) are normalized via $(\#S - \#S_{min})/(\#S_{max} - \#S_{min})$. $\#S_{max}$ and $\#S_{min}$ are the maximum and minimum number of solved instances on *training* and *testing* sets. It is able to demonstrate solving capabilities for different settings. From the observation of the heat map, if it is required to set a pair of (α, β), $(\frac{1}{1536}, 7)$ seems to be the appropriate choice according to the conducted experiment.

6.4.2 Impact of K and γ. We fixed the other parameters $\alpha = \frac{1}{1536}, \beta = 7$. First, we fixed $\gamma = 0.5$ and run solver with different Clip Constants $K \in \{4, 8, 16, 24\}$. The results are listed in Table 7 where the row of 'AVG.T on All Solved' means the average time on all solved instances by all solvers. The last column represents an experiment without the clip constant. Next, We fixed $K = 16$ and run solver with different Vote thresholds $\gamma \in \{0.1, 0.3, 0.5, 0.7, 0.9\}$. The results are listed in Table 8. We find that the selection of K and γ leads to extreme points both on *training* and *testing* sets. Therefore, if it is required to set the Clip Constant K and Vote threshold γ, 16 and 0.5 seems to be the appropriate choice.

6.4.3 Impact of DG, CM. DG and CM are the preconditions of Clip and Vote. The strategies Clip and Vote do not work if no strategy gives a candidate bit-width setting for integer variables. We fixed $K = 16$ and $\gamma = 0.5$. Only Dartagnan and MathProblems have plenty of distinction constraints, and DG reduces #AVG.R on such categories from 2.14 to 1.57 and 1.98 to 1.95, respectively. We generate 999 instances with only a distinction constraint i.e. $(distinct\ x_1\ x_2\ \cdots\ x_n)$, where n ranges from 2 to 1000. BLAN with DG will solve 990 instances, while BLAN without DG will only solve 14 instances. BLAN can handle extreme situations that contain a vast number of distinction constraints. CM does not improve any indicators compared with the solver without CM. But after Clip and Vote, the solver reduces #AVG.R from 2.25 to 0.18 and solved 337 more instances in SAT14. BLAN with CM, Clip, and Vote invokes *BB* procedure twice less than BLAN without them.



(a) On *training* set (b) On *testing* set

Figure 4: The heat map of solved instances with different (α, β) setting. The value of each element is calculated by $(\#S - \#S_{min})/(\#S_{max} - \#S_{min})$, where $\#S_{max}$ and $\#S_{min}$ are the maximum and minimum number of solved instances.

Table 7: #S and AVG.T on all solved instances with different clip constant K on *training* and *testing* sets.

		$K = 4$	$K = 8$	$K = 16$	$K = 24$	without K
training	#S	2833	2837	**2914**	2894	2900
	AVG.T on All Solved	44.97	39.89	**31.86**	40.35	40.86
testing	#S	11311	11435	**11648**	11606	11613
	AVG.T on All Solved	41.04	40.64	**28.77**	37.36	38.65

Table 8: #S and AVG.T on all solved instances with different Vote thresholds γ on *training* and *testing* sets.

		$\gamma = 0.1$	$\gamma = 0.3$	$\gamma = 0.5$	$\gamma = 0.7$	$\gamma = 0.9$
training	#S	2908	2898	**2914**	2815	2812
	AVG.T on All Solved	38.72	36.95	**33.93**	47.35	46.54
testing	#S	11622	11619	**11648**	11256	11202
	AVG.T on All Solved	34.18	33.96	**29.17**	41.00	41.69

6.5 Threats to Validity

Correctness of implementation. As far as we know, except for APROVE (not open source), no other state-of-the-art solvers has implemented flexible-size bit-blasting. We implement not only the strategies but also the flexible-sized bit-blasting method. To ensure correctness, at least three developers reviewed the source code. We also performed extensive testing on the scrambled benchmarks via official scrambler [48]. We compared results of BLAN to those of other solvers to validate the correctness.

Generalization of parameters. We have discussed the generalization of parameters in Subsection 6.4. Because the division of dataset is random, we also make another 5 random divisions. We

found that the best parameters selected on the random *training* datasets are all the same. BLAN with such parameters performs best on the *training*, *testing* and the overall set. BLAN can obtain accurate parameters via training on a small number of instances, which also gives a positive inspiration.

7 RELATED WORK

SMT(QF_BV) has been thoroughly researched. It is important in hardware verification [46], validity checking of programming language [16], and bounded model checking [6]. Most modern SMT solvers like BOOLECTOR [11], BITWUZLA [42] and so on does a lot of work on the preprocessing [12, 31]. Besides, PARTI [22] utilizes interval information to reduce search space. Trident [49] utilizes interval-guided variable assignments and dependency-guided variable ordering to facilitate efficiency on SMT(QF_BV). This line of work attempts to reduce search space via word-level information and shows that it is of great help to improve efficiency.

For SMT(QF_NIA), there has been a list of methods. ISAT [25] introduces intervals as decision elements into the famous architecture of SAT solving, Davis–Putnam–Logemann–Loveland (DPLL). It can handle boolean combinations of constraints on real numbers and integers and even transcendental functions. Barcelogic [8], and MATHSAT [14] utilize linearization that linearizes the nonlinear operations as fresh variables [8], or undefined functions [9]; next, the solvers attempt to solve the constraints in linear arithmetic theory (combining undefined function theory). Methods such as NLSAT [34] are based on cylindrical algebraic decomposition (CAD) [1], a complete projection-based algorithm in nonlinear real arithmetic, and can check inconsistency on SMT(QF_NIA). Z3 [18] combines bit-blasting and NLSAT algorithm to manage satisfiable and unsatisfiable instances. Based on NLSAT, YICES2 implements a pure MCSAT framework [33], reinforcing the resolve procedure to generate a stronger lemma than NLSAT and improving the performance on unsatisfiable SMT(QF_NIA) instances significantly. CVC5 [3] utilizes a hybrid strategy that combines all the aforementioned methods via running on different time slices, while SMTRAT [37] combines them via a general branch-and-bound architecture. Currently, most solvers use mixed strategies, but it still makes sense to explore the limits of what a single strategy can achieve.

In the last two decades, the bit-blasting method was often used for SMT(QF_NIA). APROVE gave the bit-blasting transformation formulae over natural integers [26] and application to matrix arithmetic [24]. Later MINISMT [50] extends it to integers, which completes transformation formulae in SMT(QF_NIA). Improving the efficiency of bit-blasting based method from the word or bit level is still the focus of research.

8 CONCLUSION AND FUTURE WORK

In this paper, we improve the bit-blasting based method to solve the SMT(QF_NIA) problem. We proposed the Multiplication Adaptation and Vote strategies combining the heuristics (Distinct Graph, Coefficient Matching, and Clip) to help to depict a proper search box. The Greedy Addition algorithm also reduces redundancy in successive additions. We implemented an efficient solver for solving satisfiable instances, BLAN. The experiments show the advantages of BLAN both in solving ability and speed compared with other state-of-the-art SMT solvers. Notably, BLAN finds many new results of unknown instances.

In the future, we hope to migrate the ideas to other bit-blasting based methods, for example, SMT(QF_BV). We also hope to find a more adaptive way to tune parameters. Besides, we will investigate how to solve unsatisfiable constraints more efficiently.

ACKNOWLEDGMENTS

This work has been supported by the National Natural Science Foundation of China (NSFC) under grants No.61972384 and No.62132020. Feifei Ma is also supported by the Youth Innovation Promotion Association CAS under grant No. Y202034. The authors would like to thank the anonymous reviewers for their comments and suggestions.

REFERENCES

[1] Dennis S. Arnon, George E. Collins, and Scott McCallum. 1984. Cylindrical Algebraic Decomposition I: The Basic Algorithm. *SIAM J. Comput.* 13, 4 (1984), 865–877. https://doi.org/10.1137/0213054

[2] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *ACM Comput. Surv.* 51, 3 (2018), 50:1–50:39. https://doi.org/10.1145/3182657

[3] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. 2022. cvc5: A Versatile and Industrial-Strength SMT Solver. In *TACAS 2022*, Dana Fisman and Grigore Rosu (Eds.), Vol. 13243. 415–442.

[4] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2017. *The SMT-LIB Standard: Version 2.6.* Technical Report. Department of Computer Science, The University of Iowa. Available at `www.SMT-LIB.org`.

[5] Clark W. Barrett, Leonardo Mendonça de Moura, and Aaron Stump. 2005. SMT-COMP: Satisfiability Modulo Theories Competition. In *CAV 2005 (Lecture Notes in Computer Science, Vol. 3576)*, Kousha Etessami and Sriram K. Rajamani (Eds.). Springer, 20–23. https://doi.org/10.1007/11513988_4

[6] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. 1999. Symbolic Model Checking without BDDs. In *TACAS 1999 (Lecture Notes in Computer Science, Vol. 1579)*, Rance Cleaveland (Ed.). Springer, 193–207. https://doi.org/10.1007/3-540-49059-0_14

[7] Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximillian Heisinger. 2020. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling Entering the SAT Competition 2020. In *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, Tomas Balyo, Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda (Eds.), Vol. B-2020-1. 51–53.

[8] Cristina Borralleras, Daniel Larraz, Enric Rodríguez-Carbonell, Albert Oliveras, and Albert Rubio. 2019. Incomplete SMT Techniques for Solving Non-Linear Formulas over the Integers. *ACM Trans. Comput. Log.* 20, 4 (2019), 25:1–25:36. https://doi.org/10.1145/3340923

[9] Cristina Borralleras, Salvador Lucas, Rafael Navarro-Marset, Enric Rodríguez-Carbonell, and Albert Rubio. 2009. Solving Non-linear Polynomial Arithmetic via SAT Modulo Linear Arithmetic. In *CADE 2009 (Lecture Notes in Computer Science, Vol. 5663)*, Renate A. Schmidt (Ed.). Springer, 294–305. https://doi.org/10.1007/978-3-642-02959-2_23

[10] Martin Brain. 2021. Further Steps Down The Wrong Path: Improving the Bit-Blasting of Multiplication. In *CAV 2021*, Alexander Nadel and Aina Niemetz (Eds.), Vol. 2908. 23–31.

[11] Robert Brummayer and Armin Biere. 2009. Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. In *TACAS 2009*, Stefan Kowalewski and Anna Philippou (Eds.), Vol. 5505. 174–177.

[12] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, Ziyad Hanna, Alexander Nadel, Amit Palti, and Roberto Sebastiani. 2007. A Lazy and Layered SMT($\mathcal{BV}$) Solver for Hard Industrial Verification Problems. In *CAV 2007 (Lecture Notes in Computer Science, Vol. 4590)*, Werner Damm and Holger Hermanns (Eds.). Springer, 547–560. https://doi.org/10.1007/978-3-540-73368-3_54

[13] Alessandro Cimatti, Alberto Griggio, Ahmed Irfan, Marco Roveri, and Roberto Sebastiani. 2017. Invariant Checking of NRA Transition Systems via Incremental Reduction to LRA with EUF. In *TACAS 2017*, Axel Legay and Tiziana Margaria (Eds.), Vol. 10205. 58–75.

[14] Alessandro Cimatti, Alberto Griggio, Ahmed Irfan, Marco Roveri, and Roberto Sebastiani. 2018. Experimenting on Solving Nonlinear Integer Arithmetic with

Incremental Linearization. In *SAT 2018*, Olaf Beyersdorff and Christoph M. Wintersteiger (Eds.), Vol. 10929. 383–398.
[15] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. 2013. The MathSAT5 SMT Solver. In *TACAS 2013*, Nir Piterman and Scott A. Smolka (Eds.), Vol. 7795. 93–107.
[16] Byron Cook, Daniel Kroening, and Natasha Sharygina. 2005. Cogent: Accurate Theorem Proving for Program Verification. In *CAV 2005 (Lecture Notes in Computer Science, Vol. 3576)*, Kousha Etessami and Sriram K. Rajamani (Eds.). Springer, 296–300. https://doi.org/10.1007/11513988_30
[17] Hernán Ponce de León, Florian Furbach, Keijo Heljanko, and Roland Meyer. 2020. Dartagnan: Bounded Model Checking for Weak Memory Models (Competition Contribution). In *TACAS 2020 (Lecture Notes in Computer Science, Vol. 12079)*, Armin Biere and David Parker (Eds.). Springer, 378–382. https://doi.org/10.1007/978-3-030-45237-7_24
[18] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS 2008*, C. R. Ramakrishnan and Jakob Rehof (Eds.), Vol. 4963. 337–340.
[19] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. 2008. Proofs and Refutations, and Z3. In *LPAR 2008 Workshops (CEUR Workshop Proceedings, Vol. 418)*, Piotr Rudnicki, Geoff Sutcliffe, Boris Konev, Renate A. Schmidt, and Stephan Schulz (Eds.). CEUR-WS.org. http://ceur-ws.org/Vol-418/paper10.pdf
[20] Leonardo Mendonça de Moura and Grant Olney Passmore. 2013. The Strategy Challenge in SMT Solving. In *Automated Reasoning and Mathematics - Essays in Memory of William W. McCune*, Maria Paola Bonacina and Mark E. Stickel (Eds.), Vol. 7788. 15–44.
[21] Peter Dinges and Gul A. Agha. 2014. Targeted test input generation using symbolic-concrete backward execution. In *ASE 2014*, Ivica Crnkovic, Marsha Chechik, and Paul Grünbacher (Eds.). ACM, 31–36. https://doi.org/10.1145/2642937.2642951
[22] Oscar Soria Dustmann, Klaus Wehrle, and Cristian Cadar. 2018. PARTI: a multi-interval theory solver for symbolic execution. In *ASE 2018*, Marianne Huchard, Christian Kästner, and Gordon Fraser (Eds.). 430–440.
[23] Bruno Dutertre. 2014. Yices 2.2. In *CAV 2014*, Armin Biere and Roderick Bloem (Eds.), Vol. 8559. 737–744.
[24] Jörg Endrullis, Johannes Waldmann, and Hans Zantema. 2008. Matrix Interpretations for Proving Termination of Term Rewriting. *J. Autom. Reason.* 40, 2-3 (2008), 195–220.
[25] Martin Fränzle, Christian Herde, Tino Teige, Stefan Ratschan, and Tobias Schubert. 2007. Efficient Solving of Large Non-linear Arithmetic Constraint Systems with Complex Boolean Structure. *J. Satisf. Boolean Model. Comput.* 1, 3-4 (2007), 209–236. https://doi.org/10.3233/sat190012
[26] Carsten Fuhs, Jürgen Giesl, Aart Middeldorp, Peter Schneider-Kamp, René Thiemann, and Harald Zankl. 2007. SAT Solving for Termination Analysis with Polynomial Interpretations. In *SAT 2007*, João Marques-Silva and Karem A. Sakallah (Eds.), Vol. 4501. 340–354.
[27] Natalia Gavrilenko, Hernán Ponce de León, Florian Furbach, Keijo Heljanko, and Roland Meyer. 2019. BMC for Weak Memory Models: Relation Analysis for Compact SMT Encodings. In *CAV 2019 (Lecture Notes in Computer Science, Vol. 11561)*, Isil Dillig and Serdar Tasiran (Eds.). Springer, 355–365. https://doi.org/10.1007/978-3-030-25540-4_19
[28] Martin Gebser, Max Ostrowski, and Torsten Schaub. 2009. Constraint Answer Set Solving. In *ICLP 2009 (Lecture Notes in Computer Science, Vol. 5649)*, Patricia M. Hill and David Scott Warren (Eds.). Springer, 235–249. https://doi.org/10.1007/978-3-642-02846-5_22
[29] Jürgen Giesl, Cornelius Aschermann, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Jera Hensel, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, Thomas Ströder, Stephanie Swiderski, and René Thiemann. 2017. Analyzing Program Termination and Complexity Automatically with AProVE. *J. Autom. Reason.* 58, 1 (2017), 3–31.
[30] Jürgen Giesl, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, Thomas Ströder, Stephanie Swiderski, and René Thiemann. 2014. Proving Termination of Programs Automatically with AProVE. In *IJCAR 2014*, Stéphane Demri, Deepak Kapur, and Christoph Weidenbach (Eds.), Vol. 8562. 184–191.
[31] Liana Hadarean, Kshitij Bansal, Dejan Jovanovic, Clark W. Barrett, and Cesare Tinelli. 2014. A Tale of Two Solvers: Eager and Lazy Approaches to Bit-Vectors. In *VSL 2014 (Lecture Notes in Computer Science, Vol. 8559)*, Armin Biere and Roderick Bloem (Eds.). Springer, 680–695. https://doi.org/10.1007/978-3-319-08867-9_45
[32] Matthias Heizmann, Daniel Dietsch, Jan Leike, Betim Musa, and Andreas Podelski. 2015. Ultimate Automizer with Array Interpolation - (Competition Contribution). In *TACAS 2015 (Lecture Notes in Computer Science, Vol. 9035)*. Springer, 455–457. https://doi.org/10.1007/978-3-662-46681-0_43
[33] Dejan Jovanovic. 2017. Solving Nonlinear Integer Arithmetic with MCSAT. In *VMCAI 2017*, Ahmed Bouajjani and David Monniaux (Eds.), Vol. 10145. 330–346.
[34] Dejan Jovanovic and Leonardo Mendonça de Moura. 2012. Solving Non-linear Arithmetic. In *IJCAR 2012*, Bernhard Gramlich, Dale Miller, and Uli Sattler (Eds.), Vol. 7364. 339–354.
[35] Dejan Jovanovic and Bruno Dutertre. 2017. LibPoly: A Library for Reasoning about Polynomials. In *CAV 2017*, Martin Brain and Liana Hadarean (Eds.), Vol. 1889. 28–39.
[36] Daniela Kaufmann, Armin Biere, and Manuel Kauers. 2020. From DRUP to PAC and Back. In *DATE 2020*. IEEE, 654–657. https://doi.org/10.23919/DATE48585.2020.9116276
[37] Gereon Kremer, Florian Corzilius, and Erika Ábrahám. 2016. A Generalised Branch-and-Bound Approach and Its Application in SAT Modulo Nonlinear Integer Arithmetic. In *CASC 2016*, Vladimir P. Gerdt, Wolfram Koepf, Werner M. Seiler, and Evgenii V. Vorozhtsov (Eds.), Vol. 9890. 315–335.
[38] Daniel Kroening and Ofer Strichman. 2016. *Bit Vectors*. Springer Berlin Heidelberg, Berlin, Heidelberg, 135–156. https://doi.org/10.1007/978-3-662-50497-0_6
[39] Jan Leike and Matthias Heizmann. 2018. Geometric Nontermination Arguments. In *TACAS 2018 (Lecture Notes in Computer Science, Vol. 10806)*, Dirk Beyer and Marieke Huisman (Eds.). Springer, 266–283. https://doi.org/10.1007/978-3-319-89963-3_16
[40] Nuno P. Lopes, Levent Aksoy, Vasco M. Manquinho, and José Monteiro. 2010. Optimally Solving the MCM Problem Using Pseudo-Boolean Satisfiability. *CoRR* abs/1011.2685 (2010).
[41] Y. V. Matiyasevich. 1993. *Hilbert's tenth problem*. MIT press.
[42] Aina Niemetz and Mathias Preiner. 2020. Bitwuzla at the SMT-COMP 2020. *CoRR* abs/2006.01621 (2020).
[43] Corina S. Pasareanu, Neha Rungta, and Willem Visser. 2011. Symbolic execution with mixed concrete-symbolic solving. In *ISSTA 2011*, Matthew B. Dwyer and Frank Tip (Eds.). ACM, 34–44. https://doi.org/10.1145/2001420.2001425
[44] Hao Ren, Devesh Bhatt, and Jan Hvozdovic. 2016. Improving an Industrial Test Generation Tool Using SMT Solver. In *NFM 2016 (Lecture Notes in Computer Science, Vol. 9690)*, Sanjai Rayadurgam and Oksana Tkachuk (Eds.). Springer, 100–106. https://doi.org/10.1007/978-3-319-40648-0_8
[45] Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, Percy Liang, and Aditya V. Nori. 2013. A Data Driven Approach for Algebraic Loop Invariants. In *ESOP 2013 (Lecture Notes in Computer Science, Vol. 7792)*. Springer, 574–592. https://doi.org/10.1007/978-3-642-37036-6_31
[46] Eli Singerman. 2005. Challenges in making decision procedures applicable to industry. *Proc. PDPAR'05* 144, 2 (2005).
[47] Aaron Stump, Duckki Oe, Andrew Reynolds, Liana Hadarean, and Cesare Tinelli. 2013. SMT proof checking using a logical framework. *Formal Methods Syst. Des.* 42, 1 (2013), 91–118. https://doi.org/10.1007/s10703-012-0163-3
[48] Tjark Weber. 2016. Scrambling and Descrambling SMT-LIB Benchmarks. In *SMT@IJCAR 2016 (CEUR Workshop Proceedings, Vol. 1617)*. CEUR-WS.org, 31–40. http://ceur-ws.org/Vol-1617/paper3.pdf
[49] Peisen Yao, Qingkai Shi, Heqing Huang, and Charles Zhang. 2020. Fast bit-vector satisfiability. In *ISSTA 2020*, Sarfraz Khurshid and Corina S. Pasareanu (Eds.). 38–50.
[50] Harald Zankl and Aart Middeldorp. 2010. Satisfiability of Non-linear (Ir)rational Arithmetic. In *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16*, Edmund M. Clarke and Andrei Voronkov (Eds.), Vol. 6355. 481–500.

Received 2023-02-16; accepted 2023-05-03