Polynomial Constraint Solving over Finite Domains with the Modified Bernstein Form

Federico Bergenti, Stefania Monica, and Gianfranco Rossi

Dipartimento di Matematica e Informatica
Università degli Studi di Parma
Parco Area delle Scienze 53/A, 43124 Parma, Italy
{federico.bergenti,stefania.monica,gianfranco.rossi}@unipr.it

Abstract. This paper describes an algorithm that can be used to effectively solve polynomial constraints over finite domains. Such constraints are expressed in terms of inequalities of polynomials with integer coefficients whose variables are assumed to be defined over proper finite domains. The proposed algorithm first reduces each constraint to a canonical form, i.e., a specific form of inequality, then it uses the modified Bernstein form of resulting polynomials to incrementally restrict the domains of variables. No approximation is involved in the solving process because the coefficients of the modified Bernstein form of the considered type of polynomials are always integer numbers.

1 Introduction and Motivation

In this paper we study constraints written in terms of polynomials whose coefficients are integer numbers and whose variables are defined over proper finite subsets of integer numbers. Such constraints, that we call *polynomial constraints over finite domains*, are ubiquitous in many areas of system analysis and verification (as discussed, e.g., in [1]) and they deserve specific treatment because common finite-domain techniques cannot adequately process them, even in simple cases. The language of CLP(FD) is suitable to express polynomial constraints over finite domains, but the fact that FD solvers are incomplete sometimes does not allow to treat such constraints adequately. For example, if constraints are expressed as non-linear polynomials, the FD solver that ships with SWI-Prolog (version 6.6.6) may fail in completely reducing the domains of variables [6]. Consider the constraint

```
X in -10..10, X^2 #>= 9
```

which involves only one variable over a finite interval, namely $I = [-10..10]$. It is easy to observe that such a constraint is satisfied only for values in the interval $I_R = [-10..-3] \cup [3..10]$. However, the FD solver exhibits its well-known incompleteness and it rewrites the constraint as

```
X in -10..-1\/1..10, Y in 9..100, X^2 #= Y
```

which introduces an unneeded variable and counts $\{-2, -1, 1, 2\}$ as admissible values. Concerning non-linear constraints that involve more than one variable, consider

```
X in -10..10, Y in -10..10, X*Y #>= 21
```

where both variables are over the finite domain I. Again, such a constraint is satisfied only for values in I_R for both variables, but the FD solver introduces an unneeded variable and it also allows inadmissible values in the domains of variables. As a matter of fact, the FD solver rewrites the constraint as

```
X in -10..-1\/1..10, Y in -10..-1\/1..10, Z in 21..100, X*Y #= Z.
```

In this paper we describe an algorithm to solve polynomial constraints over finite domains that overcomes the well-known incompleteness of FD solvers. The proposed algorithm ensures that the domains of variables after the solution process contain only admissible values by a proper use of the specific form of allowed constraints, namely polynomial inequalities. Polynomials with real coefficients whose variables admit values in known intervals can always be rewritten in so called *Bernstein Form* (*BF*) (see, e.g., [4] or [5]) to effectively compute a lower bound and an upper bound of their range. Such bounds can be used to effectively verify polynomial inequalities, and they have been used extensively for optimization problems. Unfortunately, even if a polynomial has only integer coefficients, no guarantees can be provided that its equivalent in BF has integer coefficients. Therefore, the BF has been traditionally used only with polynomials with real or rational variables. In this paper we show how to use a variant of the BF, that we call *Modified Bernstein Form* (*MBF*), to process polynomial constraints over finite domains with no involvement of rational or real numbers. Note that the MBF is also known in the literature as *scaled* BF [2], but we adopted the *modified* adjective since the introduced multiplicative factor is not the same for all coefficients of the BF, and the resulting patch is not geometrically scaled. As a matter of fact, if a polynomial has integer coefficients, its equivalent in MBF always has integer coefficients. This property allows effective reasoning on polynomial inequalities with integer coefficients. In details, even if the MBF does not provide accurate bounds for the range of polynomials, it provides information on the sign of suitable bounds, which is sufficient to reason on polynomial constraints over finite domains. The algorithm that we propose uses only such signs, and the actual values of bounds are not used. The current implementation of the proposed algorithm was used to validate all examples presented in this paper and it is available for download (`cmt.dmi.unipr.it/software/clppolyfd.zip`).

The paper is organized as follows. Section 2 presents the proposed approach and the underlying constraint language. Section 3 introduces the MBF and it proves notable properties that allow using the coefficients of the MBF to compute the signs of suitable lower and upper bounds of polynomials, as needed by the proposed algorithm. Section 4 illustrates in details the behaviour of the proposed algorithm by means of specific examples. Finally, Section 5 concludes the paper and presents planned developments.

2 The Proposed Approach

The interesting characteristics of the constraint solving approach that we propose derive from the specific form of constraints that we address. Polynomial constraints over finite domains are expressed in terms of inequalities of polynomials with integer coefficients whose variables are defined over finite subsets of integer numbers. The relative constraint language is described as follows.

2.1 The Constraint Language

The syntax of the language of polynomial constraints that we consider is defined, as usual, by its signature Σ, a triple $\Sigma =< \mathcal{V}, \mathcal{F}, \Pi >$ where $\mathcal{V}$ is a denumerable set of variable symbols, $\mathcal{F}$ is the set of constant and function symbols, and Π is the finite set of constraint predicate symbols allowed in the language. The set $\mathcal{F}$ of constant and function symbols is $\mathcal{F} = O \cup Z$, where $O = \{+, *\}$ is the set of function symbols representing binary operations over integer numbers, and $Z = \{0, 1, -1, 2, -2, \ldots\}$ is the denumerable set of constants representing integer numbers. The set Π of constraint predicate symbols is $\Pi = \{=, \neq, <, \leq, >, \geq\}$, and it contains only binary predicate symbols.

A *primitive constraint* is any atomic predicate built using the symbols from signature Σ. A (*non-primitive*) *constraint* is a conjunction of primitive constraints. The semantics of the language defined over signature Σ is trivial and it allows expressing systems of polynomial equalities, inequalities and disequalities with integer coefficients.

Example 1 *The following are primitive constraints expressed using signature Σ with $\mathcal{V} = \{x, y, z\}$:*

$$x * x \geq 9$$
$$3 * x * x + 9 \leq 5 * y * y + z$$
$$x \neq y + z.$$

Note that other common function symbols and predicate symbols are supported in terms of syntactic abbreviations, as follows

$$v \text{ in } a..b \rightarrow (v + (-1) * a) * (v + (-1) * b) \leq 0 \quad (1)$$
$$v \text{ nin } a..b \rightarrow (v + (-1) * a) * (v + (-1) * b) > 0 \quad (2)$$
$$-t \rightarrow (-1) * t \quad (3)$$
$$t_1 - t_2 \rightarrow t_1 + (-1) * t_2 \quad (4)$$
$$t^0 \rightarrow 1 \quad (5)$$
$$t^n \rightarrow \underbrace{t * t * \cdots * t}_{n>0} \quad (6)$$

where v is a variable symbol, a and b are integer constant symbols such that $a \leq b$, and t, t_1, and t_2 are terms.

Note that the extended language introduced by the aforementioned syntactic abbreviations does not increase the set of expressible constraints, and that we allow operator precedence and parenthesized expressions in this language.

The following example is meant to clarify why v in $a..b$ and v nin $a..b$ (read *v not in $a..b$*) are rewritten using the mentioned syntactic abbreviations.

Example 2 *Syntactic rules rewrite the constraint* x in $1..3$ *as*

$$(x + (-1) * 1) * (x + (-1) * 3) \leq 0.$$

As a matter of fact, $p(x) = (x-1)(3-x)$ *represents a downward parabola whose vertex is* $x = 2$ *(which corresponds to* $p(2) = 1$*) and whose intersections with the* x*-axis are* $x = 1$ *and* $x = 3$ *(so that* $p(1) = p(3) = 0$*). For all values of* x *not in* $[1..3]$ *the values of the parabola* $p(x)$ *are negative and it can then be concluded that the rewritten constraint is equivalent to* x *in* $[1..3]$*.*

Analogously, syntactic rules rewrite the constraint x nin $1..3$ *as*

$$(x + (-1) * 1) * (x + (-1) * 3) > 0.$$

As a matter of fact, $p(x) = (x - 1)(x - 3)$ *represents an upward parabola whose vertex is* $x = 2$ *(which corresponds to* $p(2) = -1$*) and whose intersections with the* x*-axis are* $x = 1$ *and* $x = 3$ *(so that* $p(1) = p(3) = 0$*). For all the values of* x *not in* $[1..3]$*, instead, the values of the parabola* $p(x)$ *are positive, so that it can be concluded that the rewritten constraint is equivalent to* x *not in* $[1..3]$*.*

2.2 Constraints in Canonical Form

If a total order on variable symbols is fixed, any primitive constraint can be rewritten to a canonical form. The chosen canonical form expresses any primitive constraint as $t \geq 0$, where t is a term in canonical form. We say that a term t is in canonical form if it is a summation term (i.e., $t = t_1 + \ldots + t_h$) involving only terms with the following structure

$$t_i = k * \underbrace{x_1 * x_1 * \cdots * x_1}_{n_1} * \underbrace{x_2 * x_2 * \cdots * x_2}_{n_2} * \cdots * \underbrace{x_m * x_m * \cdots * x_m}_{n_m} \qquad i \in [1..h]$$

where k is a constant symbol and the $\{x_i\}_{i=1}^{m}$ are distinct variable symbols arranged using the fixed total order. The fixed total order induces a lexicographic order of product terms that is used to arrange the $\{t_i\}_{i=1}^{h}$ in t. We can define a function $deg(v, t)$ on product terms by returning n_v if variable symbol v is repeated n_v times in t. Function $deg(v, t)$ allows writing product terms in a compact form as

$$\begin{aligned} t_i &= k * \underbrace{x_1 * x_1 * \cdots * x_1}_{deg(x_1,t_i)} * \underbrace{x_2 * x_2 * \cdots * x_2}_{deg(x_2,t_i)} * \cdots * \underbrace{x_m * x_m * \cdots * x_m}_{deg(x_m,t_i)} \\ &= k * x_1\hat{}deg(x_1, t_i) * x_2\hat{}deg(x_2, t_i) * \cdots * x_m\hat{}deg(x_m, t_i) \qquad i \in [1..h] \end{aligned}$$

Besides ordinary algebraic manipulations and the total order of variable symbols, the conversion to the canonical form uses the following ideas. First, observe that any constraint of the form $f(\mathbf{x}) \odot g(\mathbf{x})$, where $f(\mathbf{x})$ and $g(\mathbf{x})$ are polynomials with integer coefficients, $\mathbf{x}$ is the vector of variables of polynomials, and $\odot \in \Pi$, can be expressed in the form $p(\mathbf{x}) \odot 0$, where $p(\mathbf{x}) = f(\mathbf{x}) - g(\mathbf{x})$ is still a polynomial with integer coefficients. Second, in order to convert constraints in canonical form we use the following lemma, which holds for every polynomial $p(\mathbf{x})$ with integer coefficients.

Lemma 1 *The following co-implications hold for every polynomial $p(\mathbf{x})$ with integer coefficients and variables $\mathbf{x}$*

$$p(\mathbf{x}) \leq 0 \iff -p(\mathbf{x}) \geq 0 \tag{7}$$

$$p(\mathbf{x}) > 0 \iff p(\mathbf{x}) \geq 1 \iff p(\mathbf{x}) - 1 \geq 0 \tag{8}$$

$$p(\mathbf{x}) < 0 \iff p(\mathbf{x}) \leq -1 \iff -p(\mathbf{x}) - 1 \geq 0 \tag{9}$$

$$p(\mathbf{x}) = 0 \iff p^2(\mathbf{x}) \leq 0 \iff -p^2(\mathbf{x}) \geq 0 \tag{10}$$

$$p(\mathbf{x}) \neq 0 \iff p^2(\mathbf{x}) > 0 \iff p^2(\mathbf{x}) - 1 \geq 0 \tag{11}$$

The following illustrative examples are provided to exemplify how generic constraints can be written in canonical form.

Example 3 *Consider the following primitive constraints in the form $(x_1+x_2)\odot 0$ where $\odot \in \Pi$.*

1. *The constraint $x_1 + x_2 \leq 0$ can be written as $-x_1 - x_2 \geq 0$ by co-implication (7) of lemma 1, which is in canonical form.*
2. *The constraint $x_1 + x_2 > 0$ can be written as $x_1 + x_2 - 1 \geq 0$ by co-implication (8) of lemma 1, which is in canonical form.*
3. *The constraint $x_1 + x_2 < 0$ can be written as $x_1 + x_2 \leq -1$ by co-implication (9) of lemma 1, which corresponds to $-x_1 - x_2 - 1 \geq 0$ in canonical form.*
4. *The constraint $x_1 + x_2 = 0$ can be written as $(x_1 + x_2)^2 \leq 0$ by co-implication (10) of lemma 1, which is $-x_1^2 - x_2^2 - 2x_1x_2 \geq 0$ in canonical form.*
5. *The constraint $x_1 + x_2 \neq 0$ can be written as $(x_1 + x_2)^2 > 0$ by co-implication (11) of lemma 1, and it can be written as $x_1^2 + 2x_1x_2 + x_2^2 - 1 \geq 0$ in canonical form.*

2.3 The Constraint Solving Algorithm

The canonical form of primitive constraints has the interesting property of reducing constraint solving to the study of the sign of polynomials with integer coefficients whose variables are defined over finite subsets of integer numbers. In the rest of this paper we assume that constraints are always written in canonical form since the reformulation of constraints to their respective canonical forms can be considered a simple preprocessing step.

As discussed in Section 1, we assume that the user provides initial bounds for all variables, i.e., we can assume that before the solving process begins

$$x_i \in [\underline{x}_i..\overline{x}_i] \qquad \underline{x}_i \in \mathbb{Z} \qquad \overline{x}_i \in \mathbb{Z} \qquad i \in [1..m] \tag{12}$$

where $\{x_i\}_{i=1}^m$ are all the variables involved in the constraints, $\{\underline{x}_i\}_{i=1}^m$ are their lower bounds, and $\{\overline{x}_i\}_{i=1}^m$ are their upper bounds. The lower bounds and the upper bounds identify a box in $\mathbb{Z}^m$ that can be used as an initial approximation of the domain of variables. The constraint solving algorithm refines the initial box and possibly breaks it into disjoint boxes until boxes are sufficiently refined and the consistency of constraints can be certified in each of them. In details, the algorithm considers all primitive constraints and it verifies if each constraint is consistent in the current box. If no consistency can be certified, a variable is selected and the current box is split into two disjoint boxes by splitting the current domain of the selected variable. The obtained disjoint boxes are processed recursively. The algorithms always terminates because the consistency of constraints can always be verified in a box formed by intervals that contain only one element. The performances of the algorithm depend significantly on the details of the subdivision, i.e., on the techniques adopted to select a variable and to split a box into two disjoint boxes, and a large number of alternatives have been studied in the literature (see, e.g., [3]). In the rest of this paper, we assume that variables are selected using a fixed lexicographic order and that once a variable x_i is selected, its domain $[\underline{x}_i..\overline{x}_i]$ is split at

$$s_i = \left\lfloor \frac{(\underline{x}_i + \overline{x}_i)}{2} \right\rfloor \tag{13}$$

into two disjoint intervals $[\underline{x}_i..s_i]$ and $[(s_i + 1)..\overline{x}_i]$.

For each primitive constraint $p(\mathbf{x}) \geq 0$, the algorithm computes suitable lower bound l and upper bound u, with $l \leq u$, for the polynomial $p(\mathbf{x})$ in the current box, and it uses such values to verify if $p(\mathbf{x}) \geq 0$ in the current box. The following procedure summarizes the core of the constraint solving algorithm:

1. If $u < 0$ the constraint is not consistent in the current box;
2. If $l \geq 0$ the constraint is consistent in the current box;
3. Otherwise, the current box should be split and analysed recursively.

The following example shows how the proposed algorithm works for a simple constraint under the assumption that suitable lower bounds and upper bounds can be computed.

Example 4 *Let us consider the constraint $x - 3 \geq 0$ involving only variable x, which is assumed to be defined in $I = [0..5]$. Using the notation introduced in the algorithm described above, suitable values of the lower and upper bounds are $l = -3$ and $u = 2$, respectively, so that none of the first two conclusive cases of the algorithm applies. The interval I is then split into two subintervals, namely $I_L = [0..2]$ and $I_R = [3..5]$. First, consider I_L with suitable values of the lower and upper bounds as $l = -3$ and $u = -1$, respectively, so that case* (1) *holds, i.e., the constraint is not consistent in the current box I_L. Then, consider I_R, with suitable lower and upper bounds as $l = 0$ and $u = 2$, respectively, so that case* (2) *holds, i.e., the constraint is consistent in I_R. The domain of the variable x can then be set equal to I_R, and Figure 1 shows how the initial approximation of the domain is processed.*

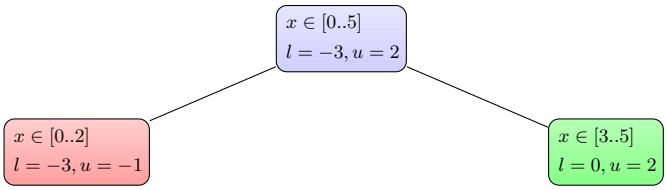

Fig. 1. Description of the refinement of the domain of variable x in Example 4.

It is worth noting that we are only interested in the signs of l and u, and not in their actual value. Any algorithm that allows the effective computation of the signs of suitable l and u can be used to support the proposed algorithm. In the following section we present a technique to effectively compute such signs using the modified Bernstein form of polynomials.

3 The Modified Bernstein Form of Polynomials

In this section we introduce the *Modified Bernstein Form* (*MBF*) of polynomials and we explain how to use it to solve polynomial constraints over finite domains. As an illustrative special case, we start by considering univariate polynomials. Problems involving only polynomials in one variable are not relevant for typical applications, however, they allow simplifying the notation and, therefore, they allow the proposed approach to be explained in a simpler way.

3.1 Univariate Polynomial Constraints

We consider a single constraint expressed in canonical form as

$$p(x) \geq 0 \tag{14}$$

where

$$p(x) = \sum_{i=0}^{n} a_i x^i \tag{15}$$

is a generic polynomial of degree n, x is the (unique) variable, $\{a_i\}_{i=0}^{n}$ are the coefficients relative to the canonical basis $\mathcal{B}$ given by the following set of monomials

$$\mathcal{B} = \{1, x, x^2, \ldots, x^n\} \tag{16}$$

and we assume that x is defined in a given interval $I = [\underline{x}..\overline{x}]$. In order to write the explicit expression of the polynomial $p(x)$ in the Bernstein basis, we first need to consider a new variable t defined in $[0, 1]$, which is related to x according to the following affine transformation

$$x = \underline{x} + (\overline{x} - \underline{x})t. \tag{17}$$

By substituting (17) in (15), one obtains

$$\sum_{i=0}^{n} a_i(\underline{x} + (\overline{x} - \underline{x})t)^i = \sum_{i=0}^{n}\sum_{k=0}^{i} a_i \binom{i}{k} \underline{x}^{i-k}(\overline{x} - \underline{x})^k t^k = \sum_{i=0}^{n} c_i t^i \tag{18}$$

where the coefficients $\{c_i\}_{i=0}^n$ are defined as

$$c_i = \sum_{k=i}^{n} \binom{k}{i} a_k \underline{x}^{k-i}(\overline{x} - \underline{x})^i \qquad i \in [0..n]. \tag{19}$$

Observe that $\{c_i\}_{i=0}^n$ are related to the coefficients $\{a_i\}_{i=0}^n$ relative to the canonical basis (16) and also to the extremes $\underline{x}$ and $\overline{x}$ of the interval I where $p(x)$ is defined. Moreover, since the coefficients $\{a_i\}_{i=0}^n$ are integer numbers, it can be concluded that also $\{c_i\}_{i=0}^n$ are integer numbers. The coefficients $\{c_i\}_{i=0}^n$ defined in (19) are necessary to derive the MBF of $p(x)$.

The Bernstein basis is given by the set of $n+1$ polynomials $\{B_i^n(x)\}_{i=0}^n$, the i-th of which is given by

$$B_i^n(x) = \binom{n}{i} \frac{(x - \underline{x})^i(\overline{x} - x)^{n-i}}{(\overline{x} - \underline{x})^n} \qquad i \in [0..n]. \tag{20}$$

Using this Bernstein basis, the polynomial defined in (15) can be written as

$$p(x) = \sum_{i=0}^{n} b_i B_i^n(x) \tag{21}$$

where the coefficients $\{b_i\}_{i=0}^n$ are related to $\{c_i\}_{i=0}^n$, and, hence, to $\{a_i\}_{i=0}^n$, according to

$$b_i = \sum_{j=0}^{i} \frac{\binom{i}{j}}{\binom{n}{j}} c_j \qquad i \in [0..n]. \tag{22}$$

The coefficients $\{b_i\}_{i=0}^n$ are useful to find a lower bound and an upper bound of the polynomial $p(x)$. As a matter of fact the following inequalities hold

$$\min_{i \in [0..n]} \{b_i\} \leq p(x) \leq \max_{i \in [0..n]} \{b_i\} \qquad x \in I. \tag{23}$$

Observe that, even if the coefficients $\{c_i\}_{i=0}^n$ are integer numbers, the coefficients $\{b_i\}_{i=0}^n$ are not guaranteed to be integer because of the divisions by binomial coefficients. For this reason, the treatment of the $\{b_i\}_{i=0}^n$ requires representing rational numbers, which often involves approximations. However, it is possible to consider a modified Bernstein basis whose i-th element is given by

$$\tilde{B}_i^n(x) = \frac{(x - \underline{x})^i(\overline{x} - \underline{x})^{n-i}}{(\overline{x} - \underline{x})^n} \qquad i \in [0..n]. \tag{24}$$

Each element $\tilde{B}_i^n(x)$ of the modified Bernstein basis is obtained from the corresponding element $B_i^n(x)$ of the Bernstein basis using the following equality

$$B_i^n(x) = \binom{n}{i} \tilde{B}_i^n(x). \tag{25}$$

From (21), the polynomial $p(x)$ can be written in terms of the modified Bernstein basis as

$$p(x) = \sum_{i=0}^{n} \tilde{b}_i \tilde{B}_i^n(x) \tag{26}$$

where the coefficients $\{\tilde{b}_i\}_{i=0}^n$ are

$$\tilde{b}_i = \binom{n}{i} b_i \qquad i \in [0..n]. \tag{27}$$

The right-hand side of (26) represents the MBF of polynomial $p(x)$. The advantage of considering the MBF of $p(x)$ instead of its Bernstein form is that the coefficients $\{\tilde{b}_i\}_{i=0}^n$ are integer numbers and they can therefore be represented with no approximation, provided that integer numbers can be represented with arbitrary precision. As a matter of fact, since

$$\binom{n}{i} \frac{\binom{i}{j}}{\binom{n}{j}} = \binom{n-j}{i-j} \tag{28}$$

the coefficients $\{\tilde{b}_i\}_{i=0}^n$ can be written as

$$\tilde{b}_i = \sum_{j=0}^{i} \binom{n-j}{i-j} c_j \qquad i \in [0..n] \tag{29}$$

from which it can be easily deduced that they are integer numbers. It is worth noting that binomial coefficients are integer numbers and they can be computed without divisions. As a matter of fact, the following recursive formula holds for binomial coefficients, when $n \geq j > 0$

$$\binom{n}{j} = \binom{n-1}{j} + \binom{n-1}{j-1}. \tag{30}$$

The recursion terminates because the binomial coefficient equals 1 for $j = 0$.

Observe that (23) holds for the coefficients $\{b_i\}_{i=0}^n$ and not for $\{\tilde{b}_i\}_{i=0}^n$. However, as previously discussed, we are only interested in the signs of the coefficients $\{b_i\}_{i=0}^n$, which are the same as that of corresponding $\{\tilde{b}_i\}_{i=0}^n$ since they only differ by a positive multiplicative factor. For this reason, it is possible to focus only on the integer coefficients $\{\tilde{b}_i\}_{i=0}^n$.

3.2 Multivariate Polynomial Constraints

A more sophisticated notation needs to be introduced in the multivariate case. We denote as $\{x_j\}_{j=1}^k$ the considered variables where $k \geq 1$ and j is an index that we associate with each variable. We define the vector $\mathbf{x}$ of all variables

$$\mathbf{x} = (x_1, \ldots, x_k) \tag{31}$$

and we define a *multi-index* I as $I = (i_1, \ldots, i_k)$, so that $\mathbf{x}^I = x_1^{i_1} \cdot \ldots \cdot x_k^{i_k}$.

A k-variate polynomial $p(\mathbf{x})$ with real coefficients can be written as

$$p(\mathbf{x}) = \sum_{I \leq N} a_I \mathbf{x}^I \tag{32}$$

where $N = (n_1, \ldots, n_k)$ is a multi-index, and operations on multi-indices are defined component-wise. Observe that the inequality $I \leq N$ among multi-indices is also component-wise, namely it corresponds to

$$i_j \leq n_j \qquad j \in [1..k]. \tag{33}$$

As in the univariate case, we start by considering a single constraint, namely

$$p(\mathbf{x}) \geq 0 \tag{34}$$

and we assume that each variable is defined over a finite interval. More precisely, we denote as $\underline{\mathbf{x}} = (\underline{x}_1, \ldots, \underline{x}_k)$ and $\overline{\mathbf{x}} = (\overline{x}_1, \ldots, \overline{x}_k)$ the extremes of the box $\mathbf{I} = [\underline{\mathbf{x}}, \overline{\mathbf{x}}]$ where the polynomial $p(\mathbf{x})$ is defined. In order to find the explicit expression in the Bernstein basis of the polynomial $p(\mathbf{x})$ defined in (32), it is necessary to consider an affine change of variable between $\mathbf{x}$ and a new variable $\mathbf{t}$ defined over the unit box $[0, 1]^k$. Using the affine transformation

$$x_i = \underline{x}_i + (\overline{x}_i - \underline{x}_i) t_i \tag{35}$$

it is possible to show that

$$p(\mathbf{x}) = \sum_{I \leq N} a_I \mathbf{x}^I = \sum_{I \leq N} c_I \mathbf{t}^I \tag{36}$$

where the coefficients c_I are related to a_I according to

$$c_I = \sum_{L=I}^{N} a_I \binom{L}{I} \underline{\mathbf{x}}^{L-I} (\overline{\mathbf{x}} - \underline{\mathbf{x}})^I \tag{37}$$

and

$$\binom{L}{I} = \binom{l_1}{i_1} \cdot \ldots \cdot \binom{l_k}{i_k}. \tag{38}$$

Then, it is possible to write the polynomial $p(\mathbf{x})$ defined in (32) as

$$p(\mathbf{x}) = \sum_{I \leq N} b_I B_I^N(\mathbf{x}) \tag{39}$$

where the polynomials $\{B_I^N(\mathbf{x})\}_{I\leq N}$ represent the Bernstein basis of the considered polynomial space with the following explicit expression

$$B_I^N(\mathbf{x}) = B_{i_1}^{n_1}(x_1)\cdot B_{i_2}^{n_2}(x_2)\cdot \ldots \cdot B_{i_k}^{n_k}(x_k) \tag{40}$$

and

$$B_{i_j}^{n_j}(x_j) = \binom{n_j}{i_j}\frac{(x_j-\underline{x}_j)^{i_j}(\overline{x}_j-x_j)^{n_j-i_j}}{(\overline{x}_j-\underline{x}_j)^{n_j}} \qquad i_j\in\{0,\ldots,n_j\} \quad j\in\{1,\ldots,k\}.$$

The coefficients b_I are related to c_I (and, hence, to a_I) according to

$$b_I = \sum_{J\leq I}\frac{\binom{I}{J}}{\binom{N}{J}}c_J \qquad I\leq N. \tag{41}$$

Notably, the range of the polynomial $p(\mathbf{x})$ satisfies

$$\text{range}(p)\subseteq[\min_{I\leq N}\{b_I\},\max_{I\leq N}\{b_I\}]. \tag{42}$$

As in the univariate case, such a well-known property of the Bernstein form is useful to study polynomial constraints because it provides effective lower and upper bounds of a polynomial, and it can be used to study its sign in a box.

We now aim at deriving the MBF of the polynomial $p(\mathbf{x})$ starting from its Bernstein form in (39). In order to do so, let us consider the modified Bernstein basis given by the following set of polynomials

$$\tilde{B}_I^N(\mathbf{x}) = \tilde{B}_{i_1}^{n_1}(x_1)\cdot \tilde{B}_{i_2}^{n_2}(x_2)\cdot \ldots \cdot \tilde{B}_{i_k}^{n_k}(x_k) \tag{43}$$

where

$$\tilde{B}_{i_j}^{n_j}(x_j) = \frac{(x_j-\underline{x}_j)^{i_j}(\overline{x}_j-x_j)^{n_j-i_j}}{(\overline{x}_j-\underline{x}_j)^{n_j}} \qquad i_j\in\{0,\ldots,n_j\} \quad j\in\{1,\ldots,k\}.$$

Using such a basis, the polynomial $p(\mathbf{x})$ can be written as

$$p(\mathbf{x}) = \sum_{i=0}^{n}\tilde{b}_I\tilde{B}_I^N(\mathbf{x}) \tag{44}$$

where the coefficients $\{\tilde{b}_I\}_{I\leq N}$ are

$$\tilde{b}_I = \sum_{J\leq I}\binom{N}{I}\frac{\binom{I}{J}}{\binom{N}{J}}c_J = \sum_{J\leq I}\binom{N-J}{I-J}c_J \qquad I\leq N. \tag{45}$$

Observe that, since all factors involved in (45) are integer numbers, the coefficients $\{\tilde{b}_I\}_{I\leq N}$ of the polynomial $p(\mathbf{x})$ in the MBF are integer numbers. Moreover, the binomial coefficients in (45) can be computed with no divisions using (30). This allows evaluating the coefficients $\{\tilde{b}_I\}_{I\leq N}$ with no approximation. As in the univariate case, even if condition (42) holds for only the coefficients $\{b_I\}_{I\leq N}$, we can focus on the coefficients $\{\tilde{b}_I\}_{I\leq N}$ since we are only interested in their signs.

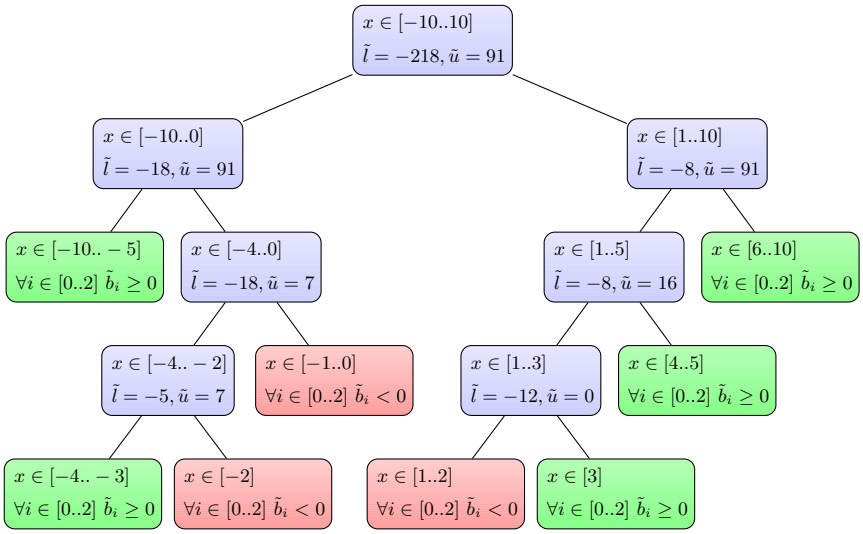

Fig. 2. Description of the refinement of the domain of variable x in (46)

4 Examples

The first motivating example proposed in Section 1 is

```
X in -10..10, X^2 #>= 9
```

and such a constraint can be rewritten in canonical form and interpreted as

$$p(x) = x^2 - 9 \geq 0 \qquad x \in [-10..10]. \tag{46}$$

The trace of the solving process for such a constraint is shown in Figure 2 where light blue nodes are non-terminal nodes, green nodes are terminal nodes with consistent domains and red nodes are terminal nodes with inconsistent domains. In non-terminal nodes, we denote as $\tilde{l}$ and $\tilde{u}$ a negative value and a positive value of $\{\tilde{b}_i\}_{i=0}^{2}$, respectively. The figure shows that after 7 branches the algorithm computes the correct answer

$$x \in [-10.. -3] \cup [3..10]. \tag{47}$$

The second motivating example shown in Section 1 is

```
X in -10..10, Y in -10..10, X*Y #>= 21
```

and such a constraint can be rewritten in canonical form and interpreted as

$$p(x) = xy - 21 \geq 0 \qquad x \in [-10..10] \quad y \in [-10..10]. \tag{48}$$

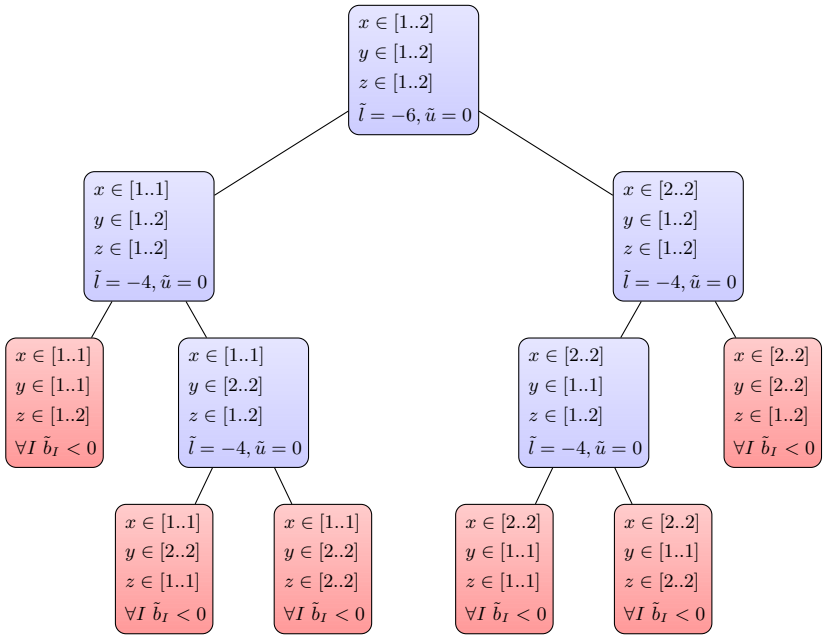

Fig. 3. Description of the refinement of the domain of variable x, y and z of in (50).

The trace of the solving process is too complex to be shown because it counts 41 branches. After all such branches, the algorithm computes the correct answer

$$x \in [-10..-3] \cup [3..10] \qquad y \in [-10..-3] \cup [3..10]. \tag{49}$$

Finally, the following is a well-known example used to show that CLP(FD) solver available in SWI-Prolog (version 6.6.6) sometimes does not capture inconsistent constraints

```
X in 1..2, Y in 1..2, Z in 1..2, X #\=Y, X #\=Z, Z #\=Y.
```

The expected result is `false`, but the solver does not change the constraint, even if `all_different` is used instead of the three inequalities. Such a constraint can be written in canonical form and interpreted as

$$x^2 - 2xy + y^2 - 1 \geq 0 \qquad x^2 - 2xz + z^2 \geq 0 \qquad y^2 - 2yz + z^2 \geq 0. \tag{50}$$

Figure 3 shows the solving process of the proposed algorithm, where, after 5 branches, all terminal nodes are red, i.e., the constraint is inconsistent. In non-terminal nodes, we denote as $\tilde{l}$ and $\tilde{u}$ a negative value and a positive value of $\{\tilde{b}_I\}_{I \leq N}$, respectively, relative to a constraint.

5 Conclusion

In this paper we presented a solver for polynomial constraints over finite domains. Such constraints are expressed as polynomials with integer coefficients whose variables are defined over finite subsets of integer numbers. The proposed algorithm uses the MBF of polynomials to effectively compute the signs of suitable lower and upper bounds of polynomials. It uses such signs to reason on constraints and to remove inadmissible values from the domains of variables. It is worth noting that the algorithm only involves algebraic manipulations over integer numbers because the coefficients of the MBF of a polynomial with integer coefficients are provably integer. The availability of arbitrary precision integer arithmetic is therefore sufficient to ensure that no approximation is involved in the solution process.

The proposed algorithm is open for further developments on possible optimizations, at least, for two important choices that impact significantly on the performances of the solution process. First, the choice of the variable for splitting the current box is crucial. Second, once a variable is selected, the choice of simply halving the current box can be too simplistic. For both choices a number of static and dynamic techniques can be found in the literature (see, e.g., [3]) and we think that both choices could benefit from the peculiar form of constraints that we address. In particular, we plan to investigate how the MFB could help in the computation of the gradient of polynomials, which is a valuable information for both such choices.

The current implementation of the proposed algorithm is available for download (`cmt.dmi.unipr.it/software/clppolyfd.zip`) in terms of a reusable Java library that is also usable as a drop-in replacement of the CLP(FD) solver that ships with SWI-Prolog.

References

1. C. Borralleras, S. Lucas, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio. SAT modulo linear arithmetic for solving polynomial constraints. *Journal of Automated Reasoning*, 48(1):107–131, 2010.
2. R. T. Farouki and V. T. Rajan. Algorithms for polynomials in Bernstein form. *Computer-Aided Geometric Design*, 5(1):1–26, 1988.
3. B. Mourrain and J.P. Pavone. Subdivision methods for solving polynomial equations. *Journal of Symbolic Computation*, 44(3):292–306, 2009.
4. S. Ray and P. S. V. Nataraj. An efficient algorithm for range computation of polynomials using the Bernstein form. *Journal of Global Optimization*, 45:403–426, 2009.
5. J. Sanchez-Reyes. Algebraic manipulation in the Bernstein form made simple via convolutions. *Computer Aided Design*, 35:959–967, 2003.
6. M. Triska. The finite domain constraint solver of SWI-Prolog. In *Procs.* 11^{th} *Int'l Symp. Functional and Logic Programming (FLOPS 2012)*, pages 307–316, Berlin, 2012. Springer.