JBernstein: A Validity Checker for Generalized Polynomial Constraints

Chih-Hong Cheng[1], Harald Ruess[1], and Natarajan Shankar[2]

[1] fortiss GmbH, Guerickestr. 25, 80805 München, Germany
{cheng,ruess}@fortiss.org
[2] SRI International, 333 Ravenswood, Menlo Park, CA 94025, United States
shankar@csl.sri.com

1 Overview

Efficient and scalable verification of nonlinear real arithmetic constraints is essential in many automated verification and synthesis tasks for hybrid systems, control algorithms, digital signal processors, and mixed analog/digital circuits. Despite substantial advances in verification technology, complexity issues with classical decision procedures for nonlinear real arithmetic are still a major obstacle for formal verification of real-world applications.

Recently, Muñoz and Narkawicz [3] proposed a procedure for deciding the validity of quantifier-free nonlinear real arithmetic based on the well-known transformation to Bernstein polynomials. Their Kodiak system outperforms tools based on cylindric algebraic decomposition, including QEPCAD [1] and REDLOG [2], in many cases.

Starting from the algorithms by Muñoz and Narkawicz [3] we extended the approach and implemented the (little) verification engine JBernstein, which checks the validity of finite conjunctions of nonlinear constraints of the form

$$\forall x_0 \in [l_0, u_0], \ldots, x_m \in [l_m, u_m] : \\ (\bigwedge_{j=1}^{n} P_j(x_0, \ldots, x_m) \prec_j c_j) \rightarrow Q(x_0, \ldots, x_m) \prec d$$

P_j and Q are real polynomials over the variables x_0 through x_m, each x_i is interpreted over closed intervals $[l_i, u_i]$ with real-valued lower and upper bounds, c_j and d are real-valued constants, and the symbols $\prec_j$ and $\prec$ are arithmetic inequalities in $\{>, \geq, <, \leq\}$. These constraints support *assume-guarantee* style of reasoning about open systems, with P_i the assumptions on the environment and Q the corresponding guarantee of the system under consideration.

The Java implementation JBernstein includes a number of algorithmic optimizations as described in Section 2, for example, for avoiding unnecessary case splits. In particular, JBernstein uses double-precision floating-point arithmetic of Java in a sound way via constraint strengthening (Section 3).

The resulting runtimes of JBernstein are compared with those reported by Muñoz and Narkawicz [3] for their Kodiak implementation, QEPCAD, and REDLOG. This comparison uses the PVS test suite as compiled by Muñoz and Narkawicz [3]. The experimental evaluation of these optimizations indicates that for complex problems, JBernstein is usually an order of magnitude faster than earlier results by Muñoz and

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 656–661, 2013.
© Springer-Verlag Berlin Heidelberg 2013

Table 1. Performance of JBernstein and other tools. Results from other tools are taken from [3]. The measured unit for execution time is in milliseconds.

Problem	JBernstein	Kodiak [3]	$REDLOG_{rlqe}$ [3]	$REDLOG_{rlcad}$ [3]	QEPCAD [3]	Metit [3]
Schwefel (∀)	159	940	490	> 300000	840	110
Schwefel (∃)	126	280	138900	> 300000	910	(n/a)
Reaction Diffusion (∀)	7	< 10	340	370	10	90
Reaction Diffusion (∃)	3	< 10	340	350	10	(n/a)
Caprasse (∀)	23	290	1750	> 300000	6540	160
Caprasse (∃)	8	310	15060	> 300000	6540	(n/a)
Lotka-Volterra (∀)	5	100	360	450	10	100
Lotka-Volterra (∃)	4	< 10	350	400	10	(n/a)
Butcher (∀)	19	200	420	> 300000	(abort)	(abort)
Butcher (∃)	65	200	360	> 300000	(abort)	(n/a)
Magnetism (∀)	125	73540	670	360	180	540
Magnetism (∃)	115	320	420	360	350	(n/a)
Heart Dipole (∀)	460	7360	> 300000	> 300000	> 300000	> 300000
Heart Dipole (∃)	405	3700	> 300000	> 300000	> 300000	(n/a)

§ For this example, we only ask > -0.0001, as precision can not be maintained for the original property.
‡ For this example, we only ask > 0.0001.

Narkawicz or our un-optimized implementation, and it is two orders of magnitude faster than QEPCAD or REDLOG. For the experimental evaluation we use a similar hardware setting (Intel Core Duo 2.4 Ghz, MacOS, 8 GB RAM) as reported in [3]. The optimization power comes with harder problems, as with these problems refinements are used heavily, and the accumulative effect of optimization comes. These initial experimental results are indeed promising, but, clearly, an extended and improved set of benchmarks is needed to obtain an indication about the asymptotic behavior of these solvers.

JBernstein is implemented in Java without further dependencies. It is freely available under the LGPL version 3 license at

`http://sourceforge.net/projects/jbernstein/`

2 Algorithmic Optimizations

We outline the algorithm of Muñoz and Narkawicz in [3] and the optimizations thereof in JBernstein using simplified constraints of the form

$$\forall x_0 \in [l_0, u_0], \ldots, x_m \in [l_m, u_m] : \phi(x_0, \ldots, x_m) > c\,.$$

The Bernstein approach first performs a *range-preserving* transformation from $[a, b]$ to $[0, 1]$ to obtain a constraint of the form

$$\forall y_0 \in [0, 1], \ldots, y_m \in [0, 1] : \phi'(y_0, \ldots y_m) > c\,,$$

such that $x_i = l_i + y_i(u_i - l_i)$ for all $i \in \{1, \ldots, m\}$ and $\phi(x_1, \ldots, x_m) = \phi'(y_1, \ldots, y_m)$. The polynomial ϕ' is then translated from *polynomial basis* into *Bernstein basis*. For example, $\phi'(y) = 4y^2 - 4y + 1$ has polynomial basis $\{y^2, y, 1\}$. It is rewritten as $\mathbf{1}\binom{0}{2}(1-y)^2$-$\mathbf{2}\binom{1}{2}y(1-y) + \mathbf{1}\binom{2}{2}(y)^2$, with $\{\binom{k}{2}y^k(1-y)^{2-k} | k = 0, 1, 2\}$ being the Bernstein basis. If all coefficients of the polynomial represented in Bernstein basis are greater than c, then the original constraint holds (true). Otherwise one checks if there exists a coefficient of an *endindex* (Bernstein basis vector where every term is either 0

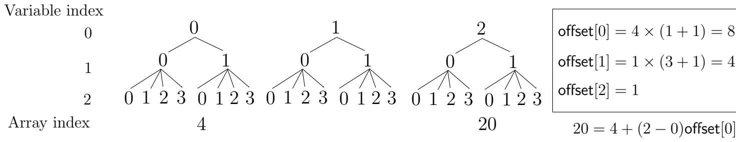

Fig. 1. Index shifting by table look-up

or of highest degree) that is smaller or equal to c. For example, for $\forall y \in [0,1] : \phi'(y) > -3$, as all Bernstein coefficients $1, -2, 1$ are greater than -3, and the property holds. If the property is $\forall y \in [0,1] : \phi'(y) > 2$, as for the first and the last coefficient (in $\{\binom{n}{2} y^k (1-y)^{2-n} | n = 0,1,2\}$, $n = 0, 2$ are endindices), we have $1 \not> 2$, and the property fails to hold at $y = 0$ and $y = 1$ (false).

Lazy refinement. The checking process returns unknown if neither of these two conditions holds, i.e., there exists some non-endindex coefficients less or equal to c. In these cases, the algorithm does range splitting on some chosen variable from $[0, 1]$ to $[0, \frac{1}{2}]$ and $[\frac{1}{2}, 1]$, it generates Bernstein polynomials for each subspace, and it checks if the property does indeed hold for both polynomials. The Bernstein polynomial checker refines a subspace only when a decisive proof has not been found for values within this subspace.

Inputs of the form $\rho > d \rightarrow \varphi > e$ are not considered within the paper by Muñoz and Narkawicz. In JBernstein, such an input is rewritten to its disjunctive form $\rho \leq d \vee \varphi > e$. The checker returns true if one of them is valid, and it returns false if there exists an endindex of both polynomials whose coefficients violate the constraint separately.

Recursive variable exploration. The refinement process involves recursive calls of variable selection and domain partitioning to generate new Bernstein polynomials. For efficiency it is important to avoid variable selections which lead to the generating of an excessive number of polynomials for which the checker returns unknown. As the algorithm is recursive, a naïve deterministic implementation can, given a fixed recursion depth, try to refine the first variable continuously until the end of the recursion depth. It turns out that this strategy often leads to a generation of numerous useless Bernstein polynomials without providing definite proofs. The reason is explained using the following extreme case. Continuously refine only one variable, say y_0, then eventually the generated Bernstein polynomial will have y_0 with its domain converged from $[0, 1]$ to one single point $\alpha \in [0, 1]$. This means that in the original polynomial, the refinement tries to analyze the polynomial by setting x_0 to $l_0 + \alpha(u_0 - l_0)$, whose proving process is, intuitively, nearly as hard as working on the original polynomial. Given a certain budget for maximum allowed recursion, first trying deep recursion over one particular variable is therefore, unlikely to succeed in most cases. Therefore, the implementation of JBernstein uses a round-robin selection strategy on the variables during recursion to avoid worst case scenarios.

Avoiding superfluous computations. The third aspect concerning efficiency focuses on reducing the unit cost for generating every new Bernstein polynomial and checking the property. Generating a new Bernstein polynomial (domain partition) entails generating

all of its coefficients, which themselves are derived from the coefficients of the original Bernstein polynomial (without domain partition). The new coefficients $b^L_{\mathbf{k}}$ for the left partitioned polynomial (for the right polynomial, the formula is similar), where y_j is chosen for partition, is achieved using the following formula [3].

$$b^L_{\mathbf{k}} = \sum_{r=0}^{k_j} \frac{1}{2^{k_j}} \binom{k_j}{r} b_{\mathbf{k}\ with\ [j:=r]} \tag{1}$$

In Eq. 1, for an m-variate polynomial, **k** is a vector of m tuples of positive integers or 0, where each term is smaller than the cardinality (i.e., every **k** is the unique signature of each Bernstein basis vector). k_j is the j-th value of **k**, and $b_{\mathbf{k}\ with\ [j:=r]}$ is the coefficient of the original Bernstein polynomial, where "$\mathbf{k}\ with\ [j := r]$" is an m-tuple that is equal to **k** on every index, except in index j where it has value r. The following optimizations are used by JBernstein:

1. Factor out the constant $\frac{1}{2^{k_j}}$ from summation.
2. Replace the computation of $\binom{k_j}{r}$ by table look-up.
3. Compute $b_{\mathbf{k}\ with\ [j:=r]}$ in an optimized way as follows:
   - Statically store **k** linearly in an array (prior to the range transformation). Figure 1 indicates how vectors are arranged. E.g., when $\mathbf{k} = (0, 1, 0)$, it is located in index 4. Store each $b_{\mathbf{k}}$ linearly in an array following the above order, for every Bernstein polynomial.
   - When the solver iterates the array to create $b^L_{\mathbf{k}}$, the index of **k** is known.
   - To find $b_{\mathbf{k}\ with\ [j:=r]}$, it amounts to finding the index of "$\mathbf{k}\ with\ [j := r]$". Due to our formulation, it can now be translated to an index offset problem: given the index of **k**, what is the offset when replacing the j-th index with value r? The offset is $(r - k_j)\mathsf{offset}[j]$, where $\mathsf{offset}[j] = (Deg_{j+1} + 1) \times \ldots \times (Deg_{m-1} + 1)$, where Deg_s is the highest degree that appears for variable v_s in the polynomial. The array offset can also be computed statically prior to the range-preserving transformation. Figure 1 illustrates the computation of the index of of $(2, 1, 0)$ from $\mathbf{k} = (0, 1, 0)$.
   - Therefore, for $b_{\mathbf{k}\ with\ [j:=r]}$ the solver uses three table look-ups (for $\mathsf{offset}[j]$, k_j and the final value), one substraction, multiplication and addition.
4. Integrate the coefficient generation and the checking process. In each Bernstein polynomial, create an internal Boolean variable field isUnknown that is initially set to false. During the construction, check if a particular coefficient satisfies the property only when it belongs to the endindex or isUnknown equals false. Once if the polynomial is diagnosed as unknown, then there is no need to check for coefficients from non-endindices. Deciding whether a certain index is an endindex is also done statically once and is replaced by table look-up in later computations.

3 Sound Usage of double

We now justify the use of double (double-precision 64-bit IEEE 754 floating point) for data representation. In JBernstein, potential errors due to imprecision of double are handled by the following methodology (for the ease of explanation, we again set the property to be $\forall x_0 \in [l_0, u_0], \ldots, x_m \in [l_m, u_m] : \phi(x_0, \ldots, x_m) > c$):

- Select a positive error-estimate ϵ such as 10^{-5}.
- To return true, in property checking all coefficients shall be greater than $c + \epsilon$.
- To return false, in property checking the solver needs to find a coefficient b_k from an endindex such that $b_k \leq c - \epsilon$.
- If neither of the above two cases holds, the solver either proceeds with domain refinement (when recursion is still allowed) or returns unknown.

The correctness relies on a crucial requirement that the accumulated error for each computed coefficient should never exceed ϵ. Instead of keeping track of the error during the computation, we apply *static analysis* on the algorithm to generate a safe error-estimate that holds for each computed coefficient, based on the polynomial constraint itself and the number of maximum refinement attempts. Due to space limits, we only review the key feature in refining a subspace to generate new Bernstein polynomials (a full text concerning the sound usage of double can be found in the technical report).

For each refinement, recall in Eq. 1 where we have $b^L_{\mathbf{k}} = \sum_{r=0}^{k_j} \frac{1}{2^{k_j}} \binom{k_j}{r} b_{\mathbf{k}\ with\ [j:=r]}$. E.g., when $k_j = 4$, $\frac{1}{2^{k_j}} \binom{k_j}{r}$ equals $\frac{1}{16}, \frac{4}{16}, \frac{6}{16}, \frac{4}{16}, \frac{1}{16}$ (the sum of these values is 1) for $r = 0, 1, 2, 3, 4$. This means that each new coefficient is a weighted sum from old coefficients. If each original coefficient $b_{\mathbf{k}\ with\ [j:=r]}$ has an error-estimate bounded by κ, in an idealized computation, the generated $b^L_{\mathbf{k}}$ also has an error-estimate bounded by κ. This gives an intuition that the growth of error should be very slow (nearly linear to the number of refinement attempts) within the refinement process. The behavior of linear growth is the key ingredient that makes our methodology applicable[1], which is one of the nice properties for Bernstein polynomials.

4 Example

For example, assume-guarantee-style constraints such as

$$\forall x_0 \in [0, 1] : (6x_0 - 1 > 0 \wedge 3x_0 - 1 < 0) \rightarrow 125x_0^3 - 175x_0^2 + 70x_0 - 8 > 0$$

are handled by the textual interface of JBernstein as described in Figure 4:

- The statement `VAR 1` in Figure 4 indicates that the constraint has one variable of name x_0, and statements such as "`BOUND x0 [0,1]`" are used to specify that x_0 is interpreted over the closed interval $[0, 1]$.
- `CONJUNCTION 1` specifies the use of one assume-guarantee rule.
- `ASSUMP 0 2` indicates that the first assume-guarantee rule (indexed 0) has two assumptions $6x_0 - 1 > 0$ and $3x_0 - 1 < 0$.
- For the first assumption (indexed 0) of the first assume-guarantee rule ($6x_0 - 1 > 0$), use "`COEF A0_0 (1) 6`", "`COEF A0_0 (0) -1`", "`SIGN A0_0 GT`", and "`VALUE A0_0 0`" to specify the polynomial.

[1] If every refinement computation brings κ twice as large, in static analysis, applying recursive expansion for small steps like 100 will make an initially small error-estimate prohibitively huge in the lastly generated Bernstein polynomial. It is also important to observe from this example that the division of 16 actually only involves the decrease of exponent by 4 in double without precision loss.

```
## FORALL x\in[0,1]: (6x-1>0 && 3x-1<0) -> (5x-1)(5x-2)(5x-4)>0

## Specify to use one variables x0
VAR 1

## Specify the NUMBER of conjunctions
## Usage: "CONJUNCTION NUMBER"
CONJUNCTION 1

## Specify the NUMBER of assumptions in the INDEX-th conjunction element
## Usage: "ASSUMP INDEX NUMBER"
ASSUMP 0 2
## Specify the coefficient of the polynomial (assumption) 5x+0>0 as
## E.g., A1_0 means the 1st assumption (indexed 0) in the 2nd conjunction (indexed 1)
## First assumption
COEF A0_0 (1) 6
COEF A0_0 (0) -1
SIGN A0_0 GT
VALUE A0_0 0

## Second assumption
COEF A0_1 (1) 3
COEF A0_1 (0) -1
SIGN A0_1 LT
VALUE A0_1 0

## Specify the coefficient of the polynomial (guarantee)
##  (5x-1)(5x-2)(5x-4) = 125x^3-175x^2+70x-8
COEF G0 (3) 125
COEF G0 (2) -175
COEF G0 (1) 70
COEF G0 (0) -8
SIGN G0 GT
VALUE G0 0

## Specify the bound for each variable
BOUND x0 [0, 1]
## Result: FALSE
```

Fig. 2. Assume-guarantee-style constraints in JBernstein

Acknowledgements. We thank Dr. César Muñoz (NASA Langley) for his support and helpful suggestions.

References

1. Brown, C.W.: QEPCAD-B: a program for computing with semi-algebraic sets using CADs. SIGSAM Bull. 37(4), 97–108 (2003)
2. Dolzmann, A., Sturm, T.: REDLOG: computer algebra meets computer logic. SIGSAM Bull. 31(2), 2–9 (1997)
3. Muñoz, C., Narkawicz, A.: Formalization of a representation of Bernstein polynomials and applications to global optimization. Journal of Automated Reasoning (2012)