Bound Propagation for Arithmetic Reasoning in Vampire

Ioan Dragan*, Konstantin Korovin †, Laura Kovács ‡, Andrei Voronkov †

* Vienna University of Technology
ioan.dragan@tuwien.ac.at

† The University of Manchester
korovin@cs.man.ac.uk, andrei@voronkov.com

‡ Chalmers University of Technology
laura.kovacs@chalmers.se

***Abstract*—This paper describes an implementation and experimental evaluation of a recently introduced bound propagation method for solving systems of linear inequalities over the reals and rationals. The implementation is part of the first-order theorem prover Vampire. The input problems are systems of linear inequalities over reals or rationals. Their satisfiability is checked by assigning values to the variables of the system and propagating the bounds on these variables. To make the method efficient, we use various strategies for representing numbers, selecting variable orderings, choosing variable values and propagating bounds. We evaluate our implementation on a large number of examples and compare it with state-of-the-art SMT solvers.**

I. Introduction

Solving a system of linear inequalities over rational and/or real numbers is a well studied problem. The main methods used to solve such systems are Simplex [4] and interior point [21]. Several new methods have recently been developed in the automated reasoning and SMT community. These include the conflict resolution method [13] and GDPLL [18], and the recently introduced bound propagation method [16]. The state-of-the-art SMT solvers, such as Z3 [7] and Yices [10], use Simplex. They also use sophisticated preprocessing algorithms to simplify the input problem.

In this paper we describe the first implementation of the bound propagation algorithm (BPA in the sequel) introduced in [16] and integrated in the first-order theorem prover Vampire [17]. When compared to Z3 and Yices, our experiments show encouraging results. In particular, BPA can solve problems which are difficult for both Z3 and Yices, see Section VII. This is especially promising, given that our implementation of BPA in Vampire does not use any preprocessing steps.

In a nutshell, BPA works as follows. Given a system of linear inequalities over reals, BPA tries to iteratively assign values to some variables and, using these values, derive bounds on other variables of the problem by *bound propagation*. By a bound on a variable x we mean a linear inequality $x \geq c$ or $x \leq c$, where c is a real constant. This process either derives an inconsistent pair of bounds on some variable, or generates an assignment that solves the system. If such a pair is found, BPA builds a *collapsing inequality*, which is used to derive a new bound on a previously assigned variable v, so that the new bound excludes the value previously assigned to this variable. In some cases this new bound is inconsistent with a previously known bound on v, which means that the system is unsatisfiable. Otherwise, we backjump to the point where we selected a value for v and select a new value for v, this time satisfying the newly derived bound.

From this brief description of BPA, one may identify many similarities between BPA and propositional DPLL [20], in particular when it comes to inequality/clause learning, variable selection/ordering and backjumping. Generalising the ideas of DPLL to arithmetic reasoning is also the key ingredient of the conflict resolution [13], [14] and GDPLL [18] method.

The generalised DPLL method of [18], shortly called GDPLL, exploits some ideas from DPLL to reason in the quantifier-free linear real arithmetic. This method tries to find an assignment satisfying a given system of linear inequalities. For doing so, values to variables are assigned in an iterative process and a new clause is learned whenever a conflict is detected. This new clause is used further to resolve conflicts. The theory reasoning part of GDPLL is essentially equivalent to the conflict resolution method [13]. When computing and assigning values of variables, conflict resolution uses an a priori fixed variable ordering. The efficiency of conflict resolution crucially depends on the chosen ordering. The inability to change the ordering during the proof search is probably the main weakness of the method.

The main difference between the aforementioned methods and BPA is that BPA incorporates bound propagation and many techniques from the SAT solving community, in particular dynamic variable ordering, lemma learning and backjumping. While dynamic selection of variables gives BPA flexibility, uncontrolled variable selection coupled with bound propagation can easily yield to a non-terminating algorithm. Therefore termination is a highly non-trivial issue. In [16] it is

shown that under a natural restriction on bound propagation, BPA always terminates. Dynamic variable selection is also used in [6], but without bound propagation which is essential ingredient in BPA. A somewhat similar approach to BPA is described in [11] in the context of integer linear programming. The key difference is that [11] does not use collapsing equalities, which are essential for turning a choice among an infinite number of values for a numeric variable in a given interval into a non-deterministic don't care selection of a value in this interval.

As evidenced by the above described approaches, reasoning about linear arithmetic crucially depends on the underlying theory and heuristics for variable orderings and clause learning. BPA has been introduced only recently in [16] and its practical efficiency is yet unknown. This paper presents the first ever implementation of the method and undertakes the first investigation into understanding the power and limitations of BPA for linear real arithmetic. To this end, we implemented and studied various heuristics for variable orderings, assigning values to variables and learning new inequalities.

The rest of the paper is structured as follows. We start with overviewing some BPA terminology in Section II and illustrate BPA on a small example in Section III.

In Section IV we describe the new mode of Vampire implementing the bound propagation method. This mode, as Vampire itself, runs on all common platforms (Linux, Windows and MacOS). A binary of Vampire for Linux containing the implementation of BPA can be downloaded from:

http://www.complang.tuwien.ac.at/ioan/boundPropagation.

Within this mode, various strategies are used to make bound propagation practically useful. These strategies are presented in Section V and include options for variable orderings, selecting and assigning values to variables and controlling bound propagation. Section VI summarizes the use of BPA for arithmetic reasoning in Vampire.

In Section VII we describe experiments with our implementation of BPA. We used the following three benchmark suites: (i) benchmarks obtained from the SMT-LIB library [3] using the Hard Reality tool [15], (ii) random problems generated by the GoRRiLA tool [15], and (iii) linear optimisation problems from the MIPLIB library [12]. Considering that our implementation is new and yet not well-optimised, the results are highly encouraging: for example, there are problems in these benchmark suites on which BPA outperforms some state-of-the-art SMT solvers. We also outline some interesting directions for improving BPA.

Finally, Section VIII concludes the paper.

II. Preliminaries

In this section we define the notation used in this paper and the relevant terminology. The material of this section is based on that of [16] adapted to our setting.

We denote variables by v, x, y, z and real constants by c, maybe with indices. We call a *literal* a variable x or its negation $-x$ and denote literals by l. By a *linear inequality* we mean an expression $c_1 l_1 + \ldots + c_n l_n + c \geq 0$, where the variables in literals are pairwise different. For simplicity of exposition, in this paper we consider only non-strict inequalities. The method (and our implementation) applies to systems that may contain both strict and non-strict inequalities.

We say that an inequality is *trivial* if it contains no variables. For simplicity, we assume that trivial inequalities are either $-1 \geq 0$ or $0 \geq 0$.

An assignment σ over a set of variables $\{x_1, \ldots, x_n\}$ is a mapping from $\{x_1, \ldots, x_n\}$ to the set of real numbers $\mathbb{R}$, that is $\sigma : \{x_1, \ldots, x_n\} \to \mathbb{R}$. For a linear term q over $\{x_1, \ldots, x_n\}$, we denote by $q\sigma$ the value of q after replacing all variables $x_i \in \{x_1, \ldots, x_n\}$ by the corresponding values $\sigma(x_i)$. An assignment σ is called a solution of a linear inequality $q \geq 0$ if $q\sigma \geq 0$ is true; in this case we also say that σ *satisfies* $q \geq 0$ (otherwise, it *violates* $q \geq 0$). An assignment σ is a *solution of a system* of linear inequalities if it is a solution of every inequality in the system. A system of linear inequalities is said to be *satisfiable* if it has a solution.

A *bound* on a variable x is an inequality of the form $x \geq c$, called *lower bound*, or $-x \geq c$, called *upper bound*. For example, $x \geq 0$ is a (lower) bound on x. The bounds $x \geq c_1$ and $-x \geq c_2$ are said to be *contradictory* or *inconsistent* if $c_1 + c_2 > 0$. A pair of inconsistent bounds on x is called a *conflict*. A (lower) bound $x \geq c_1$ *improves* a (lower) bound $x \geq c_2$ if $c_1 \geq c_2$. Similarly, an (upper) bound $-x \geq c_1$ *improves* an (upper) bound $-x \geq c_2$ if $c_1 \geq c_2$.

III. An Example of BPA

In this section we illustrate the main steps of BPA on a small example.

Consider the following system of inequalities:

$$\mathcal{S} = \{x - 3y \geq 2, x + 3y \geq 1, -x + y \geq 0\}.$$

The BPA algorithm is described in subsection IV-E, in this section we show how it behaves on this system to give the reader the flavour of the method.

(1) Initialise and Decide. In the first step of BPA we pick a variable x and assign to it an arbitrary value within the current bounds for this variable. This value is called the *decision value* for x. In this example there are initially no bounds and we choose to assign 0 to x.

(2) Bound Propagation. Using $x = 0$ and the inequality $-x + y \geq 0$, we derive a new bound $y \geq 0$. The process of deriving a new bound over y using a bound on x is called *bound propagation*. Further, $y \geq 0$ and $x - 3y \geq 2$ yield a new bound $x \geq 2$, which contradicts the decision value assignment $x = 0$. Note that the contradiction was obtained using the asserted assignment $x = 0$, so we cannot yet conclude that $\mathcal{S}$ is unsatisfiable.

(3) Conflict Analysis. Using $-x + y \geq 0$ and $x - 3y \geq 2$, that is, those inequalities from $\mathcal{S}$ that were used to derive the contradiction, we infer an inequality, called the *collapsing inequality*. In this case the collapsing inequality is the bound $-x \geq 1$ for x and can be obtained by eliminating the variable y in the above inequalities (though in general the collapsing inequality is not necessarily a bound). Note that this bound

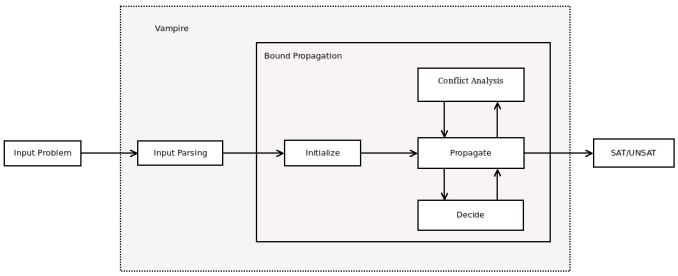

Fig. 1. Bound Propagation for Arithmetic Reasoning in Vampire.

excludes the decision value 0 for x. We next backjump to the decision value assignment $x = 0$, remove this assignment and assert the new bound $-x \geq 1$. Let us note that a collapsing inequality is always implied by S and is used for deriving a bound excluding a previously made assignment.

(4) Bound Propagation. We perform bound propagation using the new bound $-x \geq 1$, which gives us two new bounds on y. Namely, we get $-y \geq 1$ using $x - 3y \geq 2$ and $y \geq \frac{2}{3}$ using $x + 3y \geq 1$. These two new bounds are inconsistent. Since at this stage all the derived bounds do not depend on any assignment, BPA terminates and reports unsatisfiability of $\mathcal{S}$.

IV. Tool Description

This section describes how BPA is implemented in Vampire and how it can be used. We also overview some of the most relevant options implemented to control bound propagation.

Figure 1 shows the workflow of our BPA implementation in Vampire. In the following we describe the workflow of our tool in details. In order to run BPA in Vampire for solving systems of inequalities one has to invoke Vampire with the `--mode bpa` option in the command line.

A. Input problems

The input problems to BPA are systems of linear inequalities over reals or rationals. While BPA can be used for both real and rational linear arithmetic, at the moment rationals are converted internally to floating point numbers.

The input must be represented as a conjunction of linear inequalities. An extension to arbitrary boolean combinations of linear inequalities is under way.

State of the art SMT solvers implement a variety of preprocessing techniques. Our implementation does not preprocess the input problem in any way. Preprocessing is left as future work.

B. Input syntax

The input problem of BPA can be represented either in the standard SMT-LIB format of SMT solvers [3] or in the MIPLIB format used by linear optimisation tools [12]. Users can specify the format of the input problem by using the option `--input_syntax format`. This option has three possible values: `smtlib` for SMT-LIB version 1.2, `smtlib2`, for the SMT-LIB version 2.0 and `xmps` for MIPLIB.

C. Representation of reals

We provide three ways of representing numbers:

(i) by using the `long double` type of C++;
(ii) by using the GNU Multiple Precision library [1];
(iii) by using our implementation of `rational`.

One can use the option:

```
--bp_start_with_precise [on/off]
```

to choose precise numbers from the beginning. Similarly, for starting with rational numbers the option:

```
--bp_start_with_rational [on/off]
```

needs to be used.

When a satisfying assignment is found, it is checked whether it is a solution to the original constraints using the multiple precision representation. If it is not and the `long double` or `rational` representation is used, Vampire restarts the search using the multiple precision representation.

It is possible to build proof objects when BPA derives the contradiction but this has not yet been implemented.

D. Arithmetic in Vampire

Given an input problem, we first translate this problem into the internal input format of Vampire. To reason about the reals, we had to extend Vampire with typed terms and formulas. We also added the built-in sort for reals and built-in functions and predicates over them, including multiplication, addition and standard comparison operators. They are also used in the standard theorem proving mode of Vampire, together with an axiomatisation of reals.

Using these theory specific extensions of Vampire, we translate the input problem into the internal typed Vampire representation and run the bound propagation algorithm on it.

E. Bound Propagation Algorithm

A pseudo-code for our bound propagation algorithm is given in Algorithm 1, highlighting the main steps of our BPA implementation.

To resolve conflicts (i.e. inconsistent bounds), Algorithm 1 calls Algorithm 2.

Based on Algorithms 1 and 2, we now overview the main steps of BPA. Given a system $\mathcal{S}$ of linear inequalities, BPA searches for a solution by applying bound propagation, variable decision and conflict analysis until either a satisfying variable assignment is found, or otherwise a contradiction is derived.

BPA incorporates important DPLL optimisations: backjumping, lemma learning and propagation of bounds, although with essential differences. For example, our analogue of the DPLL learned clause is the collapsing inequality. They are not added to the list of inequalities but used for backjumping. The main steps of our BPA implementation are discussed below. We will highlight main differences between DPLL and BPA in our presentation.

Initialise. We start by collecting initial bounds for variables from $\mathcal{S}$ (line 3 of Alg. 1). Let $\mathcal{B}$ denote the obtained set of

Algorithm 1 The BPA Algorithm

```
1: Input: a set of linear inequalities S
2: Output: satisfying assignment/"unsatisfiable"
3: (Initialise) collect input bounds B ;
4: set decision level DL := 0;
5: while no solution found do
6:     (Propagate) propagate bounds;
7:     if conflict then
8:         (Conflict Analysis)
9:         call Conflicts Analysis (Alg. 2);
10:    else
11:        if there exist variables without decision then
12:            (Decide)
13:            DL := DL + 1;
14:            select next decision variable v (the decision
15:                  variable of level DL);
16:            select a decision value d for v;
17:            add decision bounds v ≥ d and −v ≥ −d
18:                  to B;
19:        else return the map from the variables to
20:                  their decision values as a solution;
21:        end if
22:    end if
23: end while
```

Algorithm 2 Conflict Analysis in BPA

```
1: while there are conflicting variables do
2:     build the collapsing inequality CI;
3:     if CI is −1 ≥ 0 then
4:         return "unsatisfiable";
5:     end if
6:     CV := variable in CI with the maximal decision level;
7:     DL:= decision level of CV;
8:     backjump to the decision level DL;
9:     add the new bound on CV generated by CI to B;
10:    select a new decision value for CV;
11:    add new decision bounds on CV to B;
12: end while
```

bounds. We do not assume that each variable has an initial bound. For example, in Section III we initially have $\mathcal{B} = \emptyset$ as the set $\mathcal{S}$ initially contains no bounds.

When the BPA algorithm is run, the set $\mathcal{B}$ will be changing. In addition to the bounds, we also store *assignments* of values to decision variables. An assignment is represented by an equality $v = c$ which is also treated as a pair of bounds $v \geq c$ and $v \leq c$ in $\mathcal{B}$. We can assume that initially the set of assignments is empty.

The decision level (DL) of BPA corresponds to the number of decision value assignments to variables. DL is initially assigned to 0 (line 4 of Alg. 1) – see also the *Decide* step.

Propagate. We use the bounds in $\mathcal{B}$ to derive new bounds (line 6 of Alg. 1), which are added to $\mathcal{B}$. The new bounds are logical consequences of $\mathcal{B}$ and $\mathcal{S}$. The process of deriving new bounds using other bounds is called *bound propagation*. Unlike DPLL, bound propagation can become non-terminating, for example by deriving ever improving bounds for the same variable (e.g. $x \geq 0$, $x \geq 1$, $x \geq 2$, etc). For this reason, in our BPA implementation we perform *limited bound propagation*, as described below.

If bound propagation derives an inconsistent pair of bounds $x \geq c$ and $x \leq d$ with $c > d$, then we proceed to *Conflict Analysis* (line 9 of Alg. 1). Otherwise, if some variables are not assigned, we move to the *Decide* step and select the next decision variable and a value for it (lines 13-17 of Alg. 1). Finally, if all variables of $\mathcal{S}$ are assigned, we report satisfiability of $\mathcal{S}$ and output the assignment (line 19 of Alg. 1).

Limited bound propagation. To ensure termination of BPA, we perform limited bound propagation in our BPA implementation (line 6 of Alg. 1). To this end, we use the option

`--bp_bound_improvement_limit` k

to specify the upper limit k on the number of improved bounds derived by bound propagation on the same variable at the current decision level. The value k is a positive integer, its default value is set to 3. Bound propagation will terminate as soon as some variable's bound was improved k times.

Our choice of the default value was motivated by our experiments: it turned out that in most cases value 3 (and sometimes 2) yields the best results.

Conflict Analysis. At this stage we derived an inconsistent pair of bounds on a variable (line 9 of Alg. 1). We analyse these inconsistencies (conflicts), as summarised in Algorithm 2. If there are no assigned variables, we report unsatisfiability (line 4 of Alg. 2). Otherwise, we analyse the derivation of these bounds and compute the so-called *collapsing inequality* of this derivation (line 2 of Alg. 2). The collapsing inequality is implied by $\mathcal{S}$ and can be used to derive a bound b on a previously assigned variable x such that b contradicts to the existing assignment for x. We backjump by removing all assignments made since the assignment to x and all bounds derived using these assignments (lines 6-9 of Alg. 2). We then add b to $\mathcal{B}$ (lines 10-11 of Alg. 2) and go to the *Propagate* step. It is possible to modify this method by using the collapsing inequalities also in the propagation step. This means that during the propagation step one has to take into consideration also the collapsing inequalities. This idea is similar to clause learning used by SAT solvers. One can enable this method by:

`--bp_add_collapsing_inequalities on`

During our experiments we noticed some improvement by adding collapsing inequalities, but they can also cause slowdown because bound propagation becomes more expensive.

Decide. At this stage, we have no inconsistent bounds but satisfiability of the system is not yet established (line 11 of Alg. 1). We therefore pick an unassigned variable x. Since the set of bounds $\mathcal{B}$ is satisfiable, so is the current set of bounds for x. We assign to x a *decision value* satisfying these bounds,

increment the decision level and go to the *Propagate* step again (lines 13-17 of Alg. 1).

V. Strategies for Variable and Value Selection

When selecting variables and their values, our BPA implementation uses no a priori fixed variable ordering. Picking the "right" variable and choosing its value during bound propagation clearly affects the efficiency of BPA. We studied and implemented several strategies for variable and value strategies; some of these strategies were inspired by SAT solving heuristics [8], [19], as discussed in the rest of this section. Effects of variable ordering is also studied in the framework of algebraic computation [5], [9]; integrating some of these heuristics in BPA is an interesting task for future work.

A. Variable selection

In this subsection we discuss different variable selection strategies for BPA implemented in Vampire. To specify the variable selection strategy in Vampire, one should use the option:

```
--bp_variable_selector value
```

where `value` specifies which strategy should be used (see below). If this option is omitted, BPA uses the default random selection.

In the sequel we will refer to yet unassigned variables as *eligible variables*. Below we describe various values for the variable selection option of BPA:

(`random`) Pick a random eligible variable.

(`first`) Pick the first, in some fixed predefined order, eligible variable.

(`tightest_bound`) Pick an eligible variable with the tightest bound, that is, a variable with the smallest difference between its upper and lower bounds.

(`conflicting`) This strategy, inspired by the VSIDS heuristics of [8], [19], picks an eligible variable which appears most often in conflicts.

(`conflicting_and_collapsing`) Pick an eligible variable which appears most often both in conflicts and collapsing inequalities.

(`recent_conflicting`) Pick an eligible variable which appears most often in recent conflicts.

(`recent_collapsing`) Pick an eligible variable which appears most often in recent collapsing inequalities.

B. Assigning values to variables

Different *decision values* can affect both the number of steps performed by the algorithm and the size of numbers involved in bound computations. Both are important for efficiency of the algorithm, therefore we implemented several strategies for value selection.

To specify the variable value selection strategy in Vampire one should use the option:

```
--bp_assignment_selector value
```

where `value` specifies which strategy should be used.

In addition to the value selection, we also implemented a mechanism to keep track of previously assigned values to variable, which can be toggled using:

```
--bp_conservative_assignment_selection [on/off]
```

The conservative value selection was considered in order to speed up the value selection computation. By default, this option has the value `off`. When this option is `on`, we store the history of all variable assignments. Upon the backtrack in BPA (see Alg. 2), we first try to find an old value of the variable which satisfies new bounds and only if there is no such value we choose a new variable value according to the strategy specified by `--bp_assignment_selector` option.

In what follows we describe some of the BPA strategies for assigning values to decision variables. For simplicity, if a variable has no upper bound, we will sometimes consider ∞ as its upper bound, and likewise $-\infty$ for the lower bound. We will call the *interval* for a variable the set of all values between the lower and the upper bound.

Random value assignment (`random`). If the variable has both bounds, we pick a random value between these bounds. If the upper bound is missing, the algorithm behaves as if some large positive number was the upper bound, and likewise for the lower bound.

This strategy is currently the default one.

Smallest absolute value assignment (`smallest`). This strategy picks the value with the smallest absolute value between the lower and upper bounds of the variable. If 0 belongs to the interval for the variable, we pick 0. If both bounds are positive, we pick the lower bound as the variable value; otherwise we pick the upper bound. If the selected lower bound is strict, we add a small number δ to it, and likewise for the upper bound.

Alternating lower and upper bound (`alternating`). This strategy implements alternations of picking a lower or upper bound of a variable. In case when the bound on the variable is strict, the variable value picked is the bound plus/minus some small value δ. If the selected bound is missing, a large number is taken instead.

There are also similar values `lower_bound` and `upper_bound` for this option. For example, when `upper_bound` is used, the bevaviour is as follows. Where is no upper bound, a large positive value is chosen. If there is a non-strict upper bound, this bound is chosen. Otherwise when the upper bound b is strict, we choose $b - \delta$ for some small δ.

Middle value (`middle`). This strategy picks the arithmetic mean of the interval defined by the lower and upper bound of a variable. If the upper bound is missing some very large positive value is used instead, and similar for the lower bound.

Tight (`tight`). This strategy pickes a value such that it differs from the upper or lower bound by a small δ. If both bounds are present, we randomly choose which one to use. If none is present, we choose 0.

Binary decomposition (`bmp`). Given the lower and upper bound of a variable, within this strategy we start by computing the arithmetic mean of the integer approximation of the lower

and upper bounds. That is, the arithmetic mean of the floor and ceiling of respectively the lower and upper bound is calculated. If this value lies within the interval defined by the lower and upper bounds, we assign it to the variable. Otherwise, if this value is greater than the upper bound, we compute the arithmetic mean of this value and the floor of the lower bound, and re-iterate the process. Otherwise, if the value is smaller than the lower bound, we compute the arithmetic mean of the interval defined by this value and the ceiling of the upper bound, and re-iterate the process again.

The advantage of using this strategy is that division by 2 is precise, up to a certain limit, and cheap.

Continued fraction decomposition (`cfd`). This strategy is based on the continued fraction decomposition of rationals. The method implemented is based on iteratively representing the number as the sum of integer part and the reciprocal of the fractional part. In order to pick a good value in between two numbers one has to iteratively decompose both numbers and stop at the point where the integer part of the numbers first differ. When this happens we start computing the result of fractions obtained until that point. The procedure ensures that we are always picking a rational value with the smallest denominator and numerator among all rationals in the interval. Using this method we can pick the value with the best rational representation in the interval.

In particular, this method always picks an integer value, if it exists in the interval. This is convenient due to the fact that working with integer numbers is much less expensive in terms of speed and memory consumption. The major bottleneck associated to this method is that it requires in some cases a considerable number of extra computation steps. Our implementation of the method is designed in such a way that cheap operations are preferred, meaning that we are trying to avoid division and multiplication as much as possible.

C. Combinations of strategies

During experiments we extensively evaluated many different combinations of the previously described options. In many cases different combinations solve different sets of problems. Some combinations of options prove to solve more problems than others. Let us note that, even if a strategy solves only a small number of problems, we cannot qualify it as a bad strategy since some problems could be solved only by this strategy.

There are however a few combination of strategies which turned out to perform the best in most of our experimenst. For example, the combination of the assignment selector `tight`, `cfd` and the variable selector `tightest_bound` manages to find all unsatisfiable cases, but on satisfiable cases does not give the best results. Although in general this strategy is not the best for satisfiable problems, we found a few examples that could be solved only by these value selections strategies. Using `cfd` in combination with internal conversion of native numbers to rationals or precise proves to solve most of the problems. Experimenting with strategies which dynamically adjust options during the run of the algorithm is an interesting task to be investigated as future work.

VI. Tool Usage and Implementation

To use BPA in Vampire for solving systems of linear inequalities, one should execute the following command:

```
vampire  --mode bpa problem
         --input_syntax [smtlib/smtlib2/xmps]
         --bp_start_with_precise [on/off]
         [Optional Parameters]
```

where the input parameter `problem` is the problem to be solved in the corresponding syntax and the `bp_start_with_precise [on/off]` flag control the number representation used in BPA. In the same manner one can use `bp_start_with_rational [on/off]`. The `Optional Parameters` refer to the strategies for selecting variables and their values, as discussed in Section V.

For controlling the time limit when running Vampire with BPA, one should use the `--time_limit seconds` option. The default time limit of BPA is 60 seconds.

Our BPA implementation adds an arithmetic decision procedure to Vampire. We extended Vampire with the new built-in sort of reals and built-in theory symbols for them. We added theory axiomatisations for these symbols and extended Vampire with typed first-order formulas. We also changed the SMT parser of Vampire to read SMT-LIB problems for linear real arithmetic. Further, we implemented the BPA algorithm as described above. As for data structures and memory management, we used the standard libraries, data structures and memory allocations functions of Vampire [17]. All together, the BPA implementation in Vampire contains about 8500 lines of C++ code.

VII. Experiments

We evaluated our BPA implementation on a large number of examples, as detailed below. All experiments reported in this section were performed with a 60 seconds time limit, on the Infragrid infrastructure of the West University of Timişoara [2]. This grid contains 100 machines, each of them equipped with an Intel Quad-core, with 2.00 GHz frequency and 14GB RAM per CPU.

Benchmarks. In our experiments, we used the following set of examples.

(i) We ran BPA on 128 problems generated by using the Hard Reality tool [15]. These problems were generated by the tool by using the QF_LRA benchmark suite of the SMT-LIB library [3]. The reason why we could not directly run BPA on SMT-LIB problems is that SMT-LIB examples have a non-trivial Boolean structure. Hard Reality uses an SMT solver and extracts from SMT problems systems of linear inequalities hard for this solver.

(ii) We generated 21,473 hard random linear arithmetic problems using the GoRRiLa tool [15], with 60 variables and 100 inequalities on average.

Solver	Sat	Avg. Time	Unsat	Avg. Time	Unknown
Vampire	2797	0.362	17100	0.236	1576
Z3	3193	0.072	18205	0.144	75
Yices	3206	0.001	18267	0.001	0

TABLE I
EXPERIMENTS ON PROBLEMS GENERATED WITH GORRILA.

Solver	Sat	Avg. Time	Unsat	Avg. Time	Unknown
Vampire	33	5.23	18	3.78	77
Z3	76	2.89	22	3.40	30
Yices	85	9.90	26	6.02	17

TABLE II
EXPERIMENTS ON THE SMT-LIB PROBLEMS GENERATED WITH HARDREALITY.

(iii) We also evaluated BPA on 224 linear optimisation problems taken from the MIPLIB library of mixed integer problems [12]. While these examples contain both integer and Boolean variables, we treated them as real variables and added 0 and 1 as the lower and the upper bounds for Boolean variables.

Our examples can be accessed online at the url of our tool: http://www.complang.tuwien.ac.at/ioan/boundPropagation.

Evaluation and strategies. We compared the Vampire results with the Simplex-based reasoners of Z3 and Yices. Both Z3 and Yices were run with the 60 seconds time limit. We used version 3.2 of Z3 and version 1.0.36 of Yices.

Strategies. We used BPA with various combinations of options. These strategies included the random, tight, conflicting, collapsing and conflicting variable selectors, as well as the random, middle, binary decomposition, continued fraction decomposition and alternating value selectors. We also used limited bound propagation with values 3, 4, 5, and 6.

We also compared all strategies presented in this paper among each other on the MIPLIB benchmarks. From the experiments we notice that choosing a good assignment selector makes a big difference when compared with the default one. Best results were obtained using the continued fraction decomposition (**cfd**) assignment selector. Alongside this selector, also the one based on binary decomposition (**bmp**) selector and the upper bound (**upper_bound**) selector proved to berform best. Also an important role is played by the variable selector, in this case thightest bound (**thightest_bound**) option proves to perform best.

We also observed that choosing an appropriate value as the upper limit of bound propagation `--bp_bound_improvement_limit` option is essential. In the one to one comparison of the options we obsereved that the best performing values for this option are 2,3,4. From our experiments we noticed that choosing bigger values for this option does not help us solve more problems.

Results. We summarise our experiments in Tables II-III. For each table, the first column lists the name of the solver. Columns two and four show the number of problems whose satisfiability (SAT), respectively unsatisfiability (UNSAT) were proved, whereas columns three and five give the respective average solving times in seconds. Column six lists the number of problems that could not be solved within 60 seconds.

Solver	Sat	Avg. Time	Unsat	Avg. Time	Unknown
Vampire	79	6.43	28	4.54	117
Z3	96	6.20	25	2.13	103
Yices	103	3.44	26	0.31	95

TABLE III
EXPERIMENTS ON MIPLIB PROBLEMS.

Table II describes our results on the 128 SMT-LIB examples we tried. The 51 problems solved by Vampire were a subset of the ones solved by Z3 and Yices. The problems solved by Vampire using BPA are not all solved using a single strategy. We have tested approximatively 3000 combinations of options on the set of problems. The numbers presented in the table represent the total number of problems which could be solved by Vampire with BPA. Also Vampire was not able to solve 77 problems from this problem set. We believe that a relatively weak performance of Vampire on these benchmarks is due to the absence of preprocessing, since the benchmarks contain many redundancies.

We further evaluated BPA on 21,473 random hard theory problems generated by GoRRiLa. The number of variables in these problems was around 60, and each problem contained about 100 inequalities. In order to obtain these results we used the predefined values for each of the options. Table I reports on these experiments. All together, Vampire solved 19,897 problems with a very small average solving time, and timed out on 1576 problems. All problems solved by Vampire were also solved by Yices, however there were 75 problems solved by Vampire but not by Z3. These 75 problems required less than 0.01 seconds solving time on the average in Vampire. It turned out that Z3 can solve all these 75 problems with the time limit of 240 seconds.

Finally, Table III describes our results on 224 problems taken from the MIPLIB library. These examples encode optimisation problems coming from academic and industrial applications of mixed integer linear programming. benchmark suite. The MIPLIB problems we used contained thousands of variables and inequalities, each problem being over a few MB in size. Since some of these problems contain integer and/or Boolean variables, we created their relaxations, where all variables are real. Further, we converted optimisation problems into corresponding satisfiability problems.

To run Vampire on these MIPLIB examples, we interfaced Vampire with a MIPLIB parser and added an output of such problems in the SMT-LIB syntax. The 107 problems solved by Vampire contained in average 1526 variables and several thousands of inequalities. All together we found 8 problems that could not be solved by Z3 within the time limit of 60 seconds but can be solved by Vampire. It is also interesting that Vampire was better than Z3 and Yices on unsatisfiable problems. The results presented in the table represent the total

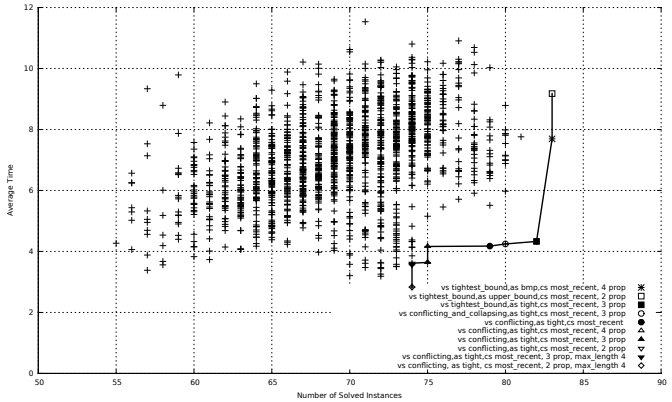

Fig. 2. Comparison between different strategies.

number of problems which Vampire was able to solve. In order to obtain these results we ran Vampire with the same combination of strategies as for problems presented in Table II. for the options.

Summarising, Table III shows that the first implementation of BPA in Vampire gives competitive results when compared to SMT solvers and performs relatively well on very large examples coming from applications.

Comparison of strategies. We also compared the performace of BPA using different strategies on the MIPLIB examples. Figure 2 summarizes our results. The X-axis of Figure 2 represents the total number of solved instances by each of the strategies. The Y-axis represents the average time needed for BPA to solve those instances. We also computed and plotted the Pareto line for minimization of average time and maximization of solved instances. Strategies which are present in the lower right corner of the figure represent best strategies, because they solve a large number of problem instances in the smallest amount of average time.

Figure 2 shows that for these experiments the best combination of strategies resulted from using the `tightest_bound` variable selection together with the `bmp` assignment selector and 2 propagation updates per variable before bound propagation; with this strategy combination we managed to solve the highest number of problems. A variation of these strategies where we use `tight` instead of `bmp` and 3 propagation updates also performs relatively well when it comes to solving time of BPA.

Further improvements. Our BPA implementation requires further experimentation and improvements. In particular, we are interested in (i) computing heuristics for choosing "good" values for options in order to address different *classes of problems*; (ii) strategies for bound propagation; (iii) extend the parsers so that one can start with constraints containing rationals; and (iv) generation of better collapsing inequalities. We believe that proper settings for (i)–(iv) can speed BPA up by orders of magnitude, similar to how it happened with the SAT solving technology in the past. Finally, one should investigate the best ways of running BPA in an SMT solver, where BPA has an advantage that its steps can be interleaved with the SAT solver steps, including backjumping and constraint propagation.

VIII. Conclusions

We describe how bound propagation is implemented and used in Vampire for solving systems of linear inequalities. Also we made an overview of options we have implemented. When compared to state-of-the-art SMT solvers, our experiments show encouraging results. We also discuss possible ways of improving the bound propagation algorithm, inspired by the recent advances in SAT solving.

Acknowledgments

We thank Kryštof Hoder for useful discussions and comments. We also acknowledge funding from the FWF grants S11410-N23 and T425-N23 and the WWTF grant ICT C-050. Konstantin Korovin is supported by a Royal Society University Fellowship.

References

[1] "http://gmplib.org," The GNU Multiple Precission Arithmetic Library.
[2] "http://hpc.uvt.ro/infrastructure/infragrid/," HPC Center - West University of Timisoara.
[3] C. Barrett, A. Stump, and C. Tinelli, "The Satisfiability Modulo Theories Library (SMT-LIB)," `www.SMT-LIB.org`, 2010.
[4] A. R. Bradley and Z. Manna, *The Calculus of Computation - Decision Procedures with Applications to Verification*. Springer, 2007.
[5] C. W. Brown and J. H. Davenport, "The complexity of quantifier elimination and cylindrical algebraic decomposition," in *Proc. of ISSAC*, 2007, pp. 54–60.
[6] S. Cotton, "Natural Domain SMT: A Preliminary Assessment," in *Proc. of FORMATS*, 2010, pp. 77–91.
[7] L. de Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *Proc. of TACAS*, 2008, pp. 337–340.
[8] N. Dershowitz, Z. Hanna, and A. Nadel, "A Clause-Based Heuristic for SAT Solvers," in *Proc. of SAT*, 2005, pp. 46–60.
[9] A. Dolzmann, A. Seidl, and T. Sturm, "Efficient projection orders for cad," in *ISSAC*, 2004, pp. 111–118.
[10] B. Dutertre and L. de Moura, "A Fast Linear-Arithmetic Solver for DPLL(T)," in *Proc. of CAV*, 2006, pp. 81–94.
[11] D. Jovanovic and L. de Moura, "Cutting to the Chase Solving Linear Integer Arithmetic," in *Proc. of CADE*, 2011, pp. 338–353.
[12] T. Koch, T. Achterberg, E. Andersen, O. Bastert, T. Berthold, R. E. Bixby, E. Danna, G. Gamrath, A. M. Gleixner, S. Heinz, A. Lodi, H. Mittelmann, T. Ralphs, D. Salvagnin, D. E. Steffy, and K. Wolter, "MIPLIB 2010," *Mathematical Programming Computation*, vol. 3, no. 2, pp. 103–163, 2011.
[13] K. Korovin, N. Tsiskaridze, and A. Voronkov, "Conflict Resolution," in *Proc. of CP*, 2009, pp. 509–523.
[14] ——, "Implementing Conflict Resolution," in *Ershov Memorial Conference*, ser. Lecture Notes in Computer Science, vol. 7162. Springer, 2012, pp. 362–376.
[15] K. Korovin and A. Voronkov, "GoRRiLA and Hard Reality," in *Proc. of Ershov Memorial Conference (PSI)*, 2011, pp. 243–250.
[16] ——, "Solving Systems of Linear Inequalities by Bound Propagation," in *Proc. of CADE*, 2011, pp. 369–383.
[17] L. Kovács and A. Voronkov, "First-Order Theorem Proving and Vampire," in *Proc. of CAV*, 2013, pp. 1–15.
[18] K. L. McMillan, A. Kuehlmann, and M. Sagiv, "Generalizing DPLL to Richer Logics," in *Proc. of CAV*, 2009, pp. 462–476.
[19] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an Efficient SAT Solver," in *Proc. of DAC*, 2001, pp. 530–535.
[20] R. Nieuwenhuis, A. Oliveras, and C. Tinelli, "Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(*T*)," *J. ACM*, vol. 53, no. 6, pp. 937–977, 2006.
[21] A. Schrijver, *Theory of Linear and Integer Programming*. John Wiley and sons, 1998.