Advanced Analysis Techniques
SMT Solvers

Michał Terepeta

January, 2012

Abstract

The aim of this report is to provide a brief survey of current approaches to satisfiability of Difference Logic (DL) and Unit Two Variable Per Inequality (UTVPI) constraints. First the report covers DL with the basic algorithms and complexity results, then goes on to describe the UTVPI case with recent algorithms. In both cases satisfiability as well as equality generation is covered. Furthermore the report is accompanied by a proof-of-concept implementation of UTVPI theory solver for the Z3 SMT.

Contents

1 Introduction

In many software verification projects, it is necessary to check for satisfiability of certain constraints. However, propositional satisfiability, is often unsuitable for direct use in cases that involve e.g. arithmetic constraints. This is one of the reasons for growing interest in SMT (Satisfiability Modulo Theories). SMT solvers provide a much richer language (or actually languages) that one can use to communicate the constraints. The commonly available theories include free/uninterpreted functions, equality, arrays, difference constraints, linear arithmetic, etc.

Many of these theories naturally arise in software verification, however, they rarely occur just by themselves — more often the constraints are defined over a combination of various theories. This leads us to one of the main achievements in SMT — the Nelson-Oppen framework [12] that specifies how one can combine various theories and still get a decidable algorithm for satisfiability. Therefore modern SMT solvers are capable of solving the problem of satisfiability of constraints over many different theories. This is one of the reasons of why more and more static analyzers/software model checkers use SMT solvers to prove (or disprove) various properties about analyzed software.

Finally there is, as always in case of software verification, the question of performance. Today's SMT solvers use many very specialized algorithms for various theories. As an example consider the fact that many SMT solvers have a separate engine for linear arithmetic and for difference logic (even though difference logic is a special case of linear arithmetic). The reason for that is simple — one of the best algorithms for linear real arithmetic called Simplex is exponential in the worst case, whereas the problem of satisfiability of linear integer arithmetic is NP-complete. On the other hand, difference constraints can be solved by using very efficient graph-based algorithms in polynomial time. Therefore specialization can often be quite beneficial. This report is exactly about exploring a specialized solver for UTVPI constraints, which are a slightly more expressive fragment of linear arithmetic yet still allows efficient polynomial satisfiability algorithms.

2 Difference Logic

2.1 Introduction

Difference constraints have the form

$$x - y \leq c$$

where $c \in \mathbb{Q}$ or $c \in \mathbb{Z}$. We will denote the set of constraints as ϕ. Note that since $\leq$ is anti-symmetric, one can express equality $x - y = c$ as two inequalities: $x-y \leq c$ and $c \leq x-y$, where the latter is equivalent to $y-x \leq -c$. Difference Logic is quite interesting from the point of view of SMT solver because constraints of that form arise quite naturally in many situations and quite a few problems can be encoded using them. Examples include arrays bound checking or job scheduling. Furthermore, there are well-known and efficient algorithms that can check satisfiability of such constraints in polynomial time. The current approach to this problem is based on transforming the constraints into graph representation, such that for every constraint of the form $x - y \leq c$ we have an edge $y \to x$ with weight c. We will denote such a graph as G_ϕ. The key result that allows efficient decision procedures is that the constraints are not satisfiable if and only if there is a negative cycle in such a graph [2].

2.2 Algorithms

The basic algorithm used for satisfiability checking, in the context of SMT solvers, is presented in [9]. As already mentioned, the problem is reduced to determining whether there is a negative cycle in the graph representing the constraint. The intuition behind this approach is as follows. Let us have a cycle $x \xrightarrow{c_1} \ldots \xrightarrow{c_n} x$ such that $c_1 + \ldots + c_n < 0$. If we convert it back to the form of inequalities we would arrive at:

$$\begin{aligned} &x - y_1 \leq c_1 \\ &y_1 - y_2 \leq c_2 \\ &\ldots \\ &y_{n-1} - x \leq c_n \end{aligned}$$

Now adding all the inequalities by sides will result in

$$x - x \leq c_1 + \ldots + c_n$$

since the intermediate variables cancel out. Simplifying we get

$$0 \leq c_1 + \ldots + c_n < 0$$

which is clearly a contradiction, indicating that the constraints are not satisfiable.

The usual approach to check if there is a negative cycle in the graph is to run the Bellman-Ford algorithm on the graph. For reference the algorithm is presented in Table 1. Since the algorithm is an SSSP algorithm (Single-Source Shortest Paths) the first step is to add a new vertex v_s and for every other vertex v in the graph an edge $v_s \to v$ with weight equal to zero. This

Table 1: The Bellman-Ford algorithm

Input: source vertex s

Output: δ (maps each vertex to its shortest distance from the source vertex) and p (parent pointer — maps each vertex to its parent in the shortest path tree rooted in s)

```
1. Set $\delta(s) \leftarrow 0$, and for every $v \in V \setminus \{s\}$ set $\delta(v) \leftarrow \infty$. For every $v \in V$ set $p(v) \leftarrow nil$.
2. For $i = 1$ to $|V| - 1$ do
   (a) For each edge $(u, v) \in E$, if $\delta(v) > \delta(u) + w(u, v)$ then
       • Set $\delta(v) \leftarrow \delta(u) + w(u, v)$
       • Set $p(v) \leftarrow u$
3. For each edge $(u, v) \in E$, if $\delta(v) > \delta(u) + w(u, v)$ then return false.
4. Return true.
```

can not introduce new negative cycles and simply makes it possible to detect negative cycles using Bellman-Ford algorithm even if the graph consists of a few disconnected subgraphs. The time complexity of the algorithm is $\mathcal{O}(|V||E|)$, where V is the set of vertices and E the set of edges. However, here we will use n to denote the number of variables and m for the number of constraints, therefore the satisfiability check of a set of constraints can be performed in $\mathcal{O}((m + n)n)$ time (the additional n comes from the added edges with source in v_s, by modifying the algorithm it is possible to achieve $\mathcal{O}(mn)$). Moreover, note that once the Bellman-Ford algorithm returned true, the δ mapping gives us a potential solution to the set of constraints.

Once we discovered that there is a negative cycle in the graph, it is often useful (e.g. to generate theory conflicts) to find the edges that cause it. This can be achieved by using the parent pointers returned by the Bellman-Ford algorithm. If we assume that in the second step of the algorithm it was vertex v such that $\delta(v) > \delta(u) + w(u, v)$ then we can use the pointers to go back in the path. There must be some vertex that will be twice in the list and forms a negative weight cycle.

Finally it is useful to notice that for the above outlined procedures, it does not really matter whether the domain is $\mathbb{Q}$ or $\mathbb{Z}$.

2.3 SVPI

In the above description we have considered only constraints of the form $x - y \leq c$ but it is easy to relax this requirement a bit and also allow $x \leq c$ (i.e. SVPI — Single Variable Per Inequality). This can be done quite easily by simply creating a fresh vertex v_0 and treating rewriting all such constraints as $x - v_0 \leq c$. The above algorithm for satisfiability checking will then work as expected. The only other thing to remember is that δ returned by Bellman-Ford algorithm might have to be shifted in order to get that $\delta(v_0) = 0$. This relies on the property that for any solution δ to the difference constraints $\delta'(v) = \delta(v) + k$ is also a solution (for every k). Intuitively, since we are only interested in the differences between the variables, we can always "shift" the solution.

2.4 Strict inequalities

So far we have limited ourselves only to non-strict inequalities and have not considered strict ones (i.e. $<$ instead of $\leq$) . However, the problem with strict inequalities can be reduced into one with non-strict ones. In case of $\mathbb{Z}$ this is quite trivial ($x - y < 5$ is equivalent to $x - y \leq 4$).

But it is also possible for $\mathbb{Q}$ as presented in [4]. The solution is to use infinitesimal value ϵ and transform the strict inequalities to the non-strict ones. For instance consider $x - y < c$, we can just rewrite that as $x - y \leq c - \epsilon$. The only necessary change is to slightly modify our domain (constants and variable assignments). We need to define $\mathbb{Q}_\epsilon$ to be pairs of rationals such that (c, k) denotes $c + k\delta$. Finally we need to define a few operations for $\mathbb{Q}_\epsilon$:

$$\begin{aligned}
(c_1, k_1) + (c_2, k_2) &\equiv (c_1 + c_2, k_1 + k_2) \\
a \times (c, k) &\equiv (a \times c, a \times k) \\
(c_1, k_1) \leq (c_2, k_2) &\equiv (c_1 \leq c_2) \vee (c_1 = c_2 \wedge k_1 \leq k_2)
\end{aligned}$$

Note that the last line above defines nothing else than standard lexicographical ordering.

2.5 Beyond satisfiability

There are some possible improvements to the above procedure. One of them is to use a different algorithm for detecting the negative cycle. There have been some newer algorithms with slightly better performance than the Bellman-Ford algorithm, see [1] for a survey. For instance [3] uses an algorithm based on [5]. Unfortunately since the experiments were focused on theory propagation, it is hard to say how much this aspect can influence the overall performance.

Apart from that, as already mentioned, SMT solvers rarely work with constraints of just one type and often employ a combination of various theories in the Nelson-Oppen framework [12]. However, one of the requirements of this framework is that all the theory solvers must share the implied equalities. Therefore it is important to have an efficient algorithm also for generating equalities that are implied by the given set of difference constraints. Here we will briefly cover the one presented in [9]. Of course we assume here that the set of constraint is satisfiable (i.e. Bellman-Ford algorithm returned true for the generated graph). A naive way to generate all the equalities is to perform the transitive

Table 2: Algorithm for equality generation

1. Let E' be the set ef edges in the graph such that $e \in E'$ if and only if $sl(e) = 0$
2. Create an induced subgraph $G'_\phi(V, E')$ from $G'_\phi(V, E)$.
3. Calculate strongly connected components (SCC) [13] of the graph.
4. For each SCC S, let $V^s_d = \{x \mid x \in S, \delta(x) = d\}$.
5. For each $V^s_d = \{x_1, \ldots, x_k\}$ we have the equalities $x_1 = x_2 = \ldots = x_k$.

closure of all the constraints and check if we can find $x \leq y$ and $y \leq x$ for some x and y. However, this is quite expensive — in the worst case would require $\mathcal{O}(n^3)$. A more efficient algorithm is presented in Table 2. However, before we discuss it we need to introduce the concept of slack, i.e. for every edge (u, v) its slack is defined as

$$sl(u, v) = \delta(v) - \delta(u) + w(u, v)$$

The algorithm is based on the idea that if we consider only the edges have slack equal to zero then any cycle formed by these edges will have weight equal to zero. So when we have two vertices u and v in the same SCC (i.e. there is a path from u to v and the other way around) and they have the same value associated by δ then we have that the weight of both paths is equal to zero. Which corresponds to $v - u \leq 0$ and $u - v \leq 0$, i.e. $u = v$.

3 UTVPI

3.1 Introduction and comparison with DL

UTVPI stands for Unit Two Variable Per Inequality and is also referred to as Octagons [10]. Here we will be interested mostly in the question of satisfiability of a set of constraints and not in the details of the constraints as an abstract domain. In general the UTVPI constraints are of the form

$$ax + by \leq c$$

where $a, b \in \{-1, 0, 1\}$ and c is either $\mathbb{Q}$ or $\mathbb{Z}$. UTVPI constraints are quite interesting because they are a bit more expressive than difference logic and so allow more problems to be encoded, yet there still exist efficient polynomial decision procedures for them. Furthermore in case of $\mathbb{Z}$ relaxing any requirement of the UTVPI (allowing non-unit coefficients or more variables) leads to NP-complete decision procedures.

The first decision procedure for UTVPI was presented in [7] and showed that transitive closure of the set of constraints with respect to the transitive and in case of $\mathbb{Z}$ also tightening rules (Table 3) is unsatisfiable if and only if it contains a contradiction, like $0 \leq d$ where $d < 0$.

Note the importance of the tightening rule for $\mathbb{Z}$. The additional difficulty of $\mathbb{Z}$ constraints is that they can have solutions that are not in $\mathbb{Z}$. So they might be satisfiable in $\mathbb{Q}$ but not in $\mathbb{Z}$. The tightening rule basically provides additional information that if we have a constraint that is equivalent to $2x \leq d$ and d is odd then we can tighten the bound on x (i.e. we have $2x \leq d - 1$).

Also note that we will not consider here the case of strict inequalities — they can be handled as in case of DL. Instead we will focus more on the case when the domain is $\mathbb{Z}$, because for UTVPI this case is actually more difficult than the one with $\mathbb{Q}$.

3.2 Algorithms

The procedure presented in [7] has been improved by [6] but the worst case time complexity remained the same — $\mathcal{O}(mn^2)$. However, a new approach was proposed in [8] which does not involve the transitive closure and uses a graph representation of the constraints to solve satisfiability (note the similarity to the difference logic). The complexity of the procedure is equal to the complexity of negative cycle detection algorithm. So using Bellman-Ford algorithm it is

Table 3: Inference rules for UTVPI

$$\frac{ax + by \leq c \qquad -ax + b'z \leq d}{by + b'z \leq c + d} \qquad \text{(Transitive)}$$

$$\frac{ax + by \leq c \qquad ax - by \leq d}{ax \leq \lfloor (c + d)/2 \rfloor} \qquad \text{(Tightening)}$$

Table 4: Edges in the UTVPI graph

UTVPI constraint	DL constraints		Graph Edges	
$x - y \leq c$	$x^+ - y^+ \leq c$	$y^- - x^- \leq c$	$y^+ \xrightarrow{c} x^+$	$x^- \xrightarrow{c} y^-$
$x + y \leq c$	$x^+ - y^- \leq c$	$y^+ - x^- \leq c$	$y^- \xrightarrow{c} x^+$	$x^- \xrightarrow{c} y^+$
$-x - y \leq c$	$x^- - y^+ \leq c$	$y^- - x^+ \leq c$	$y^+ \xrightarrow{c} x^-$	$x^+ \xrightarrow{c} y^-$
$x \leq c$	$x^+ - x^- \leq c$		$x^- \xrightarrow{2c} x^+$	
$-x \leq c$	$x^- - x^+ \leq c$		$x^+ \xrightarrow{2c} x^-$	

possible to achieve $\mathcal{O}(mn)$. One of the main ideas behind the approach is to construct a graph with two vertices for each variable.

For every variable x in the set of constraints we have two vertices x^- and x^+ that represent $-x$ and x. We will sometimes use $-v$ to denote x^- if v represents x^+ and the other way around. Now for every constraint we add one or two edges, as presented in the Table 4.

There are a few interesting properties of such a graph. First of all the graph is very similar to the one from decision procedure for DL. Yet it does not need a special vertex v_0 that was is needed it in the constraint graphs for difference logic to handle constraints of the from $x \leq c$. Another important observation about this new graph is that whenever there is an edge (u, v) there also is an edge $(-v, -u)$. Moreover this can be extended to paths — whenever we have a path from u to v, there is a path from $-v$ to $-u$. Apart from that, let δ be a valuation of all the vertices (i.e. a possible solution to the constraints), then whenever we have a shortest path $SP(u, v)$ between two vertices u and v, we know that $\delta(v) - \delta(u) \leq wSP(u, v)$ (where wSP stands for $w \circ SP$, that is the weight of the shortest path). Finally, one of the main result with respect to UTVPI/Octagons is the following [10].

Lemma 1 *A set of UTVPI constraints is unsatisfiable in $\mathbb{Q}$ if and only if the corresponding constraint graph contains a negative weight cycle.*

Note that this is only true for $\mathbb{Q}$ and not for $\mathbb{Z}$. As already mentioned constraints over integers pose additional difficulties since they can have a non-integer solutions. Therefore for the case of integers we need to introduce the concept of tightening edges. They are defined as follows:

$$T = \{(u, -u) \mid wSP(u, -u) \text{ is odd}\}$$

Note that this corresponds to an either $u^+ - u^- \leq c$ or $u^- - u^+ \leq c$, where c is odd. We are interested only in those edges because these are the only ones that can cause the Tightening rule to apply. Therefore for each edge in T we define its weight as follows

$$w_T(u, -u) = wSP(u, -u) - 1$$

Let us denote the graph with those additional edges as $G_{\phi \cup T}$. One of the main contributions of [8] is the lemma below.

Lemma 2 *A set of UTVPI constraint ϕ is unsatisfiable in $\mathbb{Z}$ if and only if the graph $G_{\phi \cup T}$ has a negative weight cycle.*

Table 5: Algorithm for satisfiability of UTVPI constraints

1. Construct G_ϕ graph.
2. Run a negative cycle detection algorithm on the graph.
   (a) If there is a negative cycle, return UNSAT.
   (b) Otherwise the algorithm will return shortest paths from the source vertex v_s to all other vertices. This can be treated as a feasible solution to the constraints. Note that $\delta(v) - \delta(u) \leq w(u, v)$ for all edges (u, v).
   (c) Let E' be the set of edges from G such that $(u, v) \in E'$ iff $\delta(u) - \delta(v) = w(u, v)$.
   (d) Create graph G'_ϕ induced by E'
   (e) Group the vertices into strongly connected components (for instance using Tarjan's algorithm [13], which is $\mathcal{O}(|V| + |E|)$).
   (f) For each vertex $u \in V$, if $-u$ is in the same SCC as u and $\delta(u) - \delta(-u)$ is odd then return UNSAT.
   (g) Return SAT.

The naïve approach would be to actually find and add all the additional edges and run the negative cycle detection algorithm on this new graph. This could be achieved by using either Floyd-Warshall or Johnson's All Pairs Shortest Paths algorithms to find the tightening edges and then use negative cycle detection algorithm. However, [8] presents a bit smarter approach. It is summarized in Tabel 5.

The intuition behind the second part of the algorithm (i.e. SCC computation) is to identify zero weight cycles and check if we can tighten some edge. If so then the zero cycle in G_ϕ will result in a negative one in $G_{\phi \cup T}$ (i.e. after adding the tightening edge). Moreover whenever there is a path P between u and v such that all the edges along the path have slack equal to zero, then $wSP(u, v) = \delta(v) - \delta(u)$. Now if u and $-u$ are in the same SCC, then there is a path from u to $-u$ and from $-u$ to u. Moreover

$$\begin{aligned} \delta(u) - \delta(-u) &= wSP(-u, u) \\ \delta(-u) - \delta(u) &= wSP(u, -u) \\ 0 &= wSP(-u, u) + wSP(u, -u) \end{aligned}$$

So u and $-u$ form a zero weight cycle. If one of the paths is odd, then we know that a corresponding tightening edge will be in T and so from a negative cycle. For more precise description and the formal soundness and completeness proofs we refer to [8].

The solver to be useful in DPLL(T) should produce theory conflicts, in this case which inequalities cause a negative cycle. Then the solver can assert the theory axiom saying that at least one of those equalities must be false. In order to find the inequalities, we must track the reason why a given edge was added to the graph. Then once we detect a negative cycle using e.g. Bellman-Ford algorithm we can simply use the parent pointers to find the cycle. If the unsatisfiability was detected in the second part of the algorithm, then all we

Table 6: Simple algorithm for equality generation

1. Construct $G_{\phi \cup T}$ graph and run a negative cycle detection algorithm to calculate a feasible solution $\delta_{\phi \cup T}$.
2. Let E_0 be a set of edges such that $e \in E_0$ iff $sl(e) = 0$.
3. Create the subgraph G_0 induced by the E_0 set and group the vertices into SCCs.
4. If vertices x^+ and y^+ are in the same SCC and $\delta_{\phi \cup T}(x^+) = \delta_{\phi \cup T}(y^+)$, then report that $x = y$.

Table 7: Improved algorithm for equality generation

1. Construct graph G_ϕ and run the satisfiability algorithm on it. We assume that the algorithm returned SAT and a possible solution δ
2. Calculate $E_2 = \{(u, v) \mid sl(u, v) \leq 2\}$ and the graph G_2 induced by E_2.
3. Let T_2 be the set of tightening edges (initially set to $\emptyset$) that we are interested in.
4. For each vertex v
   (a) Find the path P_v in G_2 from v to $-v$ with the smallest slack (if any). This can be achieved using breadth-first search or Dijkstra's algorithm.
   (b) If P_v exists
      i. Let $wSP(v, -v) = \delta(-v) - \delta(v) + sl(P_v)$.
      ii. If $wSP(v, -v)$ is odd then add the edge $(v, -v)$ to T_2 and assign it weight $w_{T_2}(v, -v) = wSP(v, -v) - 1$.
5. Add all the edge in T_2 to the G_ϕ graph to get $G_{\phi \cup T_2}$.
6. Proceed as in the previous algorithm but with $G_{\phi \cup T_2}$ instead of $G_{\phi \cup T}$.

need to do is to find the path from u to $-u$ and back in the induced graph (this can be done by Breadth First Search) — again the cycle is a proof that there are no solutions in $\mathbb{Z}$.

Finally, as in the case of DL, in order for the theory solver to be used in the DPLL(T) framework, should be capable of generating all equalities that are consequences of the considered inequalities. We will cover here two possible approaches, as introduced in [8]. The first, quite simple way of generating all inequalities is presented in Table 6. The idea behind the algorithm is pretty much the same as for the one for difference logic. That is if we know that $\delta_{\phi \cup T}(x^+) = \delta_{\phi \cup T}(y^+)$ and both vertices are in a cycle of weight zero, then they are equal.

However, this way of calculating the equalities requires us to compute all the tightening edges T and then add them to the graph. As before an improved algorithm would avoid finding all those edges and use only the ones that can lead to a cycle of weight zero. The improved algorithm makes use of an intermediate lemma that is presented below [8].

Lemma 3 *Assuming $G_{\phi \cup T}$ has no negative cycles and if C is a zero weight* *cycle in $G_{\phi \cup T}$ containing a tightening edge $(u, -u)$ then there is a cycle C' in G_ϕ containing u and $-u$ such that $w(C') \leq 2$.*

This builds on another property that whenever there is a cycle C in $G_{\phi \cup T}$ then there is cycle C' with at most two tightening edges, such that $w(C') < 0$ or $w(C') < w(C)$. This gives us the ability to consider, without loss of generality, cycles with at most two tightening edges.

Now the idea behind the finding only the tightening edges that are useful for equality generation is that if the cycle C has at most two tightening edges then there is a cycle C' without those edges and its weight is not greater than two (i.e. each of the tightening edges can decrease the weight by one only, so if we remove them the weight can get larger by at most two). The improved algorithm that taking advantage of this is presented in the Table 7. It only really changes the first step of the previous one — instead of calculating the whole $G_{\phi \cup T}$, it calculates only those tightening edges that are useful, adds them to the G_ϕ graph and then proceeds as before.

4 Implementation

4.1 Introduction

The accompanying UTPVI theory solver for Z3 SMT Solver [18] is implemented in around 1500 lines of C++ (taking advantage of recent C++11 standard [16]). It uses Boost Graph Library [15, 14] for graph manipulation and GMP [17] for arbitrary precision arithmetic. Due to time constraints it is mainly a proof-of-concept and does not have all the features necessary for usage in a combination with other theories. Currently it implements:

- decision procedure for UTVPI (and thus SVPI) for integers, both using basic types of C++ like int, as well as arbitrary precision integers from GMP library
- decision procedure for UTVPI (and thus SVPI) for rationals using arbitrary precision rationals from GMP library[1]
- generation of theory conflicts
- backtracking
- integration with Z3

Initially the implementation was to be based on the one for Fx7 SMT solver [11]. However, it implements an older algorithm described in [6]. Therefore this implementation is not based on the procedure from Fx7 and implements the more recent algorithms from [8] instead.

One of the main highlights of this implementation is the ability to easily instantiate solvers for rationals/integers with various representations. For instance the following snippet creates theories for checking satisfiability for integers that fit in `int`, integers of arbitrary precision and rationals of arbitrary precision respectively.

```
Z3_theory mach_integers  = MkTheory<UtvpiGraphZ, int>(context);
Z3_theory arbt_integers  = MkTheory<UtvpiGraphZ, mpz_class>(context);
Z3_theory arbt_rationals = MkTheory<UtvpiGraphQ, mpq_class>(context);
```

Where `context` is `Z3_context`, `mpz_class` and `mpq_class` are GMP's classes for arbitrary precision integers and rationals respectively[2]. Note that this includes running all the described graph algorithms with weights of the specified types, as well as using the appropriate functions from Z3 to parse and create numerals (in case of `int` we use `Z3_get_numeral_int`, in case of `mpz_class` we need to go through a string and use use `Z3_get_numeral_string`).

The project uses Z3 parser and predefines two predicates `Utvpi` and `Svpi` and introduces additional sort `Sign` with two constants `Minus` and `Plus` that can be used as follows:

```
(declare-fun x () Int)
(declare-fun y () Int)

(assert (or (Utvpi Plus x Minus y (~ 1)) (Utvpi Plus x Minus y 1)))
```

[1]In fact current implementation should work with basic types like `float` or `double`, however, due to their imprecision, this would probably not be very useful.

[2]GMP offers both a basic C as well as somewhat more convenient C++ interface, which overloads basic arithmetic operators.

```
(assert (Svpi Plus y 0))
(assert (Svpi Minus x (~ 1)))
```

This translates to

$$(x - y \leq -1 \vee x - y \leq 1) \wedge y \leq 0 \wedge -x \leq -1$$

Of course this can be easily changed as desired.

The tool also outputs some information on how it is solving the problem. For the above example the output is:

```
Parsing file: test.smt
Parsed the following formula:
(and (or (Utvpi Plus x Minus y (- 1)) (Utvpi Plus x Minus y 1))
     (Svpi Plus y 0)
     (Svpi Minus x (- 1)))
Z3: Assigned (Svpi Plus y 0) to 1
Z3: Assigned (Svpi Minus x -1) to 1
Z3: Push
Z3: Assigned (Utvpi Plus x Minus y -1) to 1
UtvpiGraphZ: Found negative weight cycle without tightening.
UtvpiGraph: Reason for negative cycle:
[..]
SatCheck: UNSAT.
SatCheck: asserting theory axiom:
(not (and (and (and (Svpi Minus x -1) (Utvpi Plus x Minus y -1))
               (Svpi Plus y 0))
          (Utvpi Plus x Minus y -1)))
Z3: Pop
Z3: Assigned (Utvpi Plus x Minus y 1) to 1
Z3: SAT
Z3: Reset
```

Which clearly shows how everything works:

1. First Z3 assigns the two `SVPI` predicates the value true (they are asserted so they must be true for the formula to by satisfiable).
2. Then Z3 makes `Push` callback for the case-split caused by the asserted disjunction.
3. Assigning true for the first disjuncts causes a theory conflict from the UTVPI solver (clearly $y \leq 0 \wedge x \geq 1 \wedge x - y \leq -1$ is unsatisfiable).
4. Z3 reacts making the `Pop` callback and assigning true to the other disjunct.
5. Theory solver returns that this is satisfiable.
6. Thus the formula is satisfiable.

Of course if we modify the formula removing the second disjunct, the result is:

```
Parsing file: test.smt
Parsed the following formula:
(and (Utvpi Plus x Minus y (- 1)) (Svpi Plus y 0) (Svpi Minus x (- 1)))
Z3: Assigned (Utvpi Plus x Minus y -1) to 1
```

```
Z3: Assigned (Svpi Plus y 0) to 1
Z3: Assigned (Svpi Minus x -1) to 1
UtvpiGraphZ: Found negative weight cycle without tightening.
UtvpiGraph: Reason for negative cycle:
[..]
SatCheck: UNSAT.
SatCheck: asserting theory axiom:
(not (and (and (and (Utvpi Plus x Minus y -1) (Svpi Plus y 0))
               (Utvpi Plus x Minus y -1))
          (Svpi Minus x -1)))
Z3: UNSAT
Z3: Reset
```

5 Conclusions

Both Difference Logic and UTVPI seem to be a great compromise for satisfiability checking in the context of modern SMT solvers. They both allow very efficient graph-based algorithms for problems such as satisfiability or equality generation. And at the same time they often naturally arise in software verification (array bounds checking, etc.) or in other problems like job scheduling.

This report provided a brief survey of the current state-of-the-art in the satisfiability checking for Difference Logic and UTVPI. It focused on providing intuition about various algorithms and reasons of their efficiency. Apart from that a proof-of-concept implementation of UTVPI theory solver for the Z3 SMT solver has been developed.

References

[1] Boris V. Cherkassky and Andrew V. Goldberg. Negative-cycle detection algorithms. In Josep Díaz and Maria J. Serna, editors, *ESA*, volume 1136 of *Lecture Notes in Computer Science*, pages 349–363. Springer, 1996.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (3. ed.)*. MIT Press, 2009.

[3] Scott Cotton and Oded Maler. Fast and flexible difference constraint propagation for dpll(t). In Armin Biere and Carla P. Gomes, editors, *SAT*, volume 4121 of *Lecture Notes in Computer Science*, pages 170–183. Springer, 2006.

[4] Bruno Dutertre and Leonardo Mendonça de Moura. A fast linear-arithmetic solver for dpll(t). In Thomas Ball and Robert B. Jones, editors, *CAV*, volume 4144 of *Lecture Notes in Computer Science*, pages 81–94. Springer, 2006.

[5] Daniele Frigioni, Alberto Marchetti-Spaccamela, and Umberto Nanni. Fully dynamic shortest paths and negative cycles detection on digraphs with arbitrary arc weights. In Gianfranco Bilardi, Giuseppe F. Italiano, Andrea Pietracaprina, and Geppino Pucci, editors, *ESA*, volume 1461 of *Lecture Notes in Computer Science*, pages 320–331. Springer, 1998.

[6] Warwick Harvey and Peter Stuckey. A unit two variable per inequality integer constraint solver for constraint logic programming. In *In Proceedings of the Twentieth Australasian Computer Science Conference*, pages 102–111, 1997.

[7] Joxan Jaffar, Michael J. Maher, Peter J. Stuckey, and Roland H. C. Yap. Beyond finite domains. In Alan Borning, editor, *PPCP*, volume 874 of *Lecture Notes in Computer Science*, pages 86–94. Springer, 1994.

[8] Shuvendu K. Lahiri and Madanlal Musuvathi. An efficient decision procedure for utvpi constraints. In Bernhard Gramlich, editor, *FroCos*, volume 3717 of *Lecture Notes in Computer Science*, pages 168–183. Springer, 2005.

[9] Shuvendu K. Lahiri and Madanlal Musuvathi. An efficient nelson-oppen decision procedure for difference constraints over rationals. *Electr. Notes Theor. Comput. Sci.*, 144(2):27–41, 2006.

[10] Antoine Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

[11] Michał Moskal. Fx7. `http://moskal.me/smt/en.html`.

[12] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, 1979.

[13] Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

[14] Boost graph library. `http://www.boost.org/doc/libs/1_48_0/libs/graph/doc/index.html`.

[15] Boost libraries. `http://www.boost.org/`.

[16] C++11. `http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50372`
`http://en.wikipedia.org/wiki/C%2B%2B11`.

[17] Gmp — the gnu multiple precision arithmetic library. `http://gmplib.org/`.

[18] Z3 smt solver. `http://research.microsoft.com/en-us/um/redmond/projects/z3/`.