Journal on Satisfiability, Boolean Modeling and Computation 1 (2007) 91-114

SDSAT: Tight Integration of Small Domain Encoding and Lazy Approaches in Solving Difference Logic

Malay K. Ganai[1] malay@nec-labs.com
Muralidhar Talupur[2] tmurali@cs.cmu.edu
Aarti Gupta[1] agupta@nec-labs.com

[1] *NEC Laboratories America, Princeton, NJ, USA*
[2] *Carnegie Mellon University, Pittsburgh, PA, USA*

Abstract

Existing *difference logic* (DL) solvers can be broadly classified as *eager* or *lazy,* each with its own merits and de-merits. We propose a novel *difference logic* solver *SDSAT* that combines the strengths of both these approaches and provides a robust performance over a wide set of benchmarks. The solver *SDSAT* works in two phases: *allocation* and *solve*. In the *allocation phase*, it allocates non-uniform *adequate* ranges for variables appearing in difference predicates. This phase is similar to previous small domain encoding approaches, but uses a novel algorithm *Nu-SMOD* with 1-2 orders of magnitude improvement in performance and smaller ranges for variables. Furthermore, the *difference logic* formula is not transformed into an equi-satisfiable Boolean formula in a single step, but rather done lazily in the following phase. In the *solve phase*, *SDSAT* uses a lazy refinement approach to search for a satisfying model *within the allocated ranges*. Thus, any partially DL-theory consistent model can be discarded if it cannot be satisfied within the allocated ranges. Note the crucial difference: *in eager approaches, such a partially consistent model is not allowed in the first place, while in lazy approaches such a model is never discarded.* Moreover, we dynamically refine the allocated ranges and search for a feasible solution within the updated ranges. This combined approach benefits from both the smaller search space (as in eager approaches) and also from the theory-specific graph-based algorithms (characteristic of lazy approaches). Experimental results show that our method is robust and always better than or comparable to state-of-the art solvers using similar eager or lazy techniques.

KEYWORDS: *SMT solvers, difference logic, lazy approach, small domain encoding, eager approach, range allocation, abstraction, refinement, decision procedure*

Submitted November 2006; revised March 2007; published June 2007

1. Introduction

Difference Logic (DL) extends propositional logic with predicates of the form $x + c \triangleright y$ where $\triangleright \in \{>, \geq\}$, c is a constant, and x, y are variables of some ordered infinite type *integer* or *real*. All other equalities and inequalities can be expressed in this logic. Uninterpreted functions can be handled by reducing them to Boolean equalities [1]. Difference predicates play a pivotal role in verification of timed systems [2] and hardware models with ordered data structures like queues and stacks, and modeling job scheduling problem [3]. Deciding a *difference logic* problem is *NP-Complete*. Decision procedures based on graph algorithms use a weighted directed graph to represent *difference* predicates; with nodes representing

variables appearing in the predicates and edges representing the predicates. A predicate of the form $x+c \geq y$ is represented as directed edge from node x to node y with weight c. A conjunction of *difference* predicates is consistent if and only if the corresponding graph does not have a cycle with negative accumulated weight. The task for decision procedures is reduced to finding solutions without negative cycles. Note, some decision procedures can decide the more general problem of linear arithmetic where predicates are of the form $\Sigma_i a_i x_i \geq c$ where a_i, c are constants and x_i are variables. *ICS* [4], *HDPLL* [5] , *PVS* [6], and *ASAP* [7] are based on a variable elimination technique like Fourier-Motzkin [8], while most of the recent solvers such as *Ario* [9], *MathSAT* [11], *Simplics* [12], and *Yices* [13] are based on Simplex [14]. Here, we restrict ourselves to a discussion of decision procedures dedicated for *difference logic*.

Satisfiability of a *difference logic* formula can be checked by translating the formula into an equi-satisfiable Boolean formula and checking for a satisfying model using a Boolean satisfiability solver (SAT). In the past, several dedicated decision procedures have taken this approach to leverage off recent advances in SAT engines [15]. These procedures can be classified as either *eager* or *lazy*, based on whether the Boolean model is refined (i.e., transformed) eagerly or lazily, respectively. In eager approaches [16, 17, 18, 19, 20, 21], the *difference* formula is reduced to an equi-satisfiable Boolean formula in a single step and SAT is used to check the satisfiability. Reduction to propositional logic is done either by deriving adequate ranges for formula variables (*a.k.a small domain encoding*) [16, 18, 21] or by deriving all possible transitivity constraints (*a.k.a per-constraint encoding*) [17]. A hybrid method combines the strengths of the two encoding schemes and was shown [19] to give robust performance. In lazy approaches [10, 11, 13, 22, 23, 24, 25], SAT is used to obtain a possibly feasible model corresponding to a conjunction of *difference* predicates, and feasibility of the conjunct is checked separately using graph-based algorithms. If the conjunct is infeasible, the Boolean formula is refined and thus, an equi-satisfiable Boolean formula is built lazily by adding the transitivity constraints on a need-to basis.

Both the eager and lazy approaches have relative strengths and weaknesses. Though the small model encoding approaches [16, 18, 21] reduce the range space allocated to a finite domain, Boolean encoding of the formula often leads to a large propositional logic formula, eclipsing the advantage gained from the reduced search space. Researchers have also experimented with the pseudo-Boolean Solver *PBS* [26] to obtain a polynomial size formula, but without any significant performance gain [20]. In a *per-constraint encoding* [17], the formula is abstracted by replacing each predicate with a Boolean variable, and then preemptively adding all transitivity constraints over the predicates. Often the transitivity constraints are redundant and adding them eagerly can lead to an exponentially large formula. The Boolean SAT solvers are often unable to decide "smartly" in the presence of such overwhelmingly large number of constraints. As a result the advantage gained from reduced search often takes a back-seat due to lack of proper search guidance. Lazy approaches overcome this problem by adding the constraints as required. Moreover, they use advanced graph algorithms based on Bellman-Ford shortest path algorithm [27] to detect an infeasible combination of predicates in polynomial time in the size of the graph. These approaches exploit incremental propagation and efficient backtracking schemes to obtain improved performance. Several techniques have been proposed [11, 23] to preemptively add some subset of infeasible combination of predicates. This approach has been shown

to reduce the number of backtracks significantly in some cases. Note, the feasibility check is based on detection of a negative cycle (negative accumulation of edge weights) in the graph. Potentially, there could be an exponential number of such cycles and eliminating them lazily can be quite costly. Thus, we are motivated to combine the strength of the two approaches as *tightly as possible*.

There have been some previous efforts to overcome limitations in the eager or lazy approaches. In [25], a dynamic predicate learning has been proposed, and was combined with a lazy framework. The basic idea involves detecting shorter negative cycles and adding corresponding predicates dynamically as needed, which can potentially eliminate many longer negative cycles. It has been shown to give good results for specific benchmarks such as *diamond*, which has $\sim 2^n$ cycles, where n is the number of variables. In an effort to combine eager and lazy methods such as *ASAP* [7], ranges are underestimated and iteratively increased until a solution is found or the formula is proved unsatisfiable. A related approach is followed in [28]. The problem encoded after range refinement can be quite different structurally from that before the refinement. This limits the scope of incremental formulation, and hence, the effectiveness of incremental learning [29].

We discuss a robust *difference logic* solver *SDSAT* [30] (Small Domain SATisfiability solver) that combines the strengths of both eager (small domain encoding) and lazy approaches and gives a robust performance over a wide set of benchmarks. Without overwhelming the SAT solver with a large number of constraint clauses and thereby, adversely affecting its performance, we take advantage of both the (finite) reduced search space and the need-to basis transitivity constraints, which are able to guide the SAT solver more efficiently. The solver *SDSAT* works in two phases: *allocation* and *solve*. In the *allocation phase*, it allocates non-uniform *adequate* ranges for variables appearing in difference predicates. This phase is similar to previous small domain encoding approaches, but uses a novel algorithm *Nu-SMOD*, with 1-2 orders of magnitude improvement in performance and smaller ranges for variables. Furthermore, the *difference logic* formula is not transformed into an equi-satisfiable Boolean formula in a single step, but rather done lazily in the following phase. In the *solve phase*, *SDSAT* uses a lazy refinement approach to search for a satisfying model *within the allocated ranges*. Thus, any partially DL-theory consistent model can be discarded if it cannot be satisfied within the allocated ranges. Note the crucial difference: *in eager approaches, such a partially consistent model is not allowed in the first place, while in lazy approaches such a model is never discarded.* Moreover, we dynamically refine the allocated ranges and search for a feasible solution within the updated ranges. This combined approach benefits from both the smaller search space (as in eager approaches) and also from the theory-specific graph-based algorithms (characteristic of lazy approaches). Experimental results show that our method is robust and always better than or comparable to state-of-the art solvers using similar eager or lazy techniques.

Outline: We give a short background on *difference logic* and the state-of-the-art solvers in Section 2. We describe our solver *SDSAT* in detail, highlighting the technical details and novelties in Section 3. This is followed by experiments and conclusions in Sections 4 and 5, respectively.

2. Background: Difference Logic

Difference predicates are of the form $x + c \triangleright y$ where $\triangleright \in \{>, \geq\}$, c is a constant and x, y are variables of some ordered infinite type *integer* or *real*, D. The theory of *difference logic* combined with propositional logic is *NP-Complete*. If all variables are integers then a strict inequality $x + c > y$ can be translated into a weak inequality $x + (c - 1) \geq y$ without changing the decidability of the problem. Similar transformations exist for mixed types, by decreasing c by small enough amounts, determined by remaining constants in the predicates [10]. Note, an inequality of the form $x \triangleright c$, can also be translated into a weak inequality of two variables, by introducing a reference node z. Henceforth, we will consider *difference* predicates of the form $x + c \geq y$.

2.1 State-of-the-art Lazy approach: Negative-cycle detection

We discuss briefly the essential components in the state-of-the-art *difference logic* solvers based on lazy approaches as shown in Figure 1.

2.1.1 Problem Formulation

In this class of decision procedures, a *difference logic* formula φ is abstracted into a Boolean formula φ_B by mapping predicates $x + c \geq y$ and $y + (-1 - c) \geq x$ to a Boolean variable and its negation, respectively (or vice versa, depending on some ordering of x and y.) An assignment (or interpretation) is a function mapping each variable to value in D and each Boolean variable to $\{T, F\}$. An assignment α is extended to map a *difference logic* formula φ to $\{T, F\}$ by defining the following mapping over the *difference logic* predicates, i.e., $\alpha(x+c \geq y) = T$ *iff* $\alpha(x) + c \geq \alpha(y)$. A Boolean SAT solver is used to obtain a consistent assignment for Boolean variables in φ_B. If such an assignment does not exist, it declares the problem *unsatisfiable*. On the other hand, for any satisfying assignment to φ_B, an additional consistency check is required for the underlying *difference logic* predicates. Note, incremental solvers [11, 24, 31] perform this check on a partial assignment to detect conflict early. The problem is declared SAT only when the satisfying assignment is consistent under the check.

2.1.2 Constraint Feasibility

Any partial assignment (also referred to as a partial Boolean model) to variables in φ_B represents a conjunction of *difference logic* predicates. The Boolean model is represented as a weighted directed graph (*a.k.a* constraint graph) [32], where an edge $x \rightarrow y$ with weight c (denoted as (x, y, c)) corresponds to the predicate $e \equiv (x + c \geq y)$ where $\alpha(e) = T$. The constraint graph is said to be consistent if and only if it does not have an accumulated negative weighted cycle (or simply, negative cycle.) Intuitively, a negative cycle violates the transitivity property of the *difference logic* predicates. The building of the constraint graph and detection of negative cycles, as shown in Figure 1, are done incrementally to amortize the cost of constraint propagation. It has been shown [33] that addition of a predicate and update of a feasible assignment α can be done in $O(m+n \log n)$ where m is the number of predicates and n is the number of variables. After the constraint graph is detected consistent, i.e., feasible (shown by the feasible arc in Figure 1), more assignments

are made to the unassigned variables in φ_B leading to a more constraint graph. The problem is declared *satisfiable* by Boolean SAT, if there is no conflict and no further assignments to make.

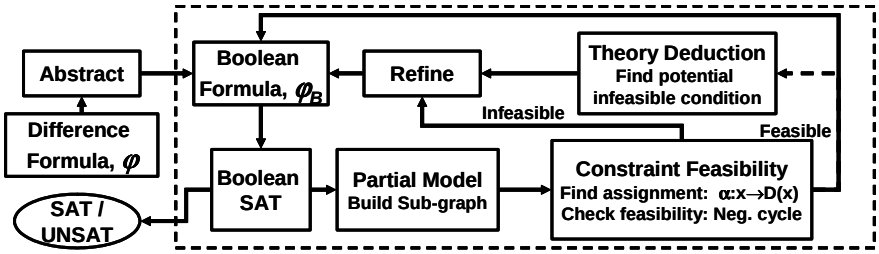

Figure 1. Overview of state-of-the-art *difference logic* solver based on lazy approach

2.1.3 Refinement

Whenever a negative cycle is encountered during constraint feasibility checking (*a.k.a.* constraint propagation), a transitivity constraint not yet implied by φ_B is learnt and added to φ_B as a conflicting clause. For example, if the subgraph corresponding to a conjunction of predicates, i.e., $e_1 \wedge e_2 \wedge e_3 \wedge \neg e_4$ has a negative cycle, then a clause $(\neg e_1 \vee \neg e_2 \vee \neg e_3 \vee\ e_4)$ is added to φ_B to avoid re-discovering it. As shown in [10], instead of stopping at the first negative cycle, one can detect all negative cycles and then choose a clause with minimum size representing a stronger constraint. Note, due to large overhead, addition of all detected negative cycle clauses is usually not done. Moreover, as in Boolean SAT solvers, incremental solvers [11, 24, 31] restore the assignments to the variables to a state just before the inconsistency was detected, instead of starting from scratch.

2.1.4 Preemptive Learning (Theory Deduction)

Some solvers [11, 23] have capabilities to add transitivity constraints preemptively to φ_B to avoid finding them later. However, as the overhead of adding all transitivity constraints can be prohibitive, as observed in a *per-constraint* eager approach [17], solvers often use heuristics to add them selectively and optionally (shown as dotted arrow in Figure 1).

2.2 Eager approach: Finite instantiation

Range allocation (*a.k.a. small domain encoding*) approaches find the adequate set of values (*a.k.a.* ranges) for each variable in the finite model. We briefly describe the range allocation problem for *difference logic* which has been discussed at greater depth in [21, 34]. Let *Vars*(φ) denote the set of variables used in a *difference logic* formula φ over the set of integers Z. We assume φ is in Non-Negated Form (NNF), i.e., every predicate occurring negatively in the formula is converted into its dual positive predicate a priori (e.g., $\neg(x + c < y) \Rightarrow (x + c \geq y)$) A domain (or range) $R(\varphi)$ of a formula φ is a function from *Vars*(φ) to 2^Z. Let *Vars*(φ) = $\{v_1,\ldots,v_n\}$ and $|R(v_i)|$ denote the number of elements in the set $R(v_i)$, the domain of v_i. The size of domain $R(\varphi)$, denoted by $|R(\varphi)|$ is given by $|R(\varphi)| = |R(v_1)| \cdot |R(v_2)| \cdots |R(v_n)|$. Let $SAT_R(\varphi)$ denote that φ is *satisfiable* in a domain

R. The goal is to find a small domain R such that

$$SAT_R(\varphi) \Leftrightarrow SAT_Z(\varphi) \qquad (1)$$

We say that a domain R is *adequate* for φ if it satisfies formula (1). Since finding the smallest domain for a given formula is at least as hard as checking the satisfiability of φ, the goal (1) is relaxed to finding the adequate domain for the set of all *difference logic* formulas with the *same set of predicates* as φ, denoted by $\Phi(\varphi)$. Adequacy for $\Phi(\varphi)$ implies adequacy for φ. As discussed in the previous section, *difference logic* predicates can be represented by a constraint directed graph $G(V, E)$. Thus, the set of all the subgraphs of G represents the set $\Phi(\varphi)$. Given G, the range allocation problem is set up to find a domain R such that every consistent subgraph of G can be satisfied from the values in R.

It has been shown [18] that for a *difference logic* formula with n variables, a range $[1 \ldots n+maxC]$ is adequate for each variable, with $maxC$ being equal to the sum of absolute constants in the formula. This leads to a state space of $(n+maxC)^n$ where all variables are given uniform ranges regardless of the formula structure. This small model encoding approach in *UCLID* [18], would require $\lceil log_2|R(x)| \rceil$ Boolean variables to encode the range $R(x)$, allocated for variable x. There has been further work [21] to reduce the overall ranges and hence, the size of the Boolean formula for the *difference logic*. A method *SMOD* was proposed [21] to allocate non-uniform ranges to variables, exploiting the problem structure. The method builds a cut-point SCC (Strongly Connected Component) graph recursively in a top-down manner and allocates ranges to the nodes bottom-up, propagating the range values. The approach is based on enumeration of all cycles and therefore, the worst-case complexity of such an approach is exponential. In a similar approach [35], ranges are obtained by not converting the dis-equalities into disjunctions of inequalities.

In this article, we discuss an efficient and robust method called *Nu-SMOD* [30], that computes non-uniform ranges in time polynomial in the number of predicate variables and size of the constants. Moreover, the ranges are comparable to, or better than, the non-uniform ranges obtained using *SMOD*, and consistently better than the uniform ranges obtained using *UCLID* [18]. We do not eagerly convert the difference logic problem into a propositional problem using these ranges. Instead, we use these ranges during theory consistency check lazily to reduce search. We specifically emphasize performance improvement, i.e., obtaining the ranges with smaller time overhead, as opposed to obtaining tight ranges. In experimental evaluation, *Nu-SMOD* completes range allocation for all the benchmarks unlike *SMOD*, with 1-2 orders of magnitude performance improvement over *SMOD*. Unlike *SMOD*, we do not compute cut-point SCCs or enumerate cycles in our new procedure *Nu-SMOD*; rather we propagate only distinct values along a path from a cut-point. Thus, our objective differs from the *SMOD* procedure and the work using finite-instantiations [35].

3. SDSAT: Integrating Small Domain and Lazy Approaches

We propose a *difference logic* Solver *SDSAT* as shown in Figure 2, that combines the strengths of both eager (small domain encoding) and lazy approaches and provides a robust performance over a wide set of benchmarks. This combined approach benefits both from the reduced search space (as in eager approaches) and also from the need-to basis refinement

of the Boolean formula with transitivity constraints (as in lazy approaches). The solver *SDSAT* proceeds in two phases: *allocation* and *solve*.

In the *allocation* phase (shown as *Phase I* in Figure 2), it computes *non-uniform adequate ranges* using an efficient technique *Nu-SMOD* that runs in polynomial time; polynomial in the number of predicate variables and size of the constants. This phase is similar to previous small domain encoding approaches. However, we do not transform the *difference logic* formula into an equi-satisfiable Boolean formula in a single step, but rather transform it lazily in the following phase.

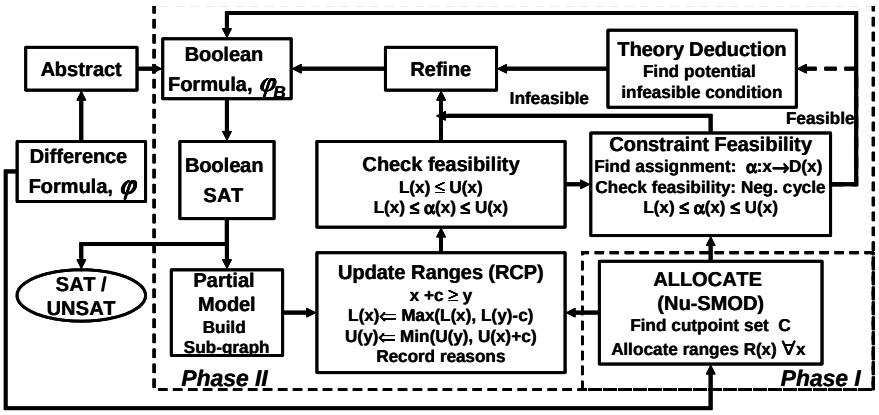

Figure 2. Overview of our *difference logic* solver *SDSAT*

In the *solve phase* (shown as *Phase II* in Figure 2), *SDSAT* searches for a satisfying model *within the allocated ranges* using a lazy refinement approach. Thus, any partially DL-theory consistent model is discarded if it cannot be satisfied within the allocated ranges (The check is done in the blocks "Check feasibility" and "Constraint feasibility" in Figure 2.) Note the key difference: *in eager approaches, such a partially consistent model is not allowed in the first place, while in lazy approaches such a model is never discarded.* By focusing on adequate ranges and not just consistency of the *difference logic* predicates, we are able to learn more constraints leading to larger reductions in search space. Furthermore, we dynamically refine the ranges allocated to variables in the *allocation phase* using range constraint propagation (described in Section 3.2.2) and search for a feasible solution within the updated ranges (shown in the block "Updated Ranges (RCP)" in Figure 2). Another novelty is in the use of cutpoints to determine whether an added edge (to a consistent model) leads to an infeasible condition. This is based on the observation that any cycle will have at least one cutpoint. (Given a directed graph $G(V, E)$, a *cutpoint* set $C \subseteq V$ is a set of nodes whose removal breaks all the cycles in G.) If an added edge $x \rightarrow y$ (corresponding to the predicate $x+c \geq y$) is not reachable from some cutpoint, and x is not a cutpoint, then a previously consistent subgraph modified with this new edge is guaranteed *not to* have a negative cycle. Moreover, like in most lazy approaches, *SDSAT* has incremental propagation and cycle detection, and preemptive learning of infeasible condition (theory deduction, shown as dotted arrow in Figure 2).

3.1 Allocation Phase: Non-Uniform Range Allocation

We discuss the algorithm *Nu-SMOD* that we use to allocate non-uniform ranges to variables in the predicates. The algorithm assumes that the constraint directed graph $G(V, E)$ is an SCC. Extension to non-SCCs is straightforward: compute the ranges for SCCs individually and then offset the ranges appropriately to account for the edges between the SCCs starting from some root SCC. As far as validity of the *difference logic* problem is concerned, it is easy to see that these edges can be removed from the problem as they will never contribute to a cycle.

Algorithm *Nu-SMOD*: We describe the procedure *Nu-SMOD* as shown in Figure 3. We first derive a cutpoint set C using polynomial approximation [36], as finding a minimal cutpoint set is an NP-Hard problem. Using the cutpoint set C as initial set of nodes I, we invoke the procedure *Nu-SMOD-1*, as shown in Figure 4, to allocate the ranges as follows: *forward range* of each node x, denoted by $R_f(x)$, is divided into several sets; each identified with a unique id or simply *level*. Let the level k set of the node x be denoted by $L^k(x)$. Note, $R_f(x)$=$\cup_k L^k(x)$. Initially, all the level sets are empty. The nodes in Level 1 set, denoted by I, are allocated 0 value, i.e., $\forall_{x \in I} L^1(x) = \{0\}$. To compute a Level $(k+1)$ value — i.e., $L^{k+1}(y)$ for node y (line 7), we offset the Level k value of an incoming node x with an edge weight c, where the edge corresponds to the predicate $x+c \geq y$. Thus, to compute all Level $(k+1)$ values, we offset each Level k value for every incoming edge to y (lines 5-7). We refer allocation of such level values also as *tight value allocation*.

After obtaining the ranges for the cutpoints C using *Nu-SMOD-1*, we obtain *reverse_dfs* values $Q[y]$, for each non-cutpoint y (lines 3-6, Figure 3). Starting from each cutpoint (line 4-5) with value M (equal to maximum range value allocated among the cutpoints), we call the procedure *reverse_dfs* (lines 8-12) to update Q values (line 10) of all the non-cutpoints, by reverse propagating a tight value (higher than the previous Q value, line 9) without traversing through any other cutpoints (line 8). Note that the reverse DFS path from a cutpoint to non-cutpoint is a simple path as there is no cycle. All the inequalities from non-cutpoint to cutpoint are satisfied using *reverse_dfs* Q values. Range of the node x, $R(x)$ is given by (line 7), $R_f(x) \cup \{Q[x]\}$.

Overall the runtime complexity of *Nu-SMOD* can be shown to be polynomial in the number of nodes n and edges m and size of the maximum edge constant (see Appendix A for details).

Example 1: We illustrate *Nu-SMOD* algorithm on an example shown in Figure 5. Let the *difference logic* formula F be $e_1 \wedge e_4 \wedge e_5 \wedge e_8 \wedge e_9 \wedge (e_2 \vee e_3) \wedge (e_6 \vee e_7)$ where e_i represents a *difference* predicate. Let $n_0 \ldots n_5$ represent the integer variables. The *difference* predicates are shown as edges e_i in Figure 5(a) (with weights in brackets). For example: $e_1 \equiv (n_0 \geq n_1)$ and $e_9 \equiv (n_5 - 1 \geq n_0)$.

In the first step, we derive the cutpoint set $\{n_2\}$ for the constraint graph $\Phi(F)$ as shown in Figure 5(a). Using the procedure *Nu-SMOD-1* with I=$\{n_2\}$, we derive the Level values $L^k(x)$ at depth k starting from nodes in the set I by doing forward traversal as shown in Figure 5(b). Note, n_2 has direct edges e_5 and e_7 to nodes n_4 and n_6, respectively. Using *tight value allocation*, i.e., $n_4 = n_2$ and $n_6 = n_2$, we obtain $L^2(n_4)$=$\{0\}$ and $L^2(n_6)$=$\{0\}$, respectively. Similarly, we obtain the level values for the other nodes as well. Now, we compute *reverse_dfs* Q values of all non-cutpoints starting from the cutpoint set $\{n_2\}$ with

Synopsis: Range allocation for an SCC G(V,E)
Input: G(V,E), Cutpoint set C
Output: R(x) for $\forall$x $\in$ V
Procedure: Nu-SMOD

1. *Nu-SMOD-1*(G, I); {**Input**: $G(V, E)$, $I = C$; **Output:** $\forall_{x \in V} R(x)$ }
2. $\forall_{y \in V \setminus C}\ Q[y] = -\infty$;
3. M=*max* $(\cup_{\forall x \in C} R_f(x))$;
4. foreach $x \in C$ do
5. *reverse_dfs*(x, M);
6. end
7. $\forall_{x \in V}\ R(x) = R_f(x) \cup \{Q[x]\}$;

Sub-procedure: reverse_dfs (x, v)

8. foreach (y, x, w) s.t. $y \notin C$ do
9. if $(Q[y] + w \geq v)$ continue;
10. $Q[y] = v - w$;
11. *reverse_dfs* $(y, Q[y])$;
12. end

Figure 3. Pseudo-code for the algorithm *Nu-SMOD*

Synopsis: Level values allocation
Input: $G(V, E)$, $I \subseteq V$
Output: $\forall_{x \in V} R_f(x)$
Procedure: Nu-SMOD-1

1. $\forall_{x \in I} L^1(x) = \{0\}$, $\forall_{x \in V \setminus I} L^1(x) = \{\}$
2. $\forall_{x \in V, 1 < k \leq |V|} L^k(x) = \{\}$,
3. foreach k, $1 \leq k < |V|$ do
4. foreach *node* $x \in V$ do
5. foreach $(x, y, c) \in$ *fanouts*(x) do
6. foreach *value* $v \in L^k(x)$ do
7. $L^{k+1}(y) = L^{k+1}(y) \cup \{v + c\}$;
8. $\forall_{x \in V} R_f(x) = \cup_{1 \leq k \leq |V|} L^k(x)$;

Figure 4. Pseudo-code for the algorithm *Nu-SMOD*-1

value M (=0), *equal to* the maximum of all level values as shown in Figure 5(c). The allocated range R for each node is the union of the level values of all levels (i.e., R_f) and *reverse_dfs* values (Q) as shown in Figure 5(d). In the following Theorem 1, we show that the ranges R so allocated are *adequate*.

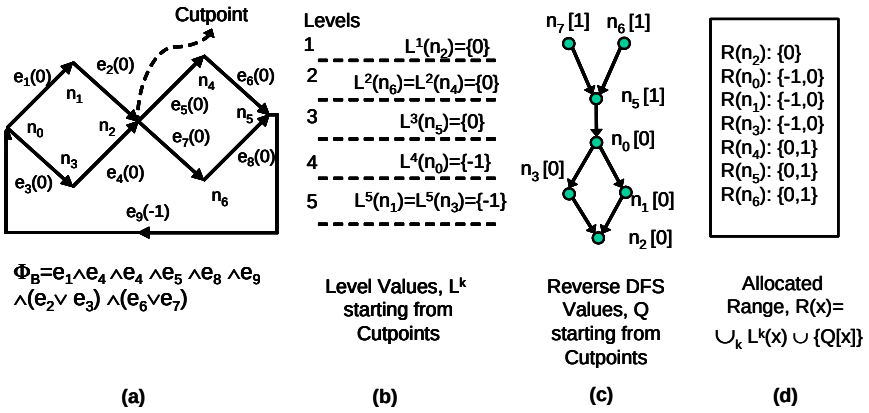

Figure 5. Procedure *Nu-SMOD* on an example

Theorem 1. *Ranges allocated by Nu-SMOD are adequate.*

Proof: We now show that the ranges allocated by *Nu-SMOD* are *adequate*, i.e., any satisfiable sub-graph $G^d(V^d, E^d)$ of $G(V, E)$ ($V^d \subseteq V$, $E^d \subseteq E$) has a satisfying assignment from the allocated set of ranges. We further assume G^d is connected. If not, then each component is a satisfiable sub-graph of G and ranges can be assigned to variables in each component independently of the other. We construct the adequacy proof by devising an assignment procedure *ASSIGN* as shown in Figure 6, which will generate a satisfying solution from the allocated set of ranges.

We first construct a set S of *root* nodes (those nodes in $V^d \cap$ C that cannot be reached from any other node in $V^d \cap$ C) in G^d (line 1). If set S is empty, either $V^d \cap C$ is empty or all nodes are in some cycle. In the former case, we skip to line 7, else we pick any node in $V^d \cap C$ and continue. We initially assign all the nodes not in S with $+\infty$ (a large positive value, line 2). We denote the value assigned to a node x as v_x. Starting from each node in S (with initial value 0 as in line 4), we call *bfm* (similar to Bellman-Ford-Moore Shortest Path algorithm [27]) procedure to assign *tight* values on the nodes that can be reached. The edge (x, y, c) is said to be *stable* if the current value of x and y is said to satisfy the constraint $(x + c \geq y)$. Note that the value of the node can change only if the current value is lower than the previously assigned value (line 11). Such an operation is also called an *edge relaxation* [27]. Only under such a scenario, the node is en-queued (line 12). Those nodes whose value are still $+\infty$, are given *reverse_dfs* Q values (line 7). To show that the given assignment procedure *ASSIGN* generates a satisfying solution from the ranges allocated, we need to prove the following lemmas. (Proof details are in Appendix B.)

Lemma 1. *The procedure ASSIGN terminates.*

Lemma 2. *All inequalities corresponding to edges of G^d are satisfied.*

Lemma 3. *Each assigned value v_x belongs to $R(x)$.*

The above theorem guarantees the existence of a solution for a satisfying subgraph G^d with all the *root nodes* in $V^d \cap$ C having special value 0 and the other nodes in $V^d \backslash$C

Synopsis: Assignment for subgraph G^d of G
Input: $G^d(V^d, E^d)$
Output: $\{(x, v_x) | x \in V^d, v_x \in R(x)\}$
Procedure: ASSIGN

1. S = {set of root nodes}
2. $\forall_{y \in V^d \setminus S}\ v_y = +\infty$;
3. foreach $x \in S$ do
4. $\quad v_x = 0$; *enqueue*(x);
5. $\quad$ *bfm*(x);
6. end
7. $\forall_{y \in V^d \setminus S}$ if $(v_y = +\infty)\ v_y = Q[y]$;

Sub-procedure: bfm(x)

8. while $(x =$ *dequeue*$()) \neq$ null)
9. $\quad$ foreach$(x, y, c) \in$ *fanouts*(x) do
10. $\quad\quad$ if $(v_x + c \geq v_y)$ continue;
11. $\quad\quad$ $v_y = v_x + c$;
12. $\quad\quad$ *enqueue*(y);
13. $\quad$ end
14. end

Figure 6. Pseudo-code for the algorithm *ASSIGN*

having either *tight* values or *reverse_dfs* values Q, depending on whether they are reachable from the root nodes or not, respectively. Note that the cutpoints do not need Q values as they are the root nodes. As we will see shortly, the *solve phase* is based primarily on this observation.

Example 1 (contd.): We illustrate a line-by-line run of the procedure *ASSIGN* on a subgraph G^d with $V^d = \{n_1, n_3, n_2, n_4, n_6, n_5\}$ and $E^d = \{e_2, e_4, e_5, e_7, e_6, e_8\}$. Note, $S = \{n_2\}$ in Line 1. In Line 4, $v_{n_2} = 0$. On execution of Line 5, we obtain $v_{n_4} = 0$, $v_{n_6} = 0$, and $v_{n_5} = 0$, and on execution of Line 7, we obtain $v_{n_1} = 0$ and $v_{n_3} = 0$ (using the *reverse_dfs* values as shown in Figure 5(c)).

3.2 Solve Phase

Similar to standard lazy solvers, we first build an abstract Boolean formula φ_B from the given *difference logic* formula φ and search for a partial consistent Boolean model. As the partial model is being incrementally built, we search for a satisfying model using a *cutpoint-relaxation algorithm* (described in Section 3.2.1) within the dynamically updated ranges achieved by *range constraint propagation* (described in Section 3.2.2). We build these algorithms by augmenting the procedure *ASSIGN (*described in Figure 6) with

- inconsistency detection due to negative cycles,

- range violations check, and
- pre-emptive learning.

In the following, we restrict our discussion to novelties in detecting the inconsistencies. (For details on pre-emptive learning please refer to [11, 23, 37]).

3.2.1 Incremental Cycle Detection Using Cutpoint Relaxation

In the past [31, 38, 33], the detection of negative cycles and finding satisfying assignments are done incrementally in a weighted digraph that is built incrementally. Each of these algorithms uses a variant (mostly in the ordering of the relaxed edges) of Bellman-Ford-Moore Shortest Path (BFMSP) algorithm and extends it with an ability to detect negative cycle. Our approach is also based on BFMSP with the following difference: *For a satisfiable sub-graph G^d, we consider only those solutions which lie within the ranges allocated by the Nu-SMOD procedure.* Note, a satisfying assignment set $\{\alpha(x)\}$ represents a class of satisfying assignments $\{\alpha(x)+k\}$ for some constant k.

As shown in the procedure *ASSIGN*, the existence of the solution for a satisfying sub-graph G^d is guaranteed with all the *root nodes* in $V^d \cap$ C having special value 0 and the other nodes in $V^d \backslash$C having either *tight* values or *reverse_dfs* values Q, depending on whether they are reachable from root nodes or not, respectively. Thus, in our approach, we restrict the set of satisfying assignments such that $\alpha(x)=0$ for the *root nodes* $x \in V^d \cap C$. We discuss the implication of such a restriction in our incremental cycle detection algorithm *cutpoint relaxation*. As will be clear shortly, the theoretical complexity of the algorithm is not different from BFMSP and its variants. In our *cutpoint relaxation* algorithm (unlike *ASSIGN* procedure) we do not change $\alpha(x)$ from $+\infty$ to $Q[x]$ if a node x is not reachable from a root node (due to incremental addition of edges, such a node may be reachable later). Now, we discuss how the incremental addition and deletion of edges affect the negative cycle detection.

Edge Addition: Suppose, we add an edge (x, y, c) to G^d and obtain a subgraph $G^{d'}$. If $\alpha(x) \neq +\infty$, x is reachable from some root node in G^d and we do the usual BFMSP. If $\alpha(x) = +\infty$, we consider two cases depending on $x \in C$ or $x \notin C$.

- *Case* $x \in C$: Clearly, x is root node in $G^{d'}$ as it is not reachable from any other root node in G^d. We choose $\alpha(x)=0$ and do the usual BFMSP with negative cycle detection after relaxing (x, y, c).
- *Case* $x \notin C$: Note, x is not reachable from any node in $V^d \cap C$. As any cycle will have at least one cutpoint and since x is not a cutpoint in G, there cannot be any cycle in subgraph $G^{d'}$ (of G) with the edge (x, y, c). Based on this observation, we skip edge relaxation and cycle detection for this case.

Edge Deletion: When an edge (x, y, c) is deleted, we need to restore the previous $\alpha(y)$ value only if it is different from $+\infty$. Since, deletion of edges takes place at the time of backtracking, we restore only those $\alpha(y)$ that got affected after the backtrack level. We use a standard stack-based approach for efficient backtracking.

Thus, our algorithm *cutpoint relaxation* has two main novelties: First, the approach allows us to identify cases where we guarantee no negative cycles in a subgraph *without* edge relaxation. Second, we reduce the search space by restricting our solution space in a spirit similar to finite instantiation. Though maintaining such a restriction on assignment values on root nodes has an overhead, we did not find it to be a significant bottleneck. Besides using cutpoints and restricted solutions to reduce the search space, we can further reduce the search space by dynamically updating the ranges of the variables as discussed in the following section.

3.2.2 Range Constraint Propagation (RCP)

Ranges computed by the *allocation phase* guarantee the adequacy for a satisfiable subgraph G^d; however, the ranges are often more than those required to obtain a satisfying solution for G^d. We allow range constraint propagation (RCP) to dynamically refine the ranges of the variables for the given subgraph G^d, while maintaining the range adequacy (Theorem 2). This approach is similar to the more general approach for interval arithmetic [39, 40], and arc-consistency used in the constraint programming community [41]. We achieve RCP as follows: Let the minimum (MIN) and maximum (MAX) values in the range of a variable x be denoted by $L(x)$ and $U(x)$, respectively. Initially, these limits are obtained during the *allocation phase*. RCP on an edge $x+c \geq y$, denoted by RCP($x+c \geq y$), updates the limits $L(x)$ and $U(y)$ as follows:

$$\begin{aligned} L(x) &\Leftarrow \text{MAX}\{L(x),\, L(y) - c\} \\ U(y) &\Leftarrow \text{MIN}\{U(y),\, U(x) + c\} \end{aligned} \qquad (2)$$

We apply this process recursively, i.e., whenever the L (or U) value of a given node changes, we update the L (or U) values of all nodes with a direct edge to (or from) the given node. The process stops when either a range violation is detected, i.e., $L(x) > U(x)$ or all the limits have stabilized. As constraint propagation reduces the range sizes monotonically, the process is guaranteed to terminate. A conflict can also be detected due to range violation of the invariant $L(x) \leq \alpha(x) \leq U(x)$ where $\alpha(x)$ is a satisfying assignment for x reachable from some *root node*. Note, these range violations can occur in a subgraph *even without a negative cycle*. (These checks are carried out in the block "Check feasibility" in Figure 2. We illustrate this with an example later.) *Thus, the reduced range space leads to faster detection of conflicts and hence, reduced search.* We can also obtain the set of conflicting edges by storing the edges as reasons for the change in minimum and maximum limits. The following theorem addresses the range adequacy after RCP.

Theorem 2. *Reduced ranges obtained by RCP are adequate for subgraph G^d.*

Proof: See Appendix C.

Example 1 (contd): We illustrate RCP and its role in reducing the search space on the diamond example (shown in Figure 5(a)). The previous approaches based on *only* negative cycle detection have to find all four negative cycles involving edge pairs (e_2, e_6), (e_2, e_7), (e_3, e_6) and (e_3, e_7), before the *difference logic* formula F is declared unsatisfiable. Using

our approach of combined negative cycle detection with RCP, we decide unsatisfiability with detection of two negative cycles and one range violation as described below.

As shown in Figure 5(d), L and U of each variable are initially set to corresponding minimum and maximum range values R obtained by *Nu-SMOD* . For example: $L(n_0) = -1$, $U(n_0) = 0$, as discussed in Section 3.1. Note that these ranges are adequate for this graph. Consider the subgraph $e_1 \wedge e_2 \wedge \neg e_3$. Assume the edges are added in the order e_1, e_2 and $\neg e_3$. The step-wise execution of RCP on these edges is shown in Figure 7(a), with ${}^*L$ and ${}^*U$ denoting changes from the previous step. In Step 1 when e_1 is added with $(L(n_0) = -1 = L(n_1)$ and $U(n_0) = 0 = U(n_1))$, $L(n_0)$ and $U(n_1)$ are unchanged (Eq. 2). In Step 2, when e_2 is added, $L(n_0)$ updates to 0 as $L(n_2) = 0$; which in turn updates $L(n_1)$ to 0. Similarly, in Step 3, with the addition of edge $\neg e_3$, $L(n_3)$ updates to 0 and $U(n_0)$ updates to -1. The latter update causes $U(n_1)$ and $U(n_2)$ to change in Step 4 and 5, respectively. At Step 5, we detect a range violation as explained in the following . As $U(n_0)$ changes in Step 3, we change $U(n_1)$ to -1 in Step 4 as the edge e_1 is incident on n_1, and $U(n_2)$ to -1 in Step 5 as the edge e_2 is incident on n_2. Now, as $L(n_2) = 0 > U(n_2) = -1$, we detect a range violation. We also learn a clause $(\neg e_1 \vee \neg e_2 \vee e_3)$ by performing conflict analysis.

Using the learnt clause by RCP, together with two other conflict clauses due to negative cycle detection, we show how the formula F can be declared unsatisfied by simply applying resolution rules, as shown in Figure 7(b). The learnt clause $(\neg e_1 \vee \neg e_2 \vee e_3)$, together with the formula clause $(e_2 \vee e_3)$ implies a clause $(\neg e_1 \vee e_3)$; which in turn with formula clause (e_1) implies (e_3). When we detect two negative cycles with edge pairs (e_3, e_7) and (e_3, e_6), we learn that e_3 *implies* $(\neg e_6 \wedge \neg e_7)$. As $(e_6 \vee e_7)$ is a formula clause, we could declare the formula F unsatisfiable, without the need to detect further negative cycles.

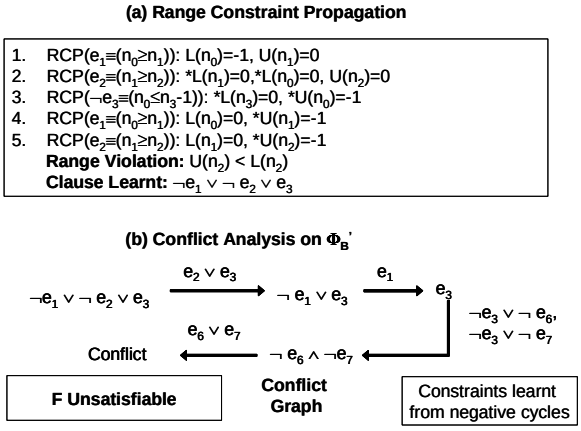

Figure 7. RCP with negative cycle detection

4. Experimental Results

We have integrated our incremental cycle detection using cutpoint relaxation and RCP with the zChaff Boolean SAT solver [42]. We have also implemented pre-emptive learning but

have not done controlled experiments to ascertain its usefulness. We conducted experiments on a set of six public benchmark suites generated from verification and scheduling problems: *diamonds, DTP, DLSAT, mathsat, sal* and *uclid* [43]. We ran our experiments on a workstation with 3.0 GHz Intel Pentium 4 processor and 2 GB of RAM running Red Hat Linux 7.2. First, we compare the range allocation algorithms; second, we evaluate the effectiveness of RCP in *SDSAT* and third, we compare it with state-of-the-art solvers (available at the time of experimentation).

4.1 Comparison of Range Allocations Algorithms

We compared our approach *Nu-SMOD* with previous approaches *SMOD* [21] and *UCLID* [18] on these benchmarks and present results in Figures 8 and 9. We used a time limit of 2 minutes for each run. Note, the *UCLID* procedure allocates to each of n nodes in an SCC a continuous range from 1 to $n+maxC$, where $maxC$ is the sum of all constant absolute values. We compare the number of Boolean variables required to encode the ranges assigned by the different approaches as the *ratio between the approach and Nu-SMOD.* Note, for range set $R(y)$, we require $\lceil log_2|R(y)|\rceil$ Boolean variables to encode the set $R(y)$.

4.1.1 UCLID v/s Nu-SMOD

Nu-SMOD, when compared to *UCLID* (Figure 8), allocates on average about 40% less range bits (about 4X less on *diamond set*). Note that such linear reductions amount to exponential reduction in search space.

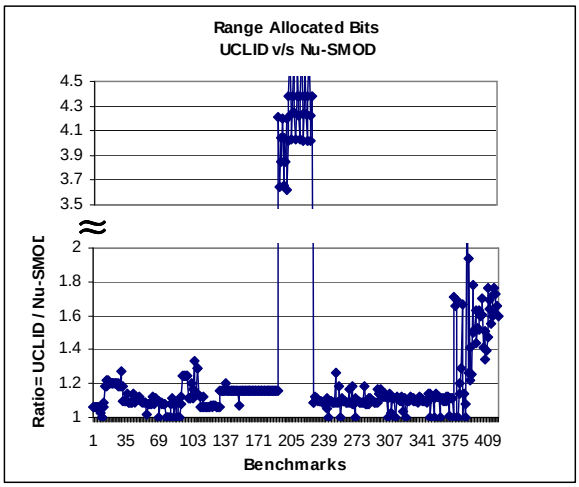

Figure 8. Ratio of range bits allocated between UCLID v/s Nu-SMOD

4.1.2 SMOD v/s Nu-SMOD

Of 432 benchmarks, *SMOD* could complete only 262 in the given time limit of 2 minutes. If we increase the time limit to 20 minutes, it solves 23 more cases. Not surprisingly, time-out occurs mostly for dense graph as also observed by the authors [21]. Barring a

few benchmarks, the ranges allocated by *Nu-SMOD* are comparable to *SMOD* as seen in Figure 9(a). Moreover, *SMOD* is 1-2 orders of magnitude slower on the completed benchmarks as compared to *Nu-SMOD*, as shown in the scatter plot (on logarithmic scale) in Figure 9(b).

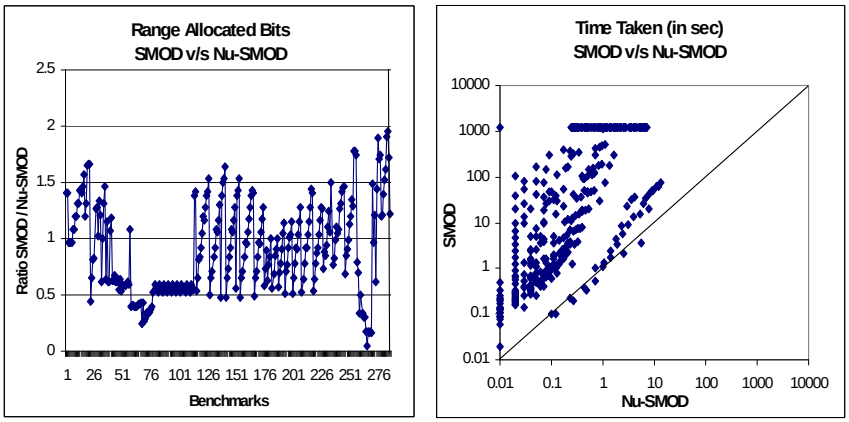

Figure 9. Ratio of range bits allocated between (a) SMOD v/s Nu-SMOD. (b) Scatter plot of time taken (in sec) between SMOD v/s Nu-SMOD

4.2 Allocation and Role of RCP in SDSAT

In the second set of experiments, we present the results of *allocation phase* and compare the effectiveness of refinement in *SDSAT* with and without RCP as shown in Table 1. In our experience, the number of refinements did not distinguish the role of RCP. We observed performance improvement using RCP with more refinements as well as with fewer refinements. Thus, instead of using the number of refinements, we introduce two metrics to measure its effectiveness: *refinement overhead* and *refinement penalty*.

We define *refinement overhead* as the time taken in the corresponding graph algorithm per refinement, and *refinement penalty* as the time taken by Boolean SAT per refinement. The former metric measures the cost in detecting the inconsistency, whereas the latter measures the cost of Boolean search after refinement, evaluating its effectiveness. Ideally, we would like to have a low number for both the metrics.

In Table 1, Column 1 shows the benchmark suites with the number in brackets indicating the number of problems considered. Columns 2-3 show the results of *allocation phase*: Column 2 shows the average size of range bits per variable computed, and Column 3 shows the average time taken. Columns 4-5 show the results of incremental negative cycle detection without RCP: Column 4 shows the average refinement overhead (in milliseconds), and Column 5 gives the average refinement penalty (in milliseconds). Similarly, Columns 6-8 show the result of incremental negative cycle detection with RCP: Column 6 shows the average refinement overhead (in milliseconds), Column 7 shows the average refinement penalty (in milliseconds), and Column 8 shows the average percentage of refinements due to RCP.

Note first that the time overhead in the *allocation phase* is not very significant. The bits allocated for the ranges average around 10 bits per variable. Though the solution space is reduced, the bit blasted translation of the formula could be quite large if we were to apply a small domain encoding [18]. Note that in the presence of RCP, the refinement overhead is not affected significantly. Moreover, a lower refinement penalty with RCP indicates improvement in the quality of refinements and Boolean search. We also observe that, except for diamonds, on average 50% refinements are due to range violations discovered during RCP.

Table 1. SDSAT: Allocation and role of RCP

Bench	Allocation		-ve cycle w/o RCP		-ve cycle with RCP		
	Avg. Range bits per var	Avg.Time taken (s)	Ref ovhd (ms)	Ref pnlty (ms)	Ref ovhd (ms)	Ref Pnlty (ms)	Range viol.(%)
DTP (59)	13	0.46	0.2	0.3	0.2	0.18	48
diamonds(36)	0.99	0.14	0.1	0.12	0.006	0.02	100
mathsat (147)	9.97	0.94	32	713	32	371	48
DLSAT (31)	11.9	3	0.2	1.6	0.3	0.9	45
sal (99)	10.9	3.34	1	36	1	19	49

4.3 Comparison with other Difference Logic Solvers

In the third set of experiments, we compare our approach *SDSAT* (the *solve phase*) with other available state-of-the-art tools (at the time of experimentation), including *UCLID* [19], *MathSAT (version 3.2.1, release 2005)* [11], *ICS* [4], *TSAT++ (version 0.5, release 2004* [10], and *Barcelogic (release 2005)* [23].

Since *allocation phase* has a constant time overhead, we use the *solver phase* run-time for comparison to understand the results better. We used a common platform and 1 hour time limit for each of the benchmarks. We present the cumulative results in Table 2. Due to unavailability of appropriate translators, we could not compare on *Uclid* benchmarks for this experiment. Pairs of the form (n t) represent that the particular approach timed out in n number of cases for that benchmark suite. Overall, we observe that *SDSAT* and *Barcelogic* have better performance compared to other lazy and eager approaches by several orders of magnitude. Comparing *SDSAT* with *Barcelogic*, we see an improvement in some suites, in particular, *diamonds* and *mathsat.* Especially for *diamonds*, *SDSAT* is able to detect unsatisfiability in less than 1 sec for 32 out of 36 problems. Though there are many negative cycles in these *diamonds* problems, RCP is able to take advantage of the significantly reduced ranges as shown in Column 2 in Table 1. On the whole, *SDSAT* times out in 7 cases as compared to 10 cases for *Barcelogic.* Thus, overall our approach is relatively more robust than the pure lazy approaches which can also benefit using our ideas.

Comment: We are aware of the newer version of the solvers such as `yices-1.0` [13], `MathSAT-3.4` [11] and `Barcelogic 1.1` [23] that were developed after our experimentation. These versions have improved data structures with incremental solving capabilities to perform even better than we report. We believe that our approach is orthogonal to these methods, and can be combined with them to further improve our results. However, due to

Table 2. Performance comparison (in sec) of state-of-the-art *difference logic* solvers

Bench	TSAT++	UCLID	MathSAT	ICS	Barcelogic	SDSAT
DTP (59)	642	122590 (34 t)	120	188592 (48 t)	10	202
diamonds (36)	6571	32489 (9 t)	24302 (1 t)	51783 (11 t)	679	41
mathsat (147)	62863 (15 t)	73751 (20 t)	41673 (9 t)	51789 (13 t)	37696 (8 t)	31279 (6 t)
DLSAT (31)	276	97334 (27 t)	429	12671 (2 t)	13	46
sal (99)	135909 (34 t)	156399 (43 t)	57401 (15 t)	107313 (28 t)	18721 (2 t)	22178 (1 t)

the unavailability of the source codes of these solvers and practical difficulty in reproducing their results, we did not integrate our approach. The results we report here are the same as that appeared in [30].

5. Conclusions

We proposed a novel *difference logic* solver *SDSAT* that takes advantage of the small domain property of *difference logic* to perform a lazy search of the state space. The solver tightly integrates the strengths of both lazy and eager approaches and provides robust performance over a wide range of benchmarks. It first allocates *non-uniform adequate ranges* efficiently and then uses the graph-based algorithms to search *lazily* for a satisfying model within the allocated ranges. It combines a state-of-the-art negative cycle detection algorithm with range constraint propagation to prune out infeasible search space very efficiently. Moreover, it also benefits from incremental propagation and cycle detection using a *cutpoint-relaxation* algorithm. Experimental evidence presented here bears out the efficacy of our ideas, which can be combined with other more recent improvements.

Acknowledgments

We sincerely thank the anonymous reviewers for their suggestions and comments in improving the quality of the paper.

Appendix A: Runtime Analysis of *Nu-SMOD*

The runtime of our algorithm *Nu-SMOD* depends on the size of the constants, i.e., the edge weights present in the graph. In the following, we denote n $(=|V|)$ to be the number of nodes and m $(=|E|)$ to be the number of edges in the graph $G(V, E)$. The worst case runtime of the basic algorithm is $O(m^n)$. This is because of the following reason: At level 1, $|L^1(x)| \leq 1$ for $\forall x \in V$. Since size of fanins of x is $O(m)$,

$$|L^k(x)| \quad \leq \quad \Sigma_{f \in fanin} \, |L^{k-1}(f)| \quad \leq m \, * \, max_{f \in fanin} \, |L^{k-1}(f)|.$$

Thus, $|L^k(x)| \leq m^{k-1}$. Since there are n levels, we can propagate $\Sigma_k\ |L^k(x)| \sim m^{n-1}$ values for x and m^n in total. So, the worst case running time is $O(m^n)$. In practice, the worst case running time is generally not nseen. The reasons are as follows: First, if the same value is propagated to a node from multiple fanin edges, it won't be propagated further. If the edge weights are bounded, we see more of such overlapping. Second, for not very dense graphs, number of fanin edges will be much smaller than the worst case bound, i.e., $O(m)$.

Here we derive a more reasonable bound on running time and ranges by assuming an upper bound N on the edge weights. We know $\forall$x, $L^1(x) \subseteq \{0\}$ and $|L^1(x)| \leq 1$. Since, N is an upper bound on the absolute values of the edge weights, $\forall x\ L^2(x)$ can have at most $2N+1$ values (i.e., *-N to N*). Similarly $\forall x\ L^3(x)$ can have at most $4N+1$ values (i.e., *-2N to 2N*). In general, $\forall x\ L^{k+1}(x)$ can have at most *2kN+1* values (i.e., *-kN to kN*). Since highest level is n, the $L^n(x)$ is at most *2(n-1)N+1*. If we do not propagate the same value again, for each x, we propagate at most *2(n-1)N+1,* i.e, *O(nN)* values and in total we propagate $O(n^2N)$ values. Thus, runtime complexity of *Nu-SMOD-1* (Figure 4) is polynomial in the number of nodes n and size of the maximum constant. This also holds for *Nu-SMOD* if we consider all nodes as cutpoints. Note, every non-cutpoint y gets one *reverse_dfs* value $Q[y]$. As the reverse path from a cutpoint to a non-cutpoint is a simple path (i.e., no cycle), the complexity of *reverse_dfs* is *O(nm)* by keeping a queue similar to *bfm.* Thus, overall the runtime complexity of *Nu-SMOD* is polynomial in the number of nodes n and edges m, and size of the maximum constant, N.

Appendix B: Theorem 1

Theorem 1. *Ranges allocated by Nu-SMOD are adequate.*

Proof: We now show that the ranges allocated by *Nu-SMOD* are *adequate*, i.e., any satisfiable sub-graph $G^d(V^d, E^d)$ of $G(V, E)$ ($V^d \subseteq V$, $E^d \subseteq E$) has a satisfying assignment from the allocated set of ranges. We further assume G^d is connected. If not, then each component is a satisfiable sub-graph of G and ranges can be assigned to variables in each component independently of the other. We construct the adequacy proof by devising an assignment procedure *ASSIGN* as shown in Figure 6, which will generate a satisfying solution from the allocated set of ranges.

We first construct a set S of *root* nodes (those nodes in $V^d \cap$ C that cannot be reached from any other node in $V^d \cap$ C) in G^d (line 1). If set S is empty, either $V^d \cap C$ is empty or all nodes are in some cycle. In the former case, we skip to line 7, else we pick any node in $V^d \cap C$ and continue. We initially assign all the nodes not in S to $+\infty$ (a large positive value, line 2). We denote the value assigned to a node x as v_x. Starting from each node in S (with initial value 0 as in line 4), we call *bfm* (similar to Bellman-Ford-Moore Shortest Path algorithm [27]) procedure to assign *tight* values on the nodes that can be reached. The edge (x, y, c) is said to be stable if the current value of x and y is said to satisfy the constraint $(x + c \geq y)$. Note that the value of the node can change only if the current value is lower than the previously assigned value (line 11). Such an operation is also called an *edge relaxation* [27]. Only under such a scenario, the node is en-queued (line 12). Those nodes whose value are still $+\infty$, are given *reverse_dfs* Q values (line 7). To show that the given assignment procedure *ASSIGN* generates a satisfying solution from the ranges allocated, we need to prove the following lemmas.

Lemma 1. *The procedure ASSIGN terminates.*

Lemma 2. *All inequalities corresponding to edges of D are satisfied.*

Lemma 3. *Each assigned value v_x belongs to $R(x)$.*

Before we prove the above lemmas, we introduce some useful definitions.

Definition: A path $P = (x_1, x_2, \ldots, x_{k+1})$ is a sequence of edges $\{(x_1, x_2, c_2), \ldots, (x_k, x_{k+1}, c)\}$ starting from x_1 and ending in x_{k+1}, with length k. A path is *simple* if all nodes are distinct. We obtain sub-path $P' = (x_i, ..., x_j)$ of P by removing the edges in the path P from x_1 to x_i and x_j to x_{k+1}. Given an edge (x, y, c), x is said to be tightly assigned with respect to y if $v_x = v_y + c$. Given a path $P = (x_1, x_2 \ldots, x_n)$ we say the path is tightly-assigned if for every edge (x_i, x_j, c), x_j is tightly assigned with respect to x_i.

Lemma 1.1. *Each $y \in V^d$ is updated at least once, i.e., $v_y \neq +\infty$.*

Proof: As G^d is connected, there is a path from some node in $x \in S$ to y or a path from y to some node $x \in S$. Therefore, v_y will be updated at least once at line 11 or line 7 of Figure 6.

Lemma 1.2. *The value assigned to a node by the bfm procedure (Figure 6: lines 8-14) belongs to some tightly assigned path P starting from $x \in S$. Furthermore, path P is simple.*

Proof: Every time *bfm* updates the value of a node, it gives a tightly assigned value (line 11, Figure 6) with respect to the previous node in the path and therefore, every assignment along path P is tightly assigned.

To show that path P is simple, it is easy to see that if the sub-graph G^d were a DAG (directed acyclic graph) then every assignment to a node is made along some simple path starting from $x \in S$. Since G^d is a consistent sub-graph, the only cycles permissible in G^d are *non-negative* cycles. We show that path P is simple by contradiction. Assume that v_y is updated along some non-simple path *P=(x,. . .,z,. . .,z,. . .,y)* starting at node $x \in S$, i.e., there exist some node z which has been updated at least twice by *bfm*. As tight values are assigned to nodes along the path, second update of v_z is possible only if the accumulated weight of the sub-path *(z,. . .,z)* is strictly negative. But this contradicts that G^d has only non-negative cycles. Therefore, we conclude that every update to y is made along some tightly-assigned simple path starting from some $x \in S$.

Lemma 1. *The procedure ASSIGN terminates.*

Proof: From Lemma 1.2, the number of updates on v_y (line 11, Figure 6) is at most equal to the number of simple paths to y from the nodes in S. Since the set S is finite, the number of simple paths to y and hence, the number of calls to *bfm* is also finite. Therefore, the procedure *bfm* terminates. One can also argue the termination of *bfm* based on the termination of Bellman-Ford-Moore algorithm for a satisfiable sub-graph.

Lemma 2. *The values assigned by ASSIGN satisfy all the inequalities represented by G^d.*

Proof: Consider the edge (u, v, c) corresponding to predicate $u+c \geq v$. We consider three cases depending on whether this edge was visited during *bfm*, and if not, whether v belongs to $K = V^d \cap C$ or not.

Case 1: *The edge was visited.*

We prove by contradiction. Assume, some inequality $u + c \geq v$ corresponding to the edge $(u, v, c) \in E^d$ is not satisfied, i.e., $fv_u + c < fv_v$ where fv_u, fv_v denote final values of u and v, respectively. Note, any calls to update v_v (line 11, Figure 6) only decrease v_v; therefore, $v_v \geq fv_v$. Since $fv_u \neq \infty$ (by Lemma 1.1), *bfm(v)* should have been invoked when v_u gets updated with fv_u as $(fv_u + c) < v_v$; thus, $v_v = fv_u + c$ and together with inequality $v_v \geq fv_v$, we get a contradiction. Therefore, the inequality for each visited edge (u, v, c) is satisfied.

Case 2: *The edge was not visited and* $v \notin K$.

Note, $u \notin K$. In this case, $v_u = Q[u]$ and $v_v = Q[v]$. As values allocated by *reverse dfs* satisfy all inequalities along the simple path from *non-cupoint* to *cutpoints*, $Q[u]+c \geq Q[v]$.

Case 3: *The edge was not visited and* $v \in K$.

Note, $u \notin K$. In this case, $v_u = Q[u]$ and $v_v \in R_f(v)$ (Lemma 3). Note, assignment of *reverse dfs* values ensures that $Q[u] + c \geq M$. But, since $M = max(\cup_{\forall x \in C} R_f(x))$, clearly, $Q[u] + c \geq v_v$ as $M \geq v_v$.

Lemma 3. *For any node y in a tightly assigned simple path P beginning with $x \in S$ whose value is 0, $v_y \in R_f(y) = \cup_{1 \leq k \leq |V|} L^k(y)$. Furthermore, if y is the k^{th} node in the path then $v_y \in L^k(y)$.*

Proof: It suffices to show that $v_y \in L^k(y)$ and $1 \leq k \leq |V|$ as $L^k(y) \subseteq R_f(y)$ for $1 \leq k \leq |V|$. Since a simple path in G^d cannot have more than $|V^d|$ nodes, it is easy to see $1 \leq k \leq |V^d| \leq |V|$.We prove $v_y \in L^k(y)$ by induction on the length of P.

Basis: Length of $P = 1$, i.e., $P = (x, y)$ with edge (x, y, c). The tightly assigned value of y, $v_y = v_x + c = c$. As per line 7 of Figure 6, $v_y \in L^2(y)$.

Induction: Given a node z in a tightly assigned simple path $P = (x, \ldots, z)$ of length $k - 1 (k > 1)$ such that $v_z \in L^k(z)$. Consider the path $P' = (x, \ldots, z, y)$ obtained by adding edge (z, y, d) to path P. Since y is tightly assigned with respect to z (Lemma 1.2), $v_y = v_z + d$. From line 7 in Figure 4, $v_y \in L^{k+1}(y)$. □

Appendix C: Theorem 2

Theorem 2. *Reduced ranges obtained by RCP (Range Constraint Propagation) are adequate for subgraph G^d.*

Proof: We show by induction that if G^d is satisfiable from some allocated range, then it is also satisfiable from the reduced ranges obtained after RCP.

Basis: Let α be a satisfying solution for subgraph G^d. The invariant $L(x) \leq \alpha(x) \leq U(x)$ is satisfied $\forall x \in V^d$ as all ranges are adequate for G^d (Theorem 1).

Induction: Assume we have applied RCP on n edges of the satisfiable subgraph G^d and that the reduced ranges are adequate, i.e., there exists a satisfying solution α such that all the invariants $L(x) \leq \alpha(x) \leq U(x)$ are satisfied $\forall x \in V^d$ where L and U denote the limits after applying RCP on n edges. Now, we apply RCP on the $(n + 1)^{th}$ added edge,

i.e., RCP($x + c \geq y$). We now show that α remains a satisfying solution from the newly reduced ranges. Let $L'(x)$, $U'(x)$ denote the changed limits as per Eq (2) after this step. Since α is a satisfying solution, $\alpha(x) \geq L(x)$ and $\alpha(y) \leq U(y)$. Moreover, since α satisfies the constraint $x + c \geq y$, $\alpha(x) \geq L(y) - c$ and $\alpha(y) \leq U(x) + c$. From definitions of L' and U', we obtain $\alpha(x) \geq L'(x)$ and $\alpha(y) \leq U'(y)$. Thus, the reduced ranges obtained by RCP are adequate for subgraph G^d. □

References

[1] W. Ackermann, "Solvable Cases of the Decision Problem," in *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1954.

[2] P. Niebert, M. Mahfoudh, E. Asarin, M. Bozga, O. Maler, and N. Jain, "Verification of Timed Automata via Satisfiability Checking," in *Proceedings of Formal Techniques in Real-Time and Fault Tolerant Systems*, 2002.

[3] J. Adams, E. Balas, and D. Zawack, "The shifting bottleneck procedure for job shop scheduling," in *Management Science*, Vol. **34**, No. 3, Focussed Issue on Heuristics (Mar., 1988), pp. 391–401.

[4] J.-C. Filliatre, S. Owre, H. Rueβ, and N. Shankar, "ICS: Integrated Canonizer and Solver," in *Proceedings of Computer-Aided Verification (CAV)*, 2001.

[5] G. Parthasarathy, M. K. Iyer, K.-T. Cheng, and C. Wang, "An Efficient Finite-Domain Constraint Solver for RTL Circuits," in *Proceedings of Design Automation Conference (DAC)*, 2004.

[6] S. Owre, J. M. Rushby, and N. Shankar, "PVS: A Prototype Verification System," in *Proceedings of Conference on Automated Deduction (CADE)*, 1992.

[7] D. Kroening, J. Ouaknine, S. A. Seshia, and O. Strichman, "Abstraction-Based Satisfiability Solving of Presburger Arithmetic," in *Proceedings of Computer-Aided Verification (CAV)*, 2004.

[8] A. J. C. Bik and H. A. G. Wijshoff, "Implementation of Fourier-Motzkin Elimination.," in *Technical Report 94-42, Dept. of Computer Science, Leiden University*, 1994.

[9] H. M. Sheini and K. A. Sakallah, "A Scalable Method for Solving Satisfiability of Integer Linear Arithmetic Logic," in *Symposium on the Theory and Applications of Satisfiability Testing (SAT)*, 2005.

[10] A. Armando, C. Castellini, E. Giunchiglia, M. Idini, and M. Maratea, "TSAT++: An Open Platform for Satisfiability Modulo Theories," in *Proceedings of Pragmatics of Decision Procedures in Automated Resonings (PDPAR)*, 2004.

[11] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P. V. Rossum, M. Schulz, and R. Sebastiani, "The MathSAT 3 system," in *Proceedings of Conference on Automated Deduction (CADE)*, 2005.

[12] SRI Team, "*Simplics*." http://fm.csl.sri.com/simplics.

[13] B. Dutertre and L. de Moura,"A fast linear-arithmetic solver for DPLL(T)," in *Proceedings of Computer-Aided Verification (CAV)*, 2006.

[14] G.B. Dantzig, "Maximization of a linear function of variables subject to linear inequalities," in: T.C. Koopmans, ed.,Activity Analysis of Production and Allocation (John Wiley and Sons, New York), 1951.

[15] L. Zhang and S. Malik, "The Quest for Efficient Boolean Satisfiability Solvers," in *Proceedings of Computer-Aided Verification (CAV)*, 2002.

[16] A. Pnueli, Y. Rodeh, O. Strichman, and M. Siegel, "The Small Model Property: How small can it be?," in *Information and Computation*, vol. **178**(1), Oct 2002, pp. 279-293.

[17] O. Strichman, S. A. Seshia, and R. E. Bryant, "Deciding Separation Formulas with SAT," in *Proceedings of Computer-Aided Verification (CAV)*, July 2002.

[18] R. E. Bryant, S. K. Lahiri, and S. A. Seshia, "Modeling and Verifying Systems using a Logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions," in *Proceedings of Computer-Aided Verification (CAV)*, 2002.

[19] S. A. Seshia, S. K. Lahiri, and R. E. Bryant, "A Hybrid SAT-based Decision Procedure for Separation Logic with Uninterpreted Functions," in *Proceedings of Design Automation Conference (DAC)*, 2003.

[20] R. E. Bryant, S. K. Lahiri, and S. A. Seshia, "Deciding CLU logic formulas via Boolean and pseudo-Boolean encodings," in *Workshop on Constraints in Formal Verification*, 2002.

[21] M. Talupur, N. Sinha, and O. Strichman, "Range Allocation for Separation Logic," in *Proceedings of Computer-Aided Verification (CAV)*, 2004.

[22] A. Stump, C. W. Barrett, and D. L. Dill, "CVC: A Cooperating Validity Checker," in *Proceedings of Computer-Aided Verification (CAV)*, 2002.

[23] R. Nieuwenhuis and A. Oliveras, "DPLL(T) with Exhaustive Theory Propogation and its Application to Difference Logic," in *Proceedings of Computer-Aided Verification (CAV)*, 2005.

[24] C. Wang, F. Ivancic, M. Ganai, and A. Gupta, "Deciding Separation Logic Formulae with SAT by Incremental Negative Cycle Elimination," in *Proceedings of Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, 2005.

[25] C. Wang, A. Gupta, and M. K. Ganai, "Predicate learning and selective theory deduction for a difference logic solver," in *Proceedings of Design Automation Conference (DAC)*, 2006.

[26] F. Aloul, A. Ramani, I. Markov, and K. Sakallah, "PBS: A backtrack search pseudo-Boolean solver," in *Symposium on the Theory and Applications of Satisfiability Testing (SAT)*, 2002.

[27] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. Cambridge, MA: MIT Press, 1990.

[28] S. K. Lahiri, and K. K. Mehra, "Interpolant-based Decision Procedure for Quantifier-free Presburger Arithmetic," in *Technical Report MSR-TR-2005-121*, 2005. http://research.microsoft.com.

[29] J. Whittemore, J. Kim, and K. Sakallah, "SATIRE: A new incremental satisfiability engine," in *Proceedings of Design Automation Conference (DAC)*, 2001.

[30] M. Ganai, M. Talupur, and A. Gupta, "SDSAT: Tight integration of lazy and eager approaches in difference logic solver," in *Proceedings of Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, 2006.

[31] S. Cotton, "Satisfiability Checking with Difference Constraints," in *Master Thesis*, IMPRS Computer Science, Saarbruceken, 2005.

[32] V. Pratt, "Two Easy Theories Whose Combination is Hard," in *Technical report*, MIT, Cambridge, 1977.

[33] G. Ramalingam, J. Song, L. Joscovicz, and R. Miller, "Solving difference constraints incrementally," in *Algorithmica* **23**: 261Y275, 1999.

[34] O. Strichman. http://iew3.techninon.ac.il/~ofers.

[35] H. Kim and F. Somenzi, "Finite Instantiations for Integer Difference Logic," in *Proceedings of Formal Method in Computer-Aided Design (FMCAD)*, 2006.

[36] D. S. Hochbaum, *Approximation Algorithms for NP-hard Problems*: PWS Publishing Company, 1997.

[37] S. Cotton and O. Maler, "Fast and Flexible Difference Constraint Propagation for DPLL(T)," in *Symposium on the Theory and Applications of Satisfiability Testing (SAT)*, pp. 170–183, 2006.

[38] B. V. Cherkassky and E. Goldberg, "Negative-cycle Detection Algorithms," in *European Symposium on Algorithms*, 1996.

[39] R. E. Moore, *Interval Analysis*. NJ: Prentice-Hall, 1966.

[40] T. Hickey, Q. Ju, and H. V. Emden, "Interval arithmetic: From principle to implementation," in *Journal of the ACM*, Vol. **48**, pp 1038–1068, 2001.

[41] V. Kumar, "Algorithms for Constraint Satisfaction Problems: A Survey," *AI Magazine*, **13**:32–44, 1992.

[42] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an Efficient SAT Solver," in *Proceedings of Design Automation Conference (DAC)*, 2001.

[43] SMT-LIB. http://goedel.cs.uiowa.edu/smtlib.