SDSAT: Tight Integration of *Small Domain Encoding* and *Lazy* Approaches in a Separation Logic Solver

Malay K Ganai[1], Muralidhar Talupur[2], and Aarti Gupta[1]

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

Abstract. Existing Separation Logic (*a.k.a Difference Logic, DL*) solvers can be broadly classified as *eager* or *lazy,* each with its own merits and de-merits. We propose a novel Separation 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 separation 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 Separation Logic formula is not transformed into an equi-satisfiable Boolean formula in one 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 can not 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.

1 Introduction

Separation Logic, (*a.k.a Difference Logic*, DL) extends propositional logic with predicates of the form $x+c \rhd y$ where $\rhd \in \{>,\geq\}$, 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 to Boolean equalities [1]. Separation 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 Separation Logic problem is *NP-Complete*. Decision procedures based on graph algorithms use a weighted directed graph to represent Separation predicates; with nodes representing variables

H. Hermanns and J. Palsberg (Eds.): TACAS 2006, LNCS 3920, pp. 135 – 150, 2006.

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 separation 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 the predicates are of the form $\Sigma_i\ a_i x_i \geq c$ where a_i, c are constants and x_i are variables. Most of them *ICS* [4], *HDPLL* [5], *PVS* [6], and *ASAP* [7] are based on a variable elimination technique like Fourier-Motzkin [8]. Here, we restrict ourselves to a discussion of decision procedures dedicated for Separation Logic.

Satisfiability of a Separation 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 [9]. 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 [10-14] , the Separation formula is reduced to an equi-satisfiable Boolean formula in one 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*) [12] or by deriving all possible transitivity constraints (*a.k.a per-constraint encoding*) [11]. A hybrid method [13] combines the strengths of the two encoding schemes and was shown to give robust performance over the two. In lazy approaches [15-19], SAT is used to obtain a possibly feasible model corresponding to a conjunction of separation 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 [12, 20] reduce the range space allocated to a finite domain, Boolean encoding of the formula often leads to large propositional logic formula, eclipsing the advantage gained from the reduced search space. Researchers [14] have also experimented with the pseudo-Boolean Solver *PBS* [21] to obtain a polynomial size formula, but without any significant performance gain. In a *per-constraint encoding*, the formula is abstracted by replacing each predicate with a Boolean variable, and then pre-emptively 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 backseat 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 [22] to detect 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. Moreover, several techniques have been proposed [17, 18] to add pre-emptively 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. Due to this reason, lazy approaches do not perform as well as eager approaches on benchmarks like *diamonds* which have an exponential number of cycles (~2^n cycles where n is the number of variables). Thus, we are naturally motivated to combine the strength of the two approaches as tightly as possible.

We propose a robust Separation Logic Solver *SDSAT* (short for 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.

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

2 Background: Separation Logic

Separation predicates are of the form $x+c \rhd y$ where $\rhd \in \{>,\geq\}$, c is a constant and x, y are variables of some ordered infinite type *integer* or *real*, D. Separation Logic is a decidable theory combining Propositional Logic with these predicates. 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 [16]. Note, an inequality of the form $x \rhd c$, can be also be translated into a weak inequality of two variables, by introducing a reference node z. Henceforth, we will consider separation 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 Separation Logic solvers based on lazy approaches as shown in Figure 1.

Problem Formulation: In this class of decision procedures, a Separation formula φ is abstracted into 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 Separation formula ψ to $\{T, F\}$ by defining the following mapping over the Separation 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 separation predicates. Note, incremental solvers [17, 19, 23] 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.

Constraint Feasibility: Any partial assignment (also referred to as a partial Boolean model) to variables in φ_B represents a conjunction of separation predicates. The Boolean model is represented as a weighted directed graph (*a.k.a* constraint graph) [24], 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 separation 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 [25] 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. Problem is declared *satisfiable* by Boolean SAT, if there are no more assignments to make.

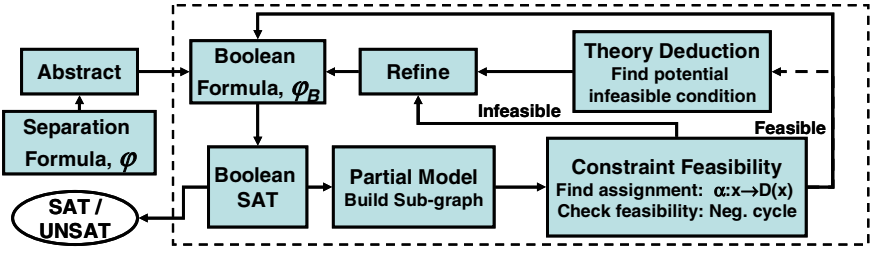


Fig. 1. Overview of state-of-the-art Separation Logic Solver based on lazy approach

Refinement: Whenever a negative cycle is encountered during constraint feasibility (*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 [16], 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, though possible, is not done usually. Moreover, like in Boolean SAT solvers, incremental solvers [17, 19, 23] restore the assignments to the variables to a state just before the inconsistency was detected, instead of starting from scratch.

Pre-emptive Learning (Theory Deduction): Some solvers [17, 18] have capabilities to add transitivity constraints preemptively to φ_B so as to avoid finding them later.

However, as the overhead of adding all transitivity constraints can be prohibitive as observed in *per-constraint* eager approach, 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 Separation Logic which has been discussed at greater depth in [20, 26]. Let $Vars(\varphi)$ denote the set of variables used in a Separation 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 $Vars(\varphi)$ to 2^Z. Let $Vars(\varphi) = \{v_1,\ldots,v_n\}$ and $|R(v_i)|$ denote the number of elements in the set $R(v_i)$, 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

$$SAT_R(\varphi) \Leftrightarrow SAT_Z(\varphi) \tag{1}$$

We say that a domain R is *adequate* for φ if it satisfies formula (1). As 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 separation formulas with the *same set of predicates* as φ, denoted by $\Phi(\varphi)$ as adequacy for $\Phi(\varphi)$ implies adequacy for φ. As discussed in the previous section, separation predicates can be represented by a constraint directed graph $G(V,E)$. Thus, the set of all the sub-graphs of G represents the set $\Phi(\varphi)$. Given G, the range allocation problem is setup to finding a domain R such that every consistent sub graph of G can be satisfied from the values in R.

It has been shown [12] that for a Separation formula with n variables, a range $[1..n+maxC]$ 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* [12], 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 [20] to reduce the overall ranges and hence, the size of the Boolean formula. In [20], a method *SMOD* was proposed to allocate non-uniform ranges to variables, exploiting the problem structure. The method builds cut-point SCC graph recursively in top-down manner and allocates ranges to the nodes in a bottom-up style, 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 this paper, we propose an efficient and robust technique *Nu-SMOD* that computes non-uniform ranges in polynomial-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* [12]. In experimental evaluation, *Nu-SMOD* completes allocation for all the benchmarks unlike *SMOD*, with 1-2 orders of magnitude performance

improvement over *SMOD*. Unlike *SMOD*, we do not compute cutpoint-graph or enumerate cycles in our new procedure *Nu-SMOD*; rather we propagate only distinct values along a path from a cut-point. As the ranges will be used subsequently by the lazy search-engine, we emphasize improvement in performance instead of ranges. Thus, our objective differs slightly from *SMOD* procedure.

3 SDSAT: Integrating Small Domain and Lazy Approaches

We propose a Separation Logic Solver *SDSAT* as shown in Figure 2, that combines the strengths of both eager (small domain encoding) and lazy approaches and gives 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 the 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, the Separation Logic formula is not transformed into an equi-satisfiable Boolean formula in one step, but rather done lazily in the following phase.

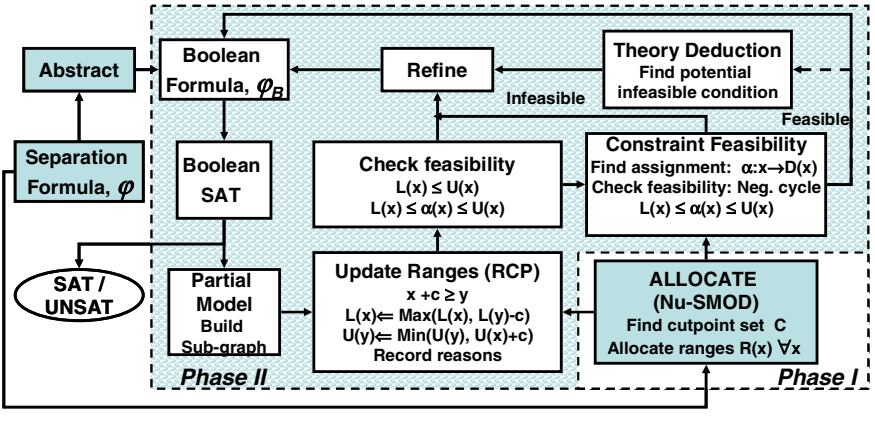


Fig. 2. Overview of our Separation Logic solver *SDSAT*

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 can not 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 separation 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, 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* to allocate non-uniform ranges to the variables in the predicates. The algorithm assumes that the constraint directed graph *G(V,E)* is a Strongly Connected Component (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 Separation 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 first derive a cutpoint set *C* using polynomial approximation [27], as finding a minimal cutpoint set is an NP-Hard problem. Using the set *C*, we invoke the procedure (line 1 of the procedure *NU-SMOD*, Figure 3) *Nu-SMOD-1* which is described as follows (lines 11-19, Figure 3): Range of each node *x*, denoted by *R(x),* is divided into several sets; each identified with unique id or simply *level*. Let the level *k* set of the node *x* be denoted by $L^k(x)$. Note, $R(x) = \cup_k L^k(x)$. Initially, all the level sets are empty. The nodes in Level 1 set, denoted by I, are allocated 0 value, i.e., $L^1(x) = \{0\}$, $\forall x \in I$. To compute Level *(k+1)* values, i.e., $L^{k+1}(y)$ for node *y* (line 17), we use the Level *k* values of the nodes *x* that have a direct edge, i.e., fanout *(x,y,c)* (corresponding to the predicate $x + c \geq y$) to *y* and offset with edge weight *c*. Note, we include only the cutpoints *C* in the set *I*. Once the ranges for the cutpoints *C* are obtained using *Nu-SMOD-1*, another pass is made (in lines 2-5) to obtain *reverse_dfs* values *Q[y]* for each non-cutpoint *y*. Starting from each cutpoint (line 4-5) with value *M* (equal to maximum range value allocated among the cutpoints), we do a *reverse_dfs* (lines 7-10) to update *Q* values (line 9) of all the non-cutpoints by reverse propagating a tight value (higher than the previous *Q* value, line 8) without traversing through any other cutpoints (line 7). 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 Allocation for an
//SCC G(V,E)

Input: G(V,E), Cutpoint set C
Output: R(x) for ∀x∈V
Procedure: Nu-SMOD
1. Nu-SMOD-1 {INPUT: G(V,E), I=C
              OUTPUT: R(x)∀x∈V }
2. ∀y∈V-C Q[y]=-∞
3. M=max(∪∀x∈CR(x))
4. foreach x∈C do
5.   reverse_dfs(x,M);
6. end

reverse_dfs(x,v)
7. foreach (y,x,w) s.t. y∉C do
8.  if (Q[y]+w ≥ v) continue;
9.  Q[y] = v - w;
10. reverse_dfs(y,Q[y])



Input: G(V,E), I⊆V
Output: R(x) for ∀x∈V
Procedure: Nu-SMOD-1

11. L^1(x)={0}∀x∈I, L^1(x)={}∀x∈V\I
12. L^k(x)={} ∀x∈V, 1<k≤|V|
13. foreach k, 1≤k<|V| do
14.  foreach node x∈V do
15.   foreach(x,y,c)∈fanouts(x)do
16.    foreach value v ∈ L^k(x) do
17.      L^k+1(y)=L^k+1(y)∪{v+c}
18.
19. ∀x∈V  R(x)=∪1≤k≤|V| L^k(x)

//Assignment for subgraph D of G
Input: D(V^d,E^d)
Output: {(x,v_x)|x∈V^d, v_x∈R(x)}
Procedure: ASSIGN
20. S = {set of root nodes}
21. ∀y ∈ V^d-S v_y = +∞;
22. foreach x ∈ S do
23.    v_x = 0; enqueue(x)
24.    bfm(x);
25. end
26. ∀y∈V^d-S if (v_y == +∞) v_y=Q[y]

bfm(x)
27. while (x = dequeue())!=null)
28.  foreach(x,y,c)∈fanouts(x)do
29.     if (v_x+c >= v_y) continue;
30.     v_y = v_x+c;
31.     enqueue(y);
32.   end
33. end
```

Fig. 3. Pseudo-code for the algorithm *Nu-SMOD* and *ASSIGN* procedures

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

Proof Sketch: We now show that the ranges allocated by *Nu-SMOD* are *adequate*, i.e., any satisfiable sub-graph $D(V^d,E^d)$ 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 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 3 (lines 20-33) 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 can not be reached from any other node in $V^d \cap$ C) in D (line 20). If 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 26, 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 21). 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 23), we call *bfm (*similar to Bellman-Ford-Moore Shortest Path algorithm [22]*)* 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 30). Such an operation is also called an *edge relaxation* [22]. Only under such a scenario, the node is en-queued (line 31). Those nodes whose value are still $+\infty$, are given *reverse_dfs* Q values (line 26). To show that the given assignment procedure *ASSIGN* generates a satisfying solution from the ranges allocated, we need to prove the following lemmas (Contact author for proof details).

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 $R(x)$.

The above theorem guarantees the existence of the solution for a satisfying subgraph D 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. 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.

3.2 Solve Phase

Similar to standard lazy solvers, we first build an abstract Boolean formula φ_B from the given Separation formula φ and search for a partial consistent Boolean model. As the partial model is being incrementally built up, we search for a satisfying model using *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 above) 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 [17, 18]).

3.2.1 Incremental Cycle Detection Using Cutpoint Relaxation

In the past [23, 28], 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 (BFM) Shortest Path algorithm and extends it with an ability to detect negative cycle. Our approach is also based on BFM with the following difference: *For a satisfiable sub-graph 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 subgraph 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 restriction in our incremental

cycle detection algorithm *cutpoint relaxation*. As will be clear shortly, the theoretical complexity of the algorithm is not different from BFM 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 D and obtain a subgraph D'. If $\alpha(x) \neq +\infty$, x is reachable from some root node in D and we do the usual BFM. 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 D' as it is not reachable from any other root node in D. We choose $\alpha(x)=0$ and do usual BFM 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 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, yet 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 D; however, the ranges are often more than those required to obtain a satisfying solution for D. We allow range constraint propagation (RCP) to dynamically refine the ranges of the variables for the given subgraph D, while maintaining the range adequacy (Theorem 2). This approach is similar to the more general approach for interval arithmetic [29, 30]. We achieve RCP as follows: Let the minimum and maximum 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:

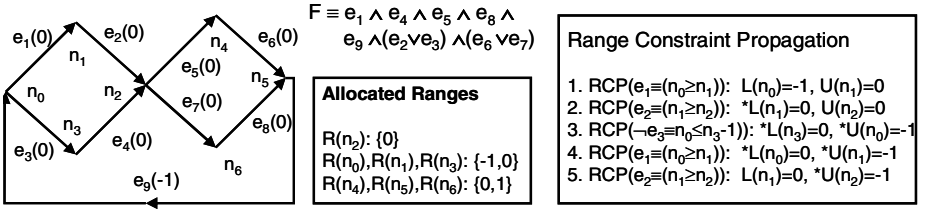
$$L(x) \Leftarrow MAX\{L(x), L(y)-c\}$$
$$U(y) \Leftarrow MIN\{U(y), U(x)+c\}$$

We apply this process recursively, i.e., whenever the L (or U) value of a node changes, we update the L (or U) values of all the nodes with a direct edge to (or from)

the 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 (please contact authors for proof details).

Theorem 2: Reduced ranges obtained by RCP are adequate for subgraph D.

Example: We illustrate RCP and its roles in reducing the search space on a diamond example shown in Figure 4. Let the Separation 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 separation predicate. Let $n_0 ... n_5$ represent the integer variables. The separation predicates are shown as edges e_i in Figure 4(a) (with weights in brackets). For example: $e_1 \equiv (n_0 \geq n_1)$ and $e_9 \equiv (n_5 - 1 \geq n_0)$. The previous approaches based on *only* negative cycle detection have to find all four negative cycles before 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.



(a) (b) (c)

Fig. 4. (a) Example (b) Allocated Ranges (c) RCP w/ negative cycle detection

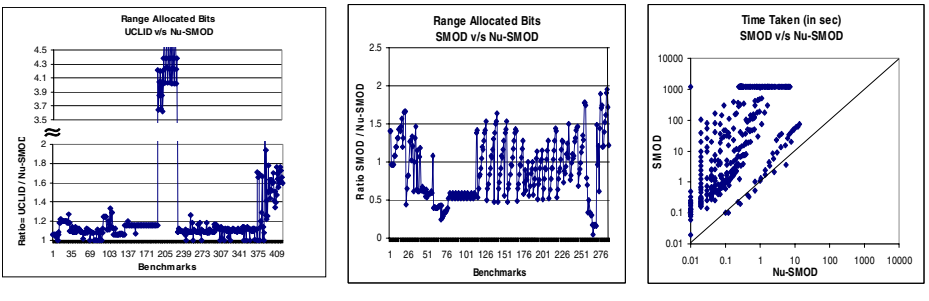
As shown in Figure 4(b), L and U of each variable are initially set to corresponding minimum and maximum range R values as obtained by *Nu-SMOD* (for example: $L(n_0)=-1$, $U(n_0)=0$). Note, that these ranges are adequate for this graph. Consider the subgraph $e_1 \wedge e_2 \wedge \neg e_3$. When we apply RCP as shown in the Figure 4(c), we detect a range violation as follows (note, $*L$ and $*U$ denote changes from the previous step): 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 and learn a clause $(\neg e_1 \vee \neg e_2 \vee e_3)$ by doing conflict analysis. 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.

4 Experimental Results

We have integrated our incremental cycle detection using cutpoint relaxation and RCP with the zChaff Boolean SAT solver [31]. 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.* 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 the state-of-the-art solvers.

Comparison of Range Allocations Algorithms: We compared our approach *Nu-SMOD* with previous approaches *SMOD* [20] and *UCLID* [12] on these benchmarks and present results in Figure 5. We used a time limit of 2 minutes for each run. Note, the *UCLID* procedure allocates 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 $R(y)$.



(a) (b) (c)

Fig. 5. Ratio of range bits allocated between (a) UCLID v/s Nu-SMOD, (b) SMOD v/s Nu-SMOD. (c) Scatter plot of time taken (in sec) between SMOD v/s Nu-SMOD.

UCLID v/s Nu-SMOD: As shown in Figure 5(a), compared to *UCLID*, *Nu-SMOD* allocates on average about 40% less range bits (about 4X less on *diamond set*). Note that such linear reductions amount to exponential reduction in range space.

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 [20]. Baring a few benchmarks, the ranges allocated by *Nu-SMOD* are comparable to *SMOD* as seen in Figure 5(b). Moreover, *SMOD* is 1-2 orders of magnitude slower on the completed benchmarks as compared to *Nu-SMOD* as shown in the scatter plot (in logarithmic scale) in Figure 5(c).

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 distinct 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 the Table 1, Column 1 shows the benchmarks suites with the number in brackets indicating the number of problems considered. Columns 2-3 show the results of *allocation phase*: especially, Column 2 shows the average size of range bits per variable computed in the *allocation phase.* Column 3 shows the average time taken for *allocation phase*. Columns 4-5 show the result 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 averages 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 small domain encoding [12]. Note that in the presence of RCP the refinement overhead is not affected significantly. Moreover, a lower refinement penalty with RCP indicates improvement in 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.

Comparison with other Separation Logic Solvers: In the third set of experiments, we compare our approach *SDSAT* (the *solve phase*) with other latest available state-of-the-art tools, including *UCLID[13], MathSAT[17], ICS[4], TSAT++[16],* and *DPLL(T)[18].* As *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 *DPLL(T)* have superior performance compared to other lazy and eager approaches by several orders of magnitude. Comparing *SDSAT* with *DPLL(T)*, 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 *DPLL(T)*. Thus, overall our approach is relatively more robust than the pure lazy approaches which can also benefit using our ideas.

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 violation (%)
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

Table 2. Performance comparison (in sec) of state-of-the-art Separation Logic Solvers

Bench	TSAT++	UCLID	MathSAT	ICS	DPLL(T)	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)

5 Conclusions

We proposed a novel Separation Logic Solver *SDSAT* that takes advantage of the small domain property of Separation logic to perform a lazy search of the state space. The solver tightly integrates the strengths of both lazy and eager approaches and gives a 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 cutpoint relaxation algorithm. Experimental evidence presented here bears out the efficacy of our technique.

References

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

[2] P. Niebert, M. Mahfoudh, E. Asarin, M. Bozga, O. Maler, and N. Jain, "Verification of Timed Automata via Satisfiability Checking," in *Proc. 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*, 1988.

[4] J.-C. Filliatre, S. Owre, H. Rueβ, and N. Shankar, "ICS: Integrated Canonizer and Solver," in *Proceedings of 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 DAC*, 2004.

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

[7] D. Kroening, J. Ouaknine, S. A. Seshia, and O. Strichman, "Abstraction-Based Satisfiability Solving of Presburger Arithmetic," in *Proceedings of 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] L. Zhang and S. Malik, "The Quest for Efficient Boolean Satisfiability Solvers," in *Proceeding of CAV*, 2002.

[10] 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.

[11] O. Strichman, S. A. Seshia, and R. E. Bryant, "Deciding Separation Formulas with SAT," in *CAV*, July 2002.

[12] 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 *Computer-Aided Verification*, 2002.

[13] 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 DAC*, 2003.

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

[15] C. Barrett, D. L. Dill, and J. Levitt, "Validity Checking for Combination of Theories with Equality," in *Proceedings of FMCAD*, 1996.

[16] 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'04)*, 2004.

[17] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P. V. Rossum, S. Schulz, and R. Sebastiani, "An Incremental and Layered Procedure for the Satisfiability of Integer Arithmetic Logic," in *Proceedings of TACAS*, 2005.

[18] R. Nieuwenhuis and A. Oliveras, "DPLL(T) with Exhaustive Theory Propogation and its Application to Difference Logic," in *CAV*, 2005.

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

[20] M. Talupur, N. Sinha, and O. Strichman, "Range Allocation for Separation Logic," in *CAV*, 2004.

[21] 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.
[22] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. Cambridge, MA: MIT Press, 1990.
[23] S. Cotton, "Satisfiability Checking with Difference Constraints," in *IMPRS Computer Science, Saarbruceken*, 2005.
[24] V. Pratt, "Two Easy Theories Whose Combination is Hard," in *Technical report, MIT*, 1977.
[25] G. Ramalingam, J. Song, L. Joscovicz, and R. Miller, "Solving difference constraints incrementally," in *Alogrithmica*, 1999.
[26] O. Strichman, "http://iew3.techninon.ac.il/~ofers."
[27] D. S. Hochbaum, *Approximation Algorithms for NP-hard Problems*: PWS Publishing Company, 1997.
[28] B. V. Cherkassky and E. Goldberg, "Negative-cycle Detection Algorithms," in *European Symposium on Algorithms*, 1996.
[29] R. E. Moore, *Interval Analysis*. NJ: Prentice-Hall, 1966.
[30] T. Hickey, Q. Ju, and H. V. Emden, "Interval Arithmetic: from principles to implementation," in *Journal of the ACM*, 2001.
[31] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an Efficient SAT Solver," in *Proceedings of Design Automation Conference*, 2001.