Range Allocation for Separation Logic

Muralidhar Talupur[1], Nishant Sinha[1], Ofer Strichman[2], Amir Pnueli[3]

[1]Carnegie Mellon University, Pittsburgh, PA, USA
[2] Technion - Israel Institute of Technology, Haifa, Israel
[3]The Weizmann Institute of Science, Rehovot, Israel

Abstract. *Separation Logic* consists of a Boolean combination of predicates of the form $v_i \geq v_j + c$ where c is a constant and v_i, v_j are variables of type `real` or `integer`. Any equality or inequality can be expressed in this logic. We propose a decision procedure for `integer` Separation Logic based on assigning small ranges to variables in a formula such that satisfiability of the given formula is preserved. Given formula φ the procedure constructs an *inequalities graph* which represents all formulas with the same set of predicates as φ. The procedure then constructs ranges for each variable by analysing this graph. This approach of finding small finite ranges and enumerating them symbolically is empirically far more efficient than methods based on case-splitting. Experimental results show that the state space (that is, the number of assignments that need to be enumerated) allocated by our procedure is frequently exponentially smaller than previous methods.

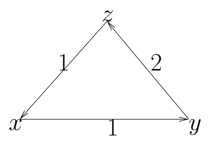
1 Introduction

We propose a decision procedure for *separation logic* based on computing a small set of assignments for each variable in a given formula which are provably sufficient for preserving its satisfiablity. Separation logic, also known as Difference logic, consists of a Boolean combination of predicates of the form $v_i \rhd v_j + c$ where $\rhd \in \{>, \geq\}$, c is a constant, and v_i, v_j are variables of type `real` or `integer`. All the other equality and inequality relations can be expressed in this logic. Uninterpreted functions can be handled as well since they can be reduced to Boolean combinations of equalities[1]. In this paper, we consider Seperation logic with `integer` constants only. Further we consider only predicates with $\geq$ relations as it does not reduce expressivity in any way.

Separation predicates are used in verification of timed systems, scheduling problems, and more. Hardware models with ordered data structures have inequalities as well. For example, if the model contains a queue of unbounded length, the test for *head* $\leq$ *tail* introduces inequalities. In fact, as observed by Pratt [6], most inequalities in verification conditions are of this form. Furthermore, since theorem provers can decide mixed theories (by invoking an appropriate decision procedure for each logic fragment[7]), a decision procedure for Separation logic will be helpful in efficient verification of any formula that contains a significant number of these predicates.

Existing decision procedures for separation logic. There are two distinct classes of decision procedures for Separation logic. The first class, which is also the more traditional one, is based on a combination of case splitting and a decision procedure for a conjunction of separation predicates (normally it is solved with a decision procedure for the richer logic of linear arithmetic, like Simplex or Fourier-Motzkin). In the naive approach, case-splitting is implemented by first transforming the formula to DNF, and then solving each clause separately with a decision procedure. There are well-studied improvements to the naive case-splitting approach, most notably the SAT-based approach implemented in several recently published systems such as CVC [9], MATHSAT [2] and others. The idea adopted by these tools is to abstract each predicate with a new Boolean variable, and search for a satisfying assignment to this abstracted formula with a standard SAT solver. The consistency of this assignment with respect to the underlying arithmetic constraints is then checked with a decision procedure for a conjunction of such constraints. If it is consistent then the procedure exits and declares the formula to be satisfiable. Otherwise it adds a constraint to the abstracted propositional formula such that the constraint prunes other solutions that contain the same subset of predicates that were just proven to be unsatisfiable. This approach has a clear advantage over naive case-splitting because it enables learning and pruning. Yet, in the worst case, it still invokes the decision procedure for conjunction of constraints an exponential number of times.

The second class of procedures is based on a full reduction to propositional logic. We are aware of three such decision procedures. The first two [8] are based on an analysis of a graph derived from the formula, called the *inequalities graph*. The inequalities graph is based on the formula's predicates, regardless of the Boolean connectives between them. Both methods encode each predicate of the form $x \geq y + c$ with a new Boolean variable e^c_{xy}, and then add constraints to the formula based on an analysis of the inequalities graph. In the first method, called *cycle enumeration*, a constraint is added to the encoded formula so as to forbid cycles with non-negative accumulated weight. Such cycles represent the only unsatisfiable combination of edges. In the second method, called *variable elimination*, nodes are gradually removed from the graph while adding transitivity constraints (similar to the Fourier-Motzkin technique). The following example illustrates both methods. Consider the formula $\varphi : x \geq y{+}1 \wedge (y \geq z{+}2 \vee z \geq x{+}1)$. As a first step, both methods abstract this formula with Boolean constraints, i.e. $\varphi' : e^1_{xy} \wedge (e^2_{yz} \vee e^1_{zx})$. As a second step, they construct the following graph:

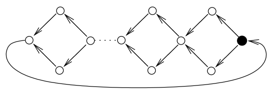


The third step is different between the two methods. In the cycle enumeration method, a constraint $\neg(e^1_{xy} \wedge e^2_{yz} \wedge e^1_{zx})$ is added to φ', because the sum of weights in this cycle is non-negative. In the variable elimination method, the nodes (variables) are eliminated one at a time while adding proper constraints to the formula. Given the order x, y, z it first eliminates x, deriving from the first and third predicates the new constraint $z - 1 \geq y + 1$. It consequently adds the constraint $e^1_{xy} \wedge e^1_{zx} \rightarrow e^2_{zy}$. Eliminating y we derive from the second and fourth constraint the new constraint $z - 2 \geq z + 2$ which is of course unsatisfiable. We thus add the constraint $e^2_{yz} \wedge e^2_{zy} \rightarrow$ FALSE .

Our approach. Our approach is based on the small model property of separation logic. That is, if a formula in this theory is satisfiable, then there is a finite model that satisfies it. Furthermore, in the case of Separation logic there exists an efficiently computable over-approximation of the finite model. This implies that the given formula can be decided by checking all possible valuations from the over-approximate model.

In the case of constraints of the form $x \rhd y$ the range $[1 \ldots n]$ is sufficient, where n is the number of variables. In other words, it is possible to check for the satisfiability of such formulas by enumerating all n^n possible valuations within this range. In the case of full separation logic, it was shown [3] that a range $[1 \ldots n + maxC]$ is required for each variable, where $maxC$ can be as high as the sum of all constants in the formula. This result leads to a state space of $(n + maxC)^n$. In this article we investigate methods for reducing this number, based on a graph analysis of the formula. A similar approach was taken in the past by Pnueli et al. [5] in the context of equality logic (Boolean combinations of equalities). This article can be seen as a natural continuation of that work.

As an example of the reduction in state space that our method can achieve consider the *diamond* graph shown below. For such a graph with n nodes and all edge weights equal to 1, the second approach discussed above will allocate n values for each node, making the total state-space size $O(n^n)$. In contrast, our approach only allocates 2 values to each node except the cutpoint (which is given a single value), resulting in a state-space with size $O(2^n)$. If the graph has non-zero edge weights, the state space in the second approach size would blow up to $O((n + \textit{sum-of-max-constants})^n)$, in our approach it would be the same.



2 Problem Formulation

Let $Vars(\varphi)$ denote the set of variables used in a Separation formula φ over the set of integers $\mathbb{Z}$. A domain (or range) $R(\varphi)$ of a formula φ is a function from $Vars(\varphi)$ to $2^{\mathbb{Z}}$. Let $Vars(\varphi) = \{v_1, \ldots v_n\}$ and $|R(v_i)|$ be equal to the number of elements in the set $R(v_i)$. The size of domain $R(\varphi)$, denoted by $|R(\varphi)|$ is given by $|R(\varphi)| = |R(v_1)| \cdot |R(v_2)| \cdot \cdots \cdot |R(v_n)|$. Now, let $\models_R \varphi$ denote that φ is valid in a domain R. Our goal is the following:

Find a small domain R such that

$$\models_R \varphi \iff \models_{\mathbb{Z}} \varphi \qquad (1)$$

We say that a domain R is *adequate* for φ if it satisfies formula (1). It is straightforward to see that if $\models_{\mathbb{Z}} \varphi$ holds then the minimal adequate domain has size 1: simply assign each variable in $Vars(\varphi)$ some constant value. Thus, finding the smallest domain for a given formula is at least as hard as checking the validity of φ. So, rather than examining φ we investigate the set of all separation formulas with the *same set of predicates* as φ, denoted by $\Phi(\varphi)$. Thus, our new, less ambitious goal is the following:

> Given a separation formula φ, find the smallest domain R which is adequate for $\Phi(\varphi)$.

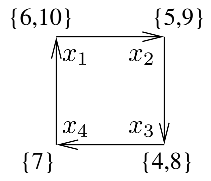
The problem of finding the smallest adequate domain for $\Phi(\varphi)$ is computationally expensive. We will therefore concentrate on finding over-approximations which are easier to compute. As was previously indicated, our solution is based on a graph analysis of the formula. We now describe the same problem in graph-theoretic terms.

Given a Separation logic formula φ, construct the *inequalities graph* $G_\varphi(V, E)$ as follows (we will write G_φ from now on). Add a node to V for each $v \in Vars(\varphi)$. For each predicate of the form $x \geq y + c$ in φ, add an edge to E from the node representing x to the node representing y, with a weight c. Note that each subgraph of G_φ represents a particular formula in $\Phi(\varphi)$ and vice-versa. Hence, we will say that a domain is *adequate for a graph* G_φ if it is adequate for $\Phi(\varphi)$.

We say that an subgraph of G_φ is *consistent* if it does not include a cycle with positive accumulated weight (*positive cycle* for short). Intuitively, a consistent subgraph represents a set of edges (constraints) that can be satisfied simultaneously. It was shown in [8] that this condition is sufficient and necessary for satisfiability of separation formulas. For example, the inequality graph presented in Section 1 is inconsistent as there is a positive cycle in it. Restating in graph-theoretic terms, our goal is:

> Given G_φ, find a domain R for each node in V such that every consistent sub graph of G_φ can be satisfied from values in R.

Example 1. The set of constants associated with each vertex in the following graph constitute an adequate domain. Each edge is assumed to have weight 1. There exists a solution from these sets to every consistent subset of edges in the graph.



For example, the set of edges $\{(x_1, x_2), (x_2, x_3), (x_3, x_4)\}$ can be satisfied by the assignment $x_1 = 10$,$x_2 = 9$,$x_3 = 8$,$x_4 = 7$. Note that there is no need to satisfy the subset containing all the edges because it is inconsistent.

It can be shown that for every simple cycle there exists an adequate domain with size $2^{|V|-1}$. In our example the size of the domain is 8. If we try to reduce this size, for example by setting $R(x_1) = \{6\}$, the domain becomes inadequate, because the subset $(x_1, x_2), (x_2, x_3), (x_3, x_4)$ is unsatisfiable under R. □

3 Range allocation algorithm

As mentioned in the introduction, it is known that for every inequalities graph (with all edge weights equal to 1), the range $1 \ldots |V|$ is adequate, resulting in a state space of size $|V|^{|V|}$. However Example 1 shows that analysis of the graph structure may yield smaller state-spaces. For example, if a graph has two unconnected sub-graphs then they can be allocated values separately, hence reducing the overall state-space. In fact, it is possible to analyze every Strongly Connected Component (SCC) separately. The edges between the SCCs can then be satisfied by appropriately shifting the allocated domains of each SCCs. Thus, a solution to this problem for SCC's implies directly a solution for the general problem.

```
Annotated-Graph Allocate-Graph (directed graph $G$)

 1. For each non-trivial SCC $S \in G$, $S =$ Allocate-SCC $(S)$
 2. Following the partial-order forest, allocate values to non-SCC
    nodes and shift the allocated values of the SCCs so they satisfy
    all constraints. // If every SCC is contracted to a single node, the
                     // resulting graph is a forest
 3. Return the Annotated graph $G$.

Annotated-Graph Allocate-SCC (SCC $S$)

 1. If $S$ has a single node $x$ assign $range(x) = \{0\}$ and return $S$.
 2. Find the set of cutpoints $C$ of $S$ // See Definition 1
 3. Construct the Cutpoint-graph $S_C$ // See Definition 2
 4. Allocate-SCC $(S_C)$.
 5. For each regular node $x$:// See description of the four phases in Sec. 3.2
    (a) (Phase 1) Find the cycle-values of $x$
    (b) (Phase 2) Find the dfs-values of $x$
    (c) (Phase 3) Let $\{C_{1x}, C_{2x}, \ldots, C_{nx}\}$ be the set of cutpoints that can
        reach $x$. Then assign $range(x) = \{u + v | u \in range(C_{ix}) \wedge (v \in$
        $cycle\text{-}values^0_{C_i}(x) \cup dfs\text{-}value^0_{C_i}(x))$
    (d) (Phase 4) Add a dfs-value corresponding to the virtual level
 6. Return the annotated SCC $S$.
```

Fig. 1. Procedures `Allocate-Graph` calls `Allocate-SCC` for each SCC. Both procedures receive a directed graph as input, and annotates each node x in this graph with a set of adequate values $range(x)$.

The pseudo-code in Figure 1 shows the overall structure of our algorithm. The main procedure, `Allocate-Graph` receives a graph as input and returns a graph where all its nodes are annotated with adequate ranges. For each node x we denote the corresponding range by $range(x)$. It allocates values for SCCs with the procedure `Allocate-SCC` and then shifts them so that these ranges are valid in relation to the other SCCs present in the graph. Given an SCC S, `Allocate-SCC` calls itself recursively on a smaller SCC derived from S (the recursion stops when S is the trivial SCC comprised of a single node). At the end of the recursion, the values assigned to the smaller SCC are used to allocate values for all the nodes in SCC S. The description of `Allocate-SCC`, which requires several definitions, appears in the next subsection.

3.1 Range allocation for SCCs

The goal of `Allocate-SCC` is to annotate each node v of a given SCC G with a finite and adequate range $range(x)$ (not necessarily continues). The primary source difficulty in assigning adequate values to SCCs is the presence of cycles. We introduce the notion of *cutpoints* to deal with cycles in the SCC.

Definition 1 (Cutpoint-set). *Given a directed graph $G(V, E)$, a set of nodes $v \subseteq V$ is a* Cutpoint-set *if the subgraph of G induced by nodes in $V \setminus v$ is cycle free.*

The notion of cutpoints is also known in the literature as *Feedback Vertex Sets* [4]. Finding a minimal set of cutpoints is an $NP\text{-}Hard$ problem, so our implementation uses a polynomial approximation (several such approximations are described in the above reference). We will refer to non-cutpoint nodes simply as *regular nodes*. This distinction refers to the current recursion level only. As we will soon see, all graphs at recursion levels other than the highest one comprise only of the cutpoints of the original SCC G.

Next, we define a *collapsing* of an SCC G on to its Cutpoint-set, which we call the *Cutpoint-graph* of G:

Definition 2 (Cutpoint-graph). *Given an SCC G and a Cutpoint-set C of G, a* cutpoint graph of S with respect to C *is a directed graph $G_C(C, E)$ such that for $u, v \in C$, $u \neq v$, edge $(u, v) \in E$ with weight w iff there is a path in G from u to v not passing through any other vertex in C and with an accumulated weight w*

Note that according to this definition there are no self loops in Cutpoint-graphs. As an example, consider the graph shown in the left of Figure 2. One possible Cutpoint-set for this graph is the set $\{A, C\}$. The Cutpoint-graph over these nodes is shown on the right hand side of the figure. We will use this graph as a running example to illustrate our algorithm.

`Allocate-SCC` (Figure 1) progresses by calling itself recursively on the graph G_C. It is easy to see that G_C is also an SCC, but smaller. This ensures that `Allocate-SCC` terminates. This observation is proven in Lemma 1 below.

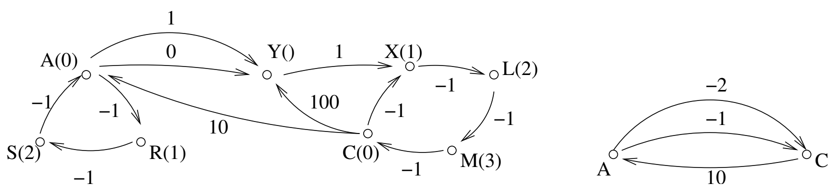


Fig. 2. An SCC G (left) and its Cutpoint-graph with respect to the Cutpoint-set {A,C}

Lemma 1. *A Cutpoint-graph G_C of an SCC G is an SCC and has less vertices than G.*

(All proofs are in Appendix A).

In case G is a single node `Allocate-SCC` assigns it the range $\{0\}$ and returns. The set of values returned from the recursive call are then used to assign values to the rest of the graph (the regular nodes at this level). The process of assigning these values is involved, and is done in four phases as described in the next subsection.

3.2 The four phases of allocating ranges to regular nodes

Ranges are assigned to regular nodes in four phases. In the first two phases the assigned ranges, called *cycle-values* and *dfs-values* respectively, should be thought of as *representative* values: they do not necessarily belong to the final ranges assigned by the algorithm. In Phase 3, these values will be combined with the ranges assigned to the cutpoints to compute the final ranges for all the regular nodes. In the description of these phases we will use the notion of *tight assignments*, defined as follows:

Definition 3 (Tight assignments). *Given a directed weighted path from node x to node y with an assignment of a single value to each node, this assignment is called* tight with respect to the value assigned to x, *if and only if all the inequalities represented by the path are satisfied and the value at each node other than x is the largest possible. In that case, the path is said to be* tight *with respect to the value at x.*

Phase 1 *The role of phase 1 is to find values that satisfy all non-positive cycles in the graph, assuming that the cutpoints in the cycles are assigned the value 0 (this assumption will be removed in phase 3).* We find all the non-positive cycles (by a non-positive cycle we mean a cycle whose edge weights add up to 0 or less). Each such cycle will have one or more cutpoints. For every path p from cutpoint C_1 to cutpoint C_2 that forms part of the cycle, we assign 0 to C_1 and *tight* values to the other nodes in p (except C_2 which could be the same as C_1) with respect to C_1's assignment. If a node x on p is assigned a value v, we say that this value is *obtained from* C_1.

At the end of this phase each node has a (possibly empty) set of values (except cutpoints which only have the value 0). For a regular node x, there is one value for every non-positive cycle that goes through it. Referring to the example in Figure 2, the sets of values at nodes R, S are $\{1\}$ and $\{2\}$ respectively. Y has an empty set of values as there are no non-positive cycles going through it. All the cycle values are shown in Figure 2 beside the nodes in round brackets. The set of values associated with each node x is called the *cycle-values* corresponding to *level* 0 and is denoted by $\textit{cycle-values}^0(x)$ (0 being the value assumed at cutpoints). Further, the subset of *cycles-values* at x obtained from C_i is denoted by $\textit{cycle-values}^0_{C_i}(x)$.

Phase 2 *The role of phase 2 is to assign values that satisfy all acyclic paths in the SCC starting from a cutpoint, assuming that the cutpoint has a value 0 (this assumption will be removed in phase 3).* In the second phase, we begin by creating a new set of values at each node called *dfs-values*. Regular nodes have one *dfs-value* corresponding to each cutpoint that can reach it *directly* (that is, without going through any other cutpoint). For a node x and a cutpoint C_i that can reach it directly, denote the *dfs-value* corresponding to C_i at level 0 by $dfs\textit{-values}^0_{C_i}(x)$. The value of $dfs\textit{-values}^0_{C_i}(x)$ is calculated as follows. Let n be the number of direct paths from C_i to x, and let $v_1, \ldots, v_n$ be values corresponding to tight assignments to x with respect to C_i. Then $dfs\textit{-values}^0_{C_i}(x) = \min_i\{v_1 \ldots v_n\}$. The implementation of Phase 2 involves a simple Depth-first search (DFS) which starts from cutpoints and backtracks on reaching any cutpoint.

Referring to Figure 2, $dfs\textit{-values}^0_A(Y) = -1$ and $dfs\textit{-values}^0_C(Y) = -100$. Thus, the *dfs-values* for Y are $\{-1, -100\}$. The other *dfs-values* are shown in Figure 3 in square brackets besides the nodes. The first value is obtained from A and the other, if present, from C.

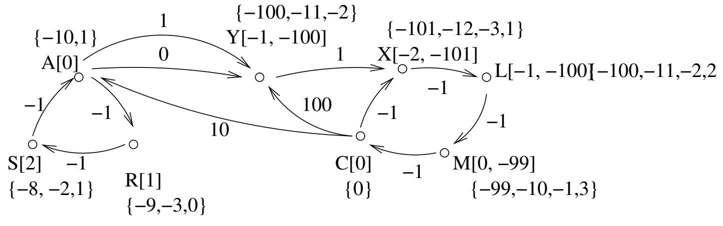


Fig. 3. The *dfs-values* of nodes appear in brackets. Both cutpoints (A and C) have a path to Y, hence the two *dfs-values*.Final ranges for all nodes are shown in curly braces

Phase 3 *The role of phase 3 is to translate the representative values computed in the first two phases into actual ranges using the ranges allocated to the cutpoints by the recursive call.* In the third phase, we use ranges assigned to cutpoints by the recursive call `Allocate-SCC` (S_C) as the base for shifting the representative values that were computed by the first two phases. The ranges assigned to the cutpoints remain unchanged (none of the three phases modify the ranges assigned to the cutpoints by deeper recursive calls).

For each regular node x we find all the cutpoints $\{C_{1x}, C_{2x}, .., C_{nx}\}$ that can reach it. Then $range(x)$ is given by

$$range(x) = \{u + v | u \in range(C_{ix}) \wedge (v \in cycle\text{-}values^0_{C_i}(x) \cup dfs\text{-}values^0_{C_i}(x))\}$$

Example 2. Consider once again the graph in Figure 2. Assume that cutpoints have already been assigned the following values: $range(A) = \{-10, -1\}$ and $range(C) = \{0\}$. For node Y, the set $cycle\text{-}values^0(Y)$ is empty and $dfs\text{-}values^0(Y) = \{-1, -100\}$. The cutpoints that can reach Y are A and C. So the range associated with Y is the union of $\{-100\}$ (the values from C) and $\{-11, -2\}$ (the values from A).

The value $u + v \in range(x)$ such that $u \in range(C_{ix})$ and v is a *dfs-value* or a *cycle-value* is said to correspond to *level* u at C_{ix}.

Phase 4 We now add one more value, called the *virtual dfs-value*, to each regular node. The need to have an additional value is as follows. Given a satisfiable subgraph $G' \subseteq G$, the values assigned in the previous phases can be used for a node x only if it can be reached from a cutpoint. For the case where x is not reachable from any cutpoint in G' we need to have an extra value.

We allocate *virtual dfs-values* by starting from the highest value at each cutpoint, and going *backward* along the edges until we reach another cutpoint. We assign tight values along each *reverse* path with respect to the starting cutpoint. At the end, for each node x we pick the maximum value among all the values assigned by different paths and make it the *virtual dfs-value*.

From now on, by *dfs-values* of a node x we will mean all the *dfs-values* corresponding to all levels and cutpoints. We use the term *cycle-values* in a similar way. Considering the graph in Figure 2, the final set of values assigned to all nodes is shown in Figure 3. The values are shown in curly braces near the nodes.

4 Correctness of the Algorithm

Assuming that `Allocate-SCC` is correct, the correctness of `Allocate-Graph` is easy to see: `Allocate-Graph` simply shifts the ranges allocated to SCCs and assigns a single value to nodes between them so that all the constraints between the SCCs are satisfied. From now on the proof will focus on proving the correctness of `Allocate-SCC`. We will prove that the procedure terminates and that the ranges it allocates are adequate.

Termination of `Allocate-SCC` is guaranteed by Lemma 1 because it implies that the number of nodes decreases as we go from G to G_C. This ensures that the size of the considered SCC decreases in successive recursive calls until it is called with a single node and returns.

We now have to show that that `Allocate-SCC` allocates adequate ranges. Assume that G is an SCC and `Allocate-SCC` used a set of cutpoints C in allocating ranges to G. Given a satisfiable subgraph $G' \subset G$, we describe an *assignment procedure* that, assigns values to its nodes from the ranges allocated to it by `Allocate-SCC`, which satisfy all of the constraints represented by G'. The assignment procedure and the proof of correctness explained using the *augmented* graph of G', denoted by G'_{Aug}.

Building the augmented graph We construct the augmented graph as follows. Find all paths P in G' starting from some cutpoint C_i and ending at a regular node x such that:

- The path P does not occur as part of any non-positive cycle in G
- If nodes in P are given corresponding $dfs\text{-}values^0_{C_i}$ values then it is not tight with respect to value 0 at cutpoint C_i.

For each such path we add an edge from C_i to x with a weight equal to $-(dfs\text{-}values^0_{C_i}(x))$ (the negation of the level 0 *dfs-value* of x corresponding to C_i). Note that such an edge need not be a part of the graph G. By the following lemma, if G' is satisfiable then the augmented graph G'_{Aug} is satisfiable as well.

Lemma 2. *If G' is a satisfiable subgraph of G then the graph G'_{Aug} as constructed above is satisfiable as well. Further, any assignment that satisfies G'_{Aug} satisfies G'.*

Example 3. Suppose we are given a graph G' as shown in Figure 4 (refer only to the solid edges). G' is a satisfiable subgraph of the graph shown in Figure 2. The *dfs-value* of X corresponding to level 0 and cutpoint A is -2. In G', the path $A \rightarrow Y \rightarrow X$, with nodes having values 0, -1, -2 respectively (shown in square brackets beside the nodes), is not tight. So we augment G' by adding two edges: one from A to Y with weight 1 and one from A to X with weight 2, both depicted as dotted edges in the graph.

4.1 The Assignment Procedure

The assignment procedure assigns values to G'_{Aug} that satisfy its constraints. By Lemma 2 these values satisfy G' as well. The assignment procedure is recursive: we first handle the graph $G'_{\{C',Aug\}}$, the cutpoint graph of G'_{Aug}, where C' is a set of cutpoints of G'_{Aug} and $C' \subseteq C$. The base case of the recursion corresponds to a graph with a single node to which we assign the value 0. Assume that we have already assigned values, recursively, to nodes in C'. Now we assign values to all the regular nodes in G'_{Aug}, by starting at cutpoints (with the values assigned to them by the deeper recursion calls) and doing a DFS-like search. The pseudocode for this procedure is shown below in Figure 5

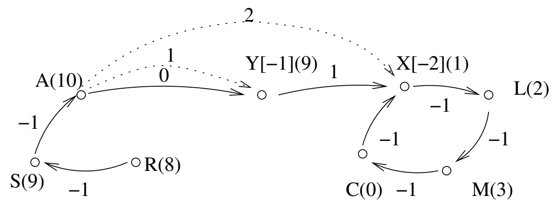


Fig. 4. The graph G' (solid edges), and its augmentation (adding the dotted edges) G'_{Aug}. The augmented graph can be assigned tightly by its *dfs-values*. The values assigned by the assignment procedure are shown in round brackets.

```
1. For each regular node $x$ find all cutpoints $\{C_{1x} \dots C_{mx}\}$ in $G'_{Aug}$ that
   can reach it.
2. For each cutpoint $C_{ix}$ find all direct paths $P$ to $x$, and for each
   such path find tight values assuming the value of cutpoint $C_{ix}$ is as
   assigned by the previous recursive call.
3. Find the minimum among all these tight values corresponding to all
   cutpoints in $\{C_{1x} \dots C_{mx}\}$ and assign it to $x$.
4. For nodes that cannot be reached from any point in $C'$ assign them
   their *virtual dfs-values* (see end of Section 3.2).
```

Fig. 5. Pseudo-code for `DFS-Assign`, the assignment procedure

Referring to Figure 4, the values assigned by the `DFS-Assign` are shown in round brackets beside each node. At the end of this procedure each node in G' is assigned a single value such that all constraints in G' are satisfied (yet to be proved)).

Then we have the following lemma:

Lemma 3. *Assuming that the assignment procedure assigns satisfying values to a cutpoint graph $G'_{\{C',Aug\}}$ from the appropriate ranges, it does so for G'_{Aug} as well from the ranges allocated by* `Allocate-SCC`*.*

4.2 The ranges allocated by `Allocate-SCC` are adequate

Our goal is to prove that `Allocate-SCC` assigns adequate ranges to the nodes of any given SCC G. We will use the following lemma

Lemma 4. *Assuming* `Allocate-SCC` *assigns adequate ranges to G_C, it assigns adequate ranges to all nodes of G.*

This lemma follows directly from Lemma 3. Now we the main theorem.

Theorem 1. `Allocate-SCC` *assigns adequate ranges to nodes of any given SCC G*

Proof. The termination of `Allocate-SCC` follows from lemma 1. We prove the correctness of `Allocate-SCC` by induction on the number of cutpoints present in the given SCC. The base case is an SCC with one node for which the theorem holds trivially. For the inductive step, it is shown in Lemma 4 that if we can assign adequate ranges to the Cutpoint-graph G_C then the four phases assign adequate ranges to all nodes of the SCC. Thus it follows that for any SCC G `Allocate-SCC` assign adequate values to the nodes.

5 Experimental Results

We now present results of running `Allocate-Graph` on different benchmarks. The results are summarized in the table below.

Example	SMOD	UCLID	SEP	Example	SMOD	UCLID	SEP
bf12.ucl	12	101	28	code27s.smv	85	104	94
bf13.ucl	15	170	48	code32s.smv	114	109	120
bf14.ucl	21	158	33	code37s.smv	57	71	90
bf6.ucl	105	481	127	code38s.smv	32	34	106
bf17.ucl	176	1714	482	code43s.smv	361	555	424
bf18.ucl	280	2102	603	code44s.smv	69	140	84
BurchDill	250	291	336	code46s.smv	264	287	225

The examples beginning with *bf* were derived from software verification problems. The examples beginning with *code* have been used in the paper [5].

We compare our approach against two other methods. The first method, named `UCLID`, is the standard implementation from UCLID [3]. This approach is discussed towards the end of Section 1. The other method, named `SEP` was presented in [8] and is discussed in Section 1. The table shows the number of Boolean variables that are required to encode the ranges assigned by the three different algorithms. The encoding is straight-forward. Given a range of values having n values the number of boolean variables required to encode them is equal to the smallest k such that $2^k \geq n$. So the number of Boolean variables needed to encode the ranges is a fair estimate of the logarithm of the state space that needs to be explored.

As we can see from the results above, our method is clearly superior to the other two methods. We outperform `UCLID` by nearly 10 times on large examples(bf17.ucl and bf18.ucl). Compared to `SEP` we outperform it on the big examples by a factor of nearly 3.

On graphs which are densely connected and have small edge weights our algorithm does not do as well as `UCLID`. For such graphs, the ideal ranges seem to be dependent on the number of variables and relatively independent of the edges. Our algorithm on the other hand is heavily dependent on the edge structure in determining the values and as the edges to nodes ratio increases, the number of values assigned by our algorithm tends to increase as well. Hence on dense graphs, our algorithm ends up assigning too many values.

Run time complexity of `Allocate-SCC` The worst case complexity of `Allocate-SCC` is exponential in the size of the given SCC. This is because we have to find all the non-positive cycles in Phase 1. Phase 2 could be exponential in the worst case as well. But, in practice our algorithm is quite fast on most examples. On all the above examples our algorithm finishes well within 10s. `Allocate-SCC` runs slowly on examples which have too many non-positive cycles since it has to traverse every such cycle.

6 Conclusion and Future Work

We have presented a novel technique for allocating small adequate ranges for variables in a Separation Logic formula based on analysing the corresponding *inequalities graph*. The state space spawned by these small ranges can be then effectively explored by standard SAT or BDD solvers. Experimental results show that our decision procedure can lead to exponential reduction in the state space to be explored.

We have developed a number of optimizations for the approach presented in this paper, which make the decision procedure faster and reduce the ranges further. Due to lack of space, we are not able to present them here.

References

1. W. Ackermann. *Solvable cases of the Decision Problem.* Studies in Logic and the Foundations of Mathematics. North-Holland, Amsterdam, 1954.
2. G. Audemard, P. Bertoli, A. Cimatti, A. Kornilowicz, and R. Sebastiani. A SAT based approach for solving formulas over boolean and linear mathematical propositions. In *Proc. 18th International Conference on Automated Deduction (CADE'02)*, 2002.
3. 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 E. Brinksma and K.G. Larsen, editors, *Proc.* 14^{th} *Intl. Conference on Computer Aided Verification (CAV'02)*, volume 2404 of *LNCS*, pages 78–91, Copenhagen, Denmark, July 2002. Springer-Verlag.
4. Dorit S. Hochbaum, editor. *approximation-algorithms for NP-hard problems.* PWS Publishing Company, 1997.
5. A. Pnueli, Y. Rodeh, O. Strichman, and M. Siegel. The small model property: How small can it be? *Information and computation*, 178(1):279–293, October 2002.
6. V. Pratt. Two easy theories whose combination is hard. Technical report, Massachusetts Institute of Technology, 1977. Cambridge, Mass.
7. R. Shostak. Deciding combinations of theories. *J. ACM*, 31(1):1–12, 1984.
8. O. Strichman, S.A. Seshia, and R.E. Bryant. Deciding separation formulas with SAT. In E. Brinksma and K.G. Larsen, editors, *Proc.* 14^{th} *Intl. Conference on Computer Aided Verification (CAV'02)*, volume 2404 of *LNCS*, pages 209–222, Copenhagen, Denmark, July 2002. Springer-Verlag.
9. A. Stump, C. Barrett, and D. Dill. CVC: a cooperating validity checker. In *Proc.* 14^{th} *Intl. Conference on Computer Aided Verification (CAV'02)*, 2002.

A Proof Supplement

This appendix contains the proofs of some of the lemmas used in the paper.
Lemma 1. *A Cutpoint-graph G_C of an SCC G is an SCC and has less vertices than G.*

Proof. For every SCC G we can take a (not-necessarily minimal) Cutpoint-set comprising of $|G| - 1$ points, since clearly the remaining point is cycle free (we can assume that there are no self loops in G because we can always replace predicates of the form $x \geq x + c$ with FALSE or TRUE , depending on the sign of c). Thus, there exists a Cutpoint-set such that $|G_C| < |G|$. We now have to prove that G_C is an GCC. Falsely assume that there exists two nodes $u, v \in G_C$ such that there is no path between them in G_C. Consider C nodes on the path from u to v in G. If there are no such points then we are done because by definition there has to be an edge in G_C from u to v. Otherwise, let u' be the closest such node to u. By definition of Cutpoint-graphs there is an edge in G_C from u to u'. Now repeat the argument for the path from u' to v. Eventually this will create a path from u to v in G_C – contradiction. □

Lemma 2. *If G' is a satisfiable subgraph of G then the graph G'_{Aug} as constructed above is satisfiable as well. Further, any assignment that satisfies G'_{Aug} satisfies G'.*

Proof. The augmenting edges are not part of a cycle, since we do not add them along non-positive cycles and there are no positive cycles in G' (since G' is assumed to be satisfiable). Thus, it cannot make a subgraph unsatisfiable. An assignment that satisfies G'_{Aug} satisfies G' as well since the former is equally or more constrained than the latter with respect to any path in the graph.

Lemma 5. *If a satisfiable subgraph E of G consists of single cutpoint C_1 and m non-positive cycles $L_1, L_2, \ldots L_m$ passing through it, then fixing the value of C_1 at 0 we can find satisfying values for all other nodes from their respective* cycle-values *corresponding to level 0 and cutpoint C_1.*

Lemma 5. *If a satisfiable subgraph E of G consists of single cutpoint C_1 and m non-positive cycles $L_1, L_2, \ldots L_m$ passing through it, then fixing the value of C_1 at 0 we can find satisfying values for all other nodes from their respective* cycle-values *corresponding to level 0 and cutpoint C_1.*

Proof. To prove this lemma we will show that for every node there is a value from the corresponding set of *cycle-values* corresponding to level 0. We will consider a node x and argue that no matter how many cycles pass through it we can always find a satisfying value for it. If there is only one cycle passing through x then we pick the *cycle-value* determined by that cycle. The case where there are two or more cycles are all handled in a similar manner. We will present the argument for the case of two cycles, this can be extended simply to higher number of cycles.

As discussed before, each cycle constrains the values of the nodes to belong to certain intervals (assuming that value of the cutpoint is fixed). Suppose node

x belongs to two cycles L_i and L_j. Let the intervals to which they constrain x be $[l_i, u_i]$ and $[l_j, u_j]$. Assume without loss of generality, $u_i < u_j$ (if we $u_i = u_j$ we have nothing to prove as we can pick that value). Now the intersection of the two intervals must be non-empty, otherwise the subgraph E will not be satisfiable. Thus $l_j \leq u_i < u_j$ and picking u_i will satisfy both the cycles.

Hence we can assign values to all nodes from their corresponding sets of *cycle-values* corresponding to level 0. □

Lemma 3. *Assuming that the assignment procedure assigns satisfying values to a cutpoint graph $G'_{\{C',Aug\}}$ from the appropriate ranges, it does so for G'_{Aug} as well from the ranges allocated by* `Allocate-SCC`*.*

Proof. We will need to define several terms for the following discussion. We first introduce the concept of *bound-paths.* Given an SCC G over a set of nodes V, a cutpoint set C and a satisfiable subgraph G' of G, a path P in G' is called a *bound-path* if no cutpoint in G' can reach its head, and its tail cannot reach any cutpoint in G'. More formally, one of the following four cases holds:

1. P starts at a cutpoint C_1 and reaches another cutpoint C_2
2. P starts at a cutpoint C_1 and no extension of it can reach a cutpoint
3. P starts at a regular node s, which cannot be reached by any cutpoint in G' and it reaches a cutpoint C_2.
4. P starts at a regular node s, which cannot be reached by any cutpoint in G' and no extension of it can reach a cutpoint

Based on this notion now define *Tight bound-paths*:

Definition 4 (Tight bound-path). *Let G, $C \subset V$ be defined as before and G' is a satisfiable subgraph that has already been given values by the assignment procedure. A* bound-path *P in G' is said to be* tight *if the values of all nodes except the last one along the path are tight with respect to the starting point of the path. The value of the last node is also required to be tight if it is not a cutpoint.*

It is easy to see that after the assignment procedure each node in G'_{Aug} belongs to some tight bound-path.

The proof proceeds as follows: we will divide up the subgraph G'_{Aug} into *bound-paths* and show that all inequalities represented by each *bound-path* P are satisfied, and that values assigned to each node in the path indeed belong to the corresponding ranges. Then it clearly follows that all the inequalities represented by G' are satisfied. Consider any *bound-path* P:

- Case 1: P is a path from cutpoint C_1 to cutpoint C_2.
  1. If the values assigned by the `DFS-Assign` procedure are tight, then clearly the values satisfy the inequalities represented by the path. To prove that the values are indeed in the ranges, we again recognize two cases:
     - Case A: the path P occurs as a part of a non-positive cycle in G,Then clearly the set of *cycle-values* associated with each node in P contain the assigned values.

        - Case B: the path P does not occur wholly as part any non-positive cycle in G, then since we consider G'_{Aug} all values assigned to the nodes must be the *dfs-values* corresponding to C_1.

      Thus in either case, the values belong to the appropriate ranges.
    2. In case the values assigned to nodes in path P are not tight, then the path can be split into parts such that each part belongs to some tight path (recall the claim made after Definition 4 that after the assignment procedure, each node in G'_{Aug} belongs to some tight bound-path). Then by reasoning similar to case B above the values must all belong to appropriate ranges. That they also satisfy all the inequalities follows from the way `DFS-Assign` is done
- Case 2: P is a cycle involving a single cutpoint C_1. Then the cycle must be a non-positive cycle. We have two cases:
  - If the values of nodes in P are tight then they belong to the set of *cycle-values* (at level t, where t is the value of the cutpoint) and they are *satisfying*.
  - If not, then some of the values belong to some other *tight-paths*.
    1. One possibility is that all these other *tight-paths* begin at C_1 itself, then by Lemma 5, the values assigned to the nodes in path P satisfy the inequalities and they also belong to the allocated ranges (to the corresponding *cycle-value* sets to be precise).
    2. Another possibility is that some of the nodes belong to a *tight-path* P_2 starting at a cutpoint C_2. Then, by previous cases, there is a path from C_2 to C_1 which overlaps with the path P. Say this overlap starts at node x, then the sub-path from x to C_1 is satisfied by the values assigned and the values are also in the appropriate ranges. Consider the part of path P starting at C_1 and ending at the node prior to x, say y. Call this part P'. The edge (x, y) must be satisfied by the values assigned by the procedure `DFS-Assign` by definition of `DFS-Assign`. Consider P'. It can be treated precisely the way we treated P, i.e., there are the following options again:
       * Given the values assigned by the assignment procedure, the path P' is a tight path. Then the clearly the values are satisfying and they belong to the appropriate *cycle-values* sets.
       * In case P' is not a tight path. Then it must be composed of sections that belong to tight paths. And the proof is similar to Case 2 within Case 1 above.

       We can thus conclude that all the values in P are satisfying and they belong to the appropriate ranges.
- Case 3: the path P starts from a cutpoint C_i and no extension of it leads to another cutpoint. In this case, we can easily see that the values assigned by `DFS-Assign` are satisfying and that the values belong to appropriate ranges also follows easily.
- Case 4: P is a *bound-path* starting at a regular node x which cannot be be reached by any cutpoint in the subgraph G'. The path may or may not end in a cutpoint. For this path the assignment procedure uses the *dfs-values* for the

nodes in the cutpoint corresponding to the *virtual level*. The condition that the values must belong to the appropriate ranges is trivially satisfied. The fact that these values satisfy the inequalities represented by P is proved by contradiction. Suppose that there is an edge $(u, v) \in P$ with weight w that is not satisfied. The values at u, say $val(u)$ is such tha $val(u)+w > val(v)$. But while finding *virtual values* we must have traversed this edge and assigned a value of $val(u) - w$ at v, but then while picking the maximum of all values at v as the *virtual-value* we picked a lower. This is not possible because our procedure picks the higest value available at node. Thus there can be no edge (u, v) that is not satisfied in P.

$\square$