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 some ordered infinite type like `real` or `integer`. Any equality or inequality can be expressed in this logic. We propose a decision procedure for Separation Logic based on allocating small domains (ranges) to the formula's variables that are sufficient for preserving satisfiability. Given a Separation Logic formula φ, our procedure constructs the *inequalities graph* of φ, based on φ's predicates. This graph represents an abstraction of the formula, as there are many formulas with the same set of predicates. Our procedure then analyzes this graph and allocates a range to each variable that is adequate for all of these formulas. This approach of finding small finite ranges and enumerating them symbolically is both theoretically and empirically more efficient than methods based on case-splitting or reduction to Propositional Logic. 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

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 some ordered infinite type like `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 [2]. In this paper, we consider Separation Logic with `integer` variables and constants only, assuming that only minor adaptations are needed for other cases. 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 [7], 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 [8]), an efficient decision procedure for Separation Logic will be helpful in verification of any formula that contains a significant number of these predicates.

There are various known methods for solving Separation Logic (see tools such as CVC [11] and MATHSAT [3] that solves this logic), which we survey in detail in the full version of this article [12]. Let us just mention here the most recent one, which is based on a reduction to Propositional Logic. In [10, 9] such a reduction is proposed, 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. It encodes each predicate of the form $x \geq y + c$ with a new Boolean variable e^c_{xy}, and then gradually removes nodes from the graph while adding transitivity constraints (similar to the Fourier-Motzkin technique). The following example illustrates this method. Consider the formula $\varphi : x \geq y + 1 \wedge (y \geq z + 2 \vee z \geq x + 1)$. As a first step, abstract this formula with Boolean constraints, i.e. $\varphi' : e^1_{xy} \wedge (e^2_{yz} \vee e^1_{zx})$.

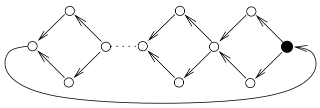
Next, nodes (variables) are eliminated one at a time while adding proper constraints to the formula. Given the order x, y, z it first eliminates x, deriving from the first and third predicates the new constraint $z - 1 \geq y + 1$. It consequently adds the constraint $e^1_{xy} \wedge e^1_{zx} \rightarrow e^2_{zy}$. Eliminating y it derives from the second and fourth constraint the new unsatisfiable constraint $z - 2 \geq z + 2$ and adds, accordingly, the constraint $e^2_{yz} \wedge e^2_{zy} \rightarrow \text{FALSE}$. This procedure may result in an exponential number of constraints and a quadratic number of variables. In contrast, the Range-Allocation approach which we present now, requires in the worst case $n \cdot \log n$ variables and no additional constraints.

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 bound on the size of the smallest satisfying model. This implies that the given formula can be decided by checking all possible valuations up to that bound.

In the case of predicates of the form $x > y$ (since we treat integers and weak inequalities this is the same as $x \geq y + 1$) 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. Informally, the reason that this range is sufficient is that every satisfying assignment imposes a partial order on the variables, and every assignment that preserves this order satisfies the formula as well. Since all orderings are possible in the range $[1 \ldots n]$, this range is sufficient, or, as we later call it, it is *adequate*. In the case of full Separation Logic (when there are arbitrary constants), it was shown [4] that in the worst case 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 $(n + maxC)^n$. These results refer to a *uniform* range allocation to all variables, regardless of the formula structure. In this article we investigate methods for reducing this number, based on an analysis of the formula's structure, which typically results in *non-uniform* range allocation.

A similar approach was taken in the past by Pnueli et al. [6] 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 uniform range allocation results in a state-space of size $O(n^n)$. In contrast, our approach allocates a single constant to one node, and 2 values to all the rest, which results in a state-space of size $O(2^n)$. If the graph has arbitrary edge weights, the state-space resulting from a uniform range allocation can grow up to $O((n+maxC)^n)$, while in our approach it would remain $O(2^n)$ (it will, however, increase the values themselves, which has a relatively minor effect on performance).



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 $Vars(\varphi) = \{v_1, \dots v_n\}$ 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 $SAT_R(\varphi)$ denote that φ is *satisfiable* in a domain R. Our goal is the following:

Find a small domain R such that

$$SAT_R(\varphi) \iff SAT_{\mathbb{Z}}(\varphi) \tag{1}$$

We say that a domain R is *adequate* for φ if it satisfies formula (1). It is straightforward to see that if $SAT_{\mathbb{Z}}(\varphi)$ holds then the minimal adequate domain has size 1: simply assign the variables in $Vars(\varphi)$ a constant value according to one of the satisfying assignment. Thus, finding the smallest domain for a given formula is at least as hard as checking the satisfiability 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 still too computationally expensive (it is still exponential, although we will not prove it here). We will therefore concentrate on finding over-approximations which are easier to compute. As was previously indicated, our solution is based on a graph-based analysis of the formula's structure.

Input formula We define a normal form for Separation logic as follows: 1) 'greater-equal than' ($\geq$) is the only allowed predicate, and 2) there are no negations in the formula. Every Separation logic formula can be transformed to this form by first translating it to NNF, reversing negated inequalities, reversing 'less than' ($<, \leq$) predicates and finally, replacing strong inequalities of the form $x > y + c$ with weak inequalities of the form $x \geq y + c + 1$.

A graph representation Given a Separation Logic formula φ, we construct the *inequalities graph* $G_\varphi(V, E)$ as follows (we will write G_φ from now on).

Definition 1 (Inequalities graph). *The* inequalities graph$G_\varphi(V, E)$ *is constructed as follows: 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.*

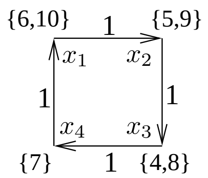
We will say that a domain is *adequate for a graph G_φ* if it is adequate for $\Phi(\varphi)$.

Definition 2 (Consistent subgraphs). *A subgraph of an inequalities graph 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 (predicates) that can be satisfied simultaneously. Restating in graph-theoretic terms, our goal is:

> Given G_φ, find a domain R for each node in V such that every consistent subgraph 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. $\square$

It is obvious that we should find small ranges with an overhead smaller than what is saved compared to the other methods mentioned earlier. In this respect we were not entirely successful, as in the worst case our algorithm is exponential (it requires at some point to look for all negative cycles in the graph). In practice, however, we did not encounter an example that takes a long time to allocate ranges for. It is left for future research to find polynomial approximations to our algorithm.

3 The 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 subgraphs 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 SCC. Thus, a solution to this problem for SCCs implies directly a solution for the general problem.

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 $R(x)$. 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). When returning from the recursive call, 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 x of a given SCC S with a finite and adequate range $R(x)$ (not necessarily continuous). The primary source of 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 3 (Cutpoint-set). *Given a directed graph $G(V, E)$, a set of nodes $v \subseteq V$ is a* Cutpoint-set *if removing them and their adjacent nodes makes the graph cycle free.*

The notion of cutpoints is also known in the literature as *Feedback Vertex Sets* [5]. Finding a minimal set of cutpoints is an *NP-$Hard$* problem, so our implementation uses a polynomial approximation (several such approximations are

`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 predicates.` // 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` $R(x) = \{0\}$ `and return` S.
2. `Find the set of cutpoints` C `of` S // See Definition 3
3. `Construct the` *`Cutpoint-graph`* S_C // See Definition 4
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`
   $$R(x) = \bigcup_{C_{ix}} (\{u{+}v | u \in R(C_{ix}) \wedge (v \in (\mathit{cycle\text{-}values}^0_{C_{ix}}(x) \cup \mathit{dfs\text{-}values}^0_{C_{ix}}(x)))\})$$
   (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 $R(x)$.

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 S.

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

Definition 4 (Cutpoint-graph). *Given an SCC S and a Cutpoint-set C of S, a* cutpoint graph of S with respect to C *is a directed graph $S_C(C, E)$ such that for $u, v \in C$, $u \neq v$, edge $(u, v) \in E$ with weight w if and only if there is a path in S 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.

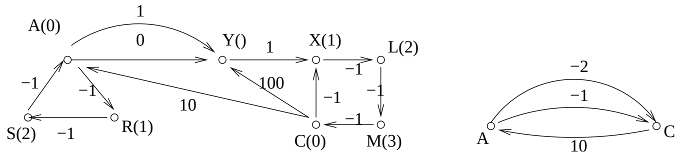


Fig. 2. An SCC S (left) and its Cutpoint-graph with respect to the Cutpoint-set {A,C}

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

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

(All proofs appear in the full version of this article [12]).

In case S 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 5 (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, each of which, by definition, has one or more cutpoints. For every path p on that cycle from cutpoint C_i to cutpoint C_j, we then assign 0 to C_i and *tight* values with respect to C_i's assignment, to the other nodes in p except C_j. Typically there is only one cutpoint in the cycle, which can be thought of as a particular case of the above one where $C_2 = C_1$. If in this step a node x on p is assigned a value v, we say that this value is *obtained from* C_i.

At the end of this phase regular nodes have one value for each non-positive cycle that goes through them, while cutpoints have the single value 0. In Figure 2 these values are shown in parenthesis. Note that Y has an empty set of values as there are no non-positive cycles going through it.

The set of values associated with each node x is called the *cycle-values* corresponding to *level* 0 and is denoted by *cycle-values*$^0(x)$ (0 being the value assumed at cutpoints). Further, the subset of *cycle-values* at x obtained from C_i is denoted by *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 (i.e. not through other cutpoints), denote the *dfs-value* corresponding to C_i at level 0 by $dfs\text{-}values^0_{C_i}(x)$. The value of $dfs\text{-}values^0_{C_i}(x)$ 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 $dfs\text{-}values^0_{C_i}(x) = \min\{v_1 \ldots v_n\}$. 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, $dfs\text{-}values^0_A(Y) = -1$ and $dfs\text{-}values^0_C(Y) = -100$. Thus, the *dfs-values* for Y are $[-1, -100]$. The other *dfs-values* are shown in Figure 3 in square brackets. The first value is obtained from A and the other, if present, from C.

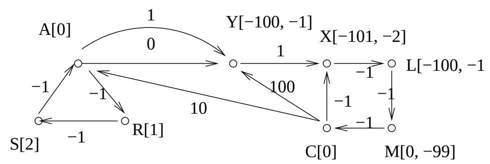


Fig. 3. The *dfs-values* of nodes appear in square brackets. Both cutpoints (A and C) have a path to Y, hence the two *dfs-values*. The final allocated ranges for all nodes are shown in curly brackets

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 not through other cutpoints. Then $R(x)$ is given by

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

Example 2. Consider once again the graph in Figure 2. Assume that cutpoints have already been assigned the following values: $R(A) = \{-10, -1\}$ and $R(C) = \{0\}$. For node Y, the set $\textit{cycle-values}^0(Y)$ is empty and $\textit{dfs-values}^0(Y) = \{-1, -100\}$. 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 $u+v \in R(x)$ such that $u \in R(C_{ix})$ 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 explained as follows. Given a satisfiable subgraph $S' \subseteq G$, 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 S' 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 4. Underlined numbers are those that were allocated in Phase 4.

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 inequality constraints between the SCCs are satisfied. From now on we will focus on proving

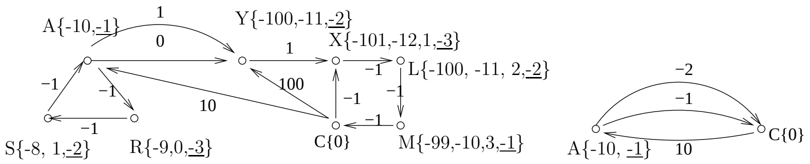


Fig. 4. The final allocated ranges for all nodes are shown in curly brackets. The right figure shows the ranges allocated for the Cutpoint-Graph, which are preserved in higher recursion levels, as can be seen on the left graph. Underlined numbers are allocated in Phase 4.

the correctness of *Allocate-SCC* (the proofs of most of the lemmas in this section appear in the full version of this article [12], and are omitted here due to lack of space). 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 S to S_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 *Allocate-SCC* allocates adequate ranges. Assume that S is an SCC and *Allocate-SCC* uses a set of cutpoints C in allocating ranges to S. Given a satisfiable subgraph $S' \subset S$, 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 predicates represented by S'. The assignment procedure and the proof of correctness are explained using an *augmented* graph of S', denoted by S'_{Aug}. Essentially the augmented graph reflects the tightest constraints in the original graph between nodes in S'. Clearly if we satisfy tighter constraints than we can satisfy the original set of constraints in S'.

Building the augmented graph We construct the augmented graph as follows. Find all paths P in S' 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 S
– 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 S' is satisfiable then the augmented graph S'_{Aug} is satisfiable as well.

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

Example 3. Suppose we are given a graph S' as shown in Figure 5 (refer only to the solid edges). S' is a satisfiable subgraph of the graph shown in Figure 2. The *dfs-value* of X and Y corresponding to level 0 and cutpoint A are -1 and -2 respectively (shown in square brackets). In S', the path $A \to Y \to X$ is not tight with respect to these values. We therefore augment S' 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. These edges reflect the tightest constraints between A, Y and X.

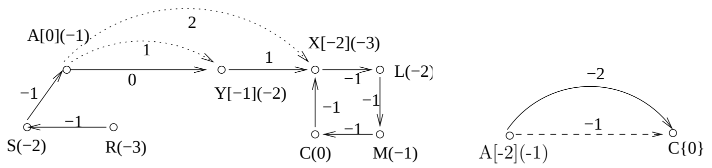


Fig. 5. The graph S' (solid edges), and its augmentation (adding the dotted edges) S'_{Aug} for the first and second recursion levels. The augmented graph can be assigned tight *dfs-values*. The values assigned by the assignment procedure (see Section 4.1) are shown in parenthesis.

4.1 The Assignment Procedure

The assignment procedure assigns values to S'_{Aug} that satisfy its predicates. By Lemma 2 these values satisfy S' as well. The assignment procedure is recursive: we first handle the graph $S'_{\{C',Aug\}}$, the cutpoint graph of S'_{Aug}, where C' is a set of cutpoints of S'_{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 S'_{Aug}, by starting at cutpoints (with the values assigned to them by the deeper recursion calls) and doing a DFS-like search. This procedure is shown below in Figure 6

Referring to Figure 5, the values assigned by `DFS-Assign` are shown in parenthesis beside each node. At the end of this procedure each node in S' is assigned a single value such that all predicates in S' 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 $S'_{\{C',Aug\}}$ from the appropriate ranges, it does so for S'_{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 S. We will use the following lemma

```
Input: Augmented satisfiable subgraph S'_Aug
Output: A satisfying assignment to S'_Aug from the ranges allocated by
Allocate-SCC.
```

1. For each regular node x find all cutpoints $\{C_{1x} \dots C_{mx}\}$ in S'_{Aug} that can reach it directly, not through any other cutpoint.
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. 6. The assignment procedure `DFS-Assign`, which demonstrates how to assign values to a given satisfiable subgraph S' from the ranges allocated by *Allocate-SCC*.

Lemma 4. *Assuming Allocate-SCC assigns adequate ranges to S_C, it assigns adequate ranges to all nodes of S.*

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

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

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 S_C then the four phases assign adequate ranges to all nodes of the SCC. Thus it follows that for any SCC S *Allocate-SCC* assigns 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 [6].

We compare our approach against two other methods. The first method, named UCLID, is the standard implementation from UCLID [4], which is based on giving a full range of $1 \ldots n + maxC$, as described in the Introduction ($maxC$ being the sum of all constants). The other method, named SEP was presented in [10] and is discussed in the Introduction as well. The table shows the number of Boolean variables that are required to encode the ranges assigned by the three different algorithms (given a set of n values the number of Boolean variables required to encode them is logarithmic in n).

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.

6 Conclusion and Future Work

We have presented a technique for allocating small adequate ranges for variables in a Separation Logic formula based on analyzing the corresponding *inequalities graph*. The state-space spawned by these small ranges can be then 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. A number of optimizations for making the decision procedure faster and reduce the ranges further were not presented here due to lack of space. The tool that we have developed, SMOD, is available for research purposes from [1].

As future work, the most important question still needs to be answered is whether it is possible to find a polynomial algorithm for range allocation. Although we did not experience long run-times in all the experiments that we conducted, this can become a bottleneck in the presence of many non-positive cycles in the inequalities graph.

Acknowledgment We would like to thank Shuvendu Lahiri for helping us by integrating our tool into UCLID and much more.

References

1. www.cs.cmu.edu/~nishants/smod.tar.gz.

2. W. Ackermann. *Solvable cases of the Decision Problem*. Studies in Logic and the Foundations of Mathematics. North-Holland, Amsterdam, 1954.
3. 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.
4. 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.
5. D. S. Hochbaum, editor. *approximation-algorithms for NP-hard problems*. PWS Publishing Company, 1997.
6. 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.
7. V. Pratt. Two easy theories whose combination is hard. Technical report, Massachusetts Institute of Technology, 1977. Cambridge, Mass.
8. R. Shostak. Deciding combinations of theories. *J. ACM*, 31(1):1–12, 1984.
9. O. Strichman. On solving Presburger and linear arithmetic with SAT. In *Formal Methods in Computer-Aided Design (FMCAD 2002)*, pages 160 – 170, Portland, Oregon, Nov 2002.
10. 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.
11. 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.
12. M. Talupur, N. Sinha, O. Strichman, and A. Pnueli. Range allocation for separation logic (Full version). Technical Report TR-04-iem/ise-1, Technion, Industrial Engineering and Management, 2004.