Trace Partitioning as an Optimization Problem

Charles Babu M[1,2][0009−0000−8598−2485], Matthieu Lemerre[1][0000−0002−1081−0467], Sébastien Bardin[1][0000−0002−6509−3506], and Jean-Yves Marion[2][0009−0002−8262−3887]

[1] Université Paris-Saclay, CEA, List, France
{matthieu.lemerre, sebastien.bardin}@cea.fr
[2] University of Lorraine, LORIA, France
{charles-babu.mamidisetti, jean-yves.marion}@loria.fr

Abstract. Imprecision is a very common phenomenon in static analyses that results in false alarms when used for program verification. Designing automatic techniques to improve static analysis precision is an old dream, but it is highly non-trivial. In the last two decades, static analysis gave rise to refinement techniques to improve precision through various forms of sensitivity. Yet, prior attempts are either specialized to particular domains or based on syntactic rules and heuristics that are tedious to design and prone to path explosion. In this paper, we cast the problem of improving static analysis precision as an optimization problem and propose a generic search-based method to solve it. We identify the challenges that one faces when solving this problem (like the large search space, path explosion, redundant computations, or non-monotonic operations in abstract domains) and provide adequate solutions to each. Finally, we provide a first implementation of the method, demonstrating both its feasibility and potential over standard benchmark (our early prototype is able to prove some goals that state-of-the-art software model checkers cannot), and providing valuable feedback when implementing this method in a static analyzer.

1 Introduction

Context. Automatic verification techniques examine the source code of a program to prove that some properties hold. Because for most classes of properties, this verification is undecidable, the program behavior is approximated, using notably the theory of abstract interpretation [20,19]. As a result, automated verification techniques are in general *sound*, i.e. the programs proved safe are

actually safe, but *incomplete*, i.e., there may be false alarms on safe programs due to imprecisions.

Problem. False alarms due to imprecision are inherent to any sound tool verifying undecidable program properties, and it is the main factor hindering the practical adoption of software verification. *Completeness* [24,14] represents the ideal situation where no false alarm is produced, but it rarely occurs in practice. As a consequence, in the last two decades, *automatic* techniques marked the trend toward automating the removal of false alarms, reducing the need for manual inspection. Model-checking [17] pioneered this trend through the Counter Example-Guided Abstraction Refinement (CEGAR) principle. In the static analysis world [20], although the problem of abstraction refinement to improve precision was studied by Giacobazzi et al. [25] from a theoretical point of view, the approach did not provide any actionable refinement techniques that could be applied to any program and abstract domains.

Generic refinement techniques to automatically improve precision are highly desirable, as they can be adapted to a wide range of abstractions. Despite its success, CEGAR is not generic and should be manually adapted for each abstract domain [6,11,30]. In contrast, static analysis gave rise to generic techniques to improve precision based on various forms of *sensitivity* frameworks [37], having in common to introduce disjunctions into abstract states in a structured way. We will coin any such a framework as *semantic-directed* [39] if it has a general strategy to choose which disjunctions to preserve, based on the results of the analysis. Among various sensitivities, the most general one is *trace partitioning* [42,29,46], which exploits the execution traces of a program to partition its abstract states and improve precision by effectively *delaying control-flow joins*. Some well-known instances of trace partitioning include loop-unrolling, context-sensitivity, etc. However, implementations of trace partitioning framework are generally not semantic-directed [39]; instead, they use heuristics which can fail to separate traces that would improve precision, or introduce useless separations. The latter is unfortunate, as too many trace splits lead to a combinatorial explosion in the size of the program. Some automated sensitivity-based semantic-directed refinement techniques do exist in static analysis, but they are specialized to some particular application domains [39,40].

Goal. *We address the challenge of providing a new automatic refinement framework for static analysis, based on trace partitioning, that is generic and semantic-directed. Our framework takes as input a program and tries to find a refinement of it (through delayed joins) with maximum precision in the search space of all the possible refinements with regards to a given objective, and as minimal as possible in size.*

Challenges. There are several reasons why this goal is challenging, some immediately coming to mind (like path explosion and the size of the search space), but some that are much less obvious. The first contribution of this work is to identify the challenges of a search-based strategy to find an optimal refinement (detailed in Sec. 2, together with our solutions), such as:

- The possibly large size of the refined programs, as program disjunctions tend to multiply, and rapidly increase the number of program paths (*path explosion*);
- The *large size of the (refinement) search space*, which is impossible to exhaustively explore[3]. This is because possible precision improvements are not independent: sometimes several changes must be done simultaneously to observe a precision improvement; moreover sometimes changing a part of the program may make a previous change in another part unnecessary;
- The *accumulated computation time*: fixpoint computation can be an expensive operation, and doing it repeatedly during the search seems to be prohibitively expensive;
- The fact that often abstract interpreters have *non-monotonic operators* (like widening), meaning that local improvements can actually decrease the overall precision;
- The problem of *characterizing improvement*: it is often possible to endlessly make a program more precise by unrolling it. Which parts of the program are worth improving, and is it better to improve these parts simultaneously, or one after the other?

Contributions. To solve the above challenges, we propose the following contributions that allow implementing a search-based method for automated program refinement and make it more efficient.

- First, the possible program refinements are represented as *tuples* for which each dimension represents a choice of a parameter (i.e. split locations) that can be improved, and where a higher value in this dimension represents a program which is both larger and more precise. We maintain correspondence between tuples and the refined program using a homomorphism, and provide an example describing how control flow can be delayed in Sec. 4 (we implement and evaluate other possible improvements, like context-sensitivity for different call sites or unrolling). We then propose comparing the precision of the program refinements on specific *location of interest*. These tuples are the basis for our *ratchet search* strategy, which alternates phases that try to improve the precision with phases that decrease the program size within the same precision class, and allows pruning the search space by not testing regions that would decrease precision;
- We implement *incremental computation* (Sec. 5) to avoid recomputing the full fixpoint when testing a new program refinement. Computing incremental program refinement derives from the homomorphism, and we discuss different optimizations to improve the incremental fixpoint computations;
- We use a modified incremental fixpoint computation as a way to *force monotonicity* of the refinement process for non-monotonic abstract operators (Sec. 6), i.e. our incremental fixpoint computation is more precise than recomputing the fixpoint from scratch, and guarantees that further refining a program cannot decrease precision;

[3] To illustrate, even with a maximum delay length $k = 100$ and a total of $n = 20$ split-points, we could be faced with up to $\geq 100^{20}$ refinements.

– We provide a reference implementation of our search algorithm, and prove some properties about it (Sec. 7); in particular, given a bound on our search space, our algorithm guarantees that it can return a maximally precise program with minimal size within that bound. We also provide search strategies to quickly find suitable program refinements: Iterative Deepening Depth-First Delay Search (IDDDS) and Synchronized Delay Search (SDS), the latter mimicking common trace-partitioning heuristics;
– We implement our framework on top of existing abstract domain libraries and evaluate it in Sec. 8 on 1126 examples from the SVComp benchmark, demonstrating both its feasibility and its interest, as it performs better than standard baselines (no trace partitioning, full trace partitioning, SDS-partitioning), explores only a small fraction of the refinement space while finding complex refinements of interest (e.g., with refinement depth up to 160) and manages to prove some properties out of reach of well-optimized state-of-the-art CEGAR software model checkers [6,11,30,4];
– Finally, there is a large gap between problematic issues in theory and how well they work in practice; and we report on the important experimental points that make our approach practical (Sec. 8.5).

Overall, our method allows us to outperform standard trace partitioning heuristic strategies, especially being able to deal with large delays and to solve some problems out of reach of the currently best-known refinement approaches. Our findings and results should help static analyzer developers to integrate search-based strategies to improve static analysis operations that are cheap and yet provide the most important precision gains, or to research alternative strategies. To that end, we provide the code and benchmark used to perform our experiments at `https://zenodo.org/records/13308605`.

2 Analysis Refinement as an Optimization Problem

2.1 Program / Motivating Example

Consider the program *C-flow.c* as shown in Fig. 2 with unreachable *error*. Initially, $y \in \{1, 2\}$. Before the error at line 12, y is only modified in the `if`-branch with $y = 1$. Since the guard $y >= 3$ is never taken, the error is unreachable.

Let us consider analysis using the standard Constant Propagation `Cp` [36]. For the rest of the paper, we represent programs using control-flow graphs (CFGs): the CFG $P_{(0,0,0)}$ shown in Fig. 1 corresponds to *C-flow*. Additionally, we annotate each location with the abstract state (a pair $(x^\sharp, y^\sharp)$, denoting values of variables (x, y)) obtained by the `Cp` analysis. The analysis with `Cp` approximates y as $\top$ at location 2, given that the possible values for y are $\{1, 2\}$. Upon the propagation of $y = \top$ across the two branches, namely $y > 2$ and $y \leq 2$, the imprecision along branch $y > 2$ at location 8 causes the error location 12 to be (mistakenly) considered reachable in our analysis.

Enhancing precision through program refinement.. Given the initial CFG graph $P_{(0,0,0)}$ of *C-flow*, we a priori select *split-points* 2, 5, and 9 in the CFG

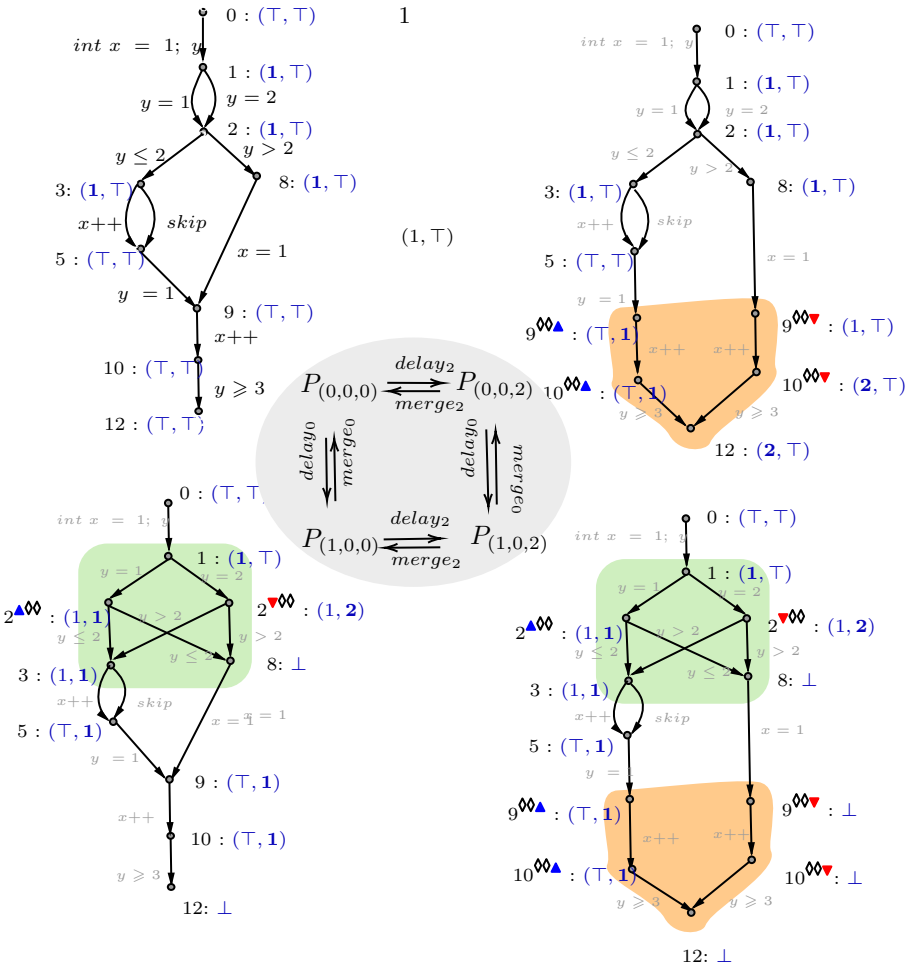


Fig. 1. Graph refinements of *C-flow.c*, and some possible transformations are: $P_{(1,0,0)}$= $\texttt{delay}_0(P_{(0,0,0)})$, $P_{(0,0,2)}$=$\texttt{delay}_2(\texttt{delay}_2(P_{(0,0,0)}))$, $P_{(1,0,2)}$=$\texttt{delay}_0(P_{(0,0,2)})$, $P_{(0,0,2)}$= $\texttt{merge}_0(P_{(1,0,2)})$. We represent Constant Propagation (`Cp`) abstract domain elements as pairs $(x^\sharp, y^\sharp)$ or else bottom $\bot$ at the locations of the CFGs

which are locations with multiple incoming edges. The key idea is to separate the merged paths to improve precision. $P_{(0,0,0)}$ can be transformed to $P_{(0,0,2)}$ by *delaying* the join at 9 by two steps. Intuitively, this split separates the paths from 9 and merges them at location 12. This signifies that the abstract state at 9 is now represented by the disjunction of the states at $9^{\Diamond\blacktriangle}$ and $9^{\Diamond\blacktriangledown}$ (delay leads to location duplication, and we distinguish different duplicates by colors labels). This disjunction enhances the precision at location 12. We can further transform $P_{(0,0,2)}$ into $P_{(1,0,2)}$ by delaying the split-point location 2 by one step, which further improves the precision at 12. The precision now at 12 is sufficient

to prove that the error is unreachable. We can go further and merge the delayed split in $P_{(1,0,2)}$, which results in $P_{(1,0,0)}$ with no loss of precision at 12. Instead of going through intermediate steps, we can also directly delay the split-point location 2 in the original CFG resulting in the transformed graph $P_{(1,0,0)}$.

```
1  int x = 1, y;
2  assume (1 <= y <= 2);
3  if (y <= 2) {
4      if (*)
5          x = x + 1;
6      y = 1;
7  }
8  else
9      x = 1;
10 x++;
11 if (y >= 3)
12     error();
```

Fig. 2. Example *C-flow.c*

Takeaways. The key idea provided by this example is that *refining* the program graph may improve the precision of the analysis, but increases the size of the program CFG. In this section, we will consider that refining programs consist of delaying join nodes, but other kinds of refinement are also possible (e.g., loop unrolling or increasing context sensitivity, that we consider in our evaluation; we show there that simultaneously combining different kinds of refinements is often required to improve precision). There are different choices (or dimensions) in program refinement that can be made, e.g., choosing which join node should be split. The dimensions are not orthogonal: refining in one direction may make a previous transformation in another dimension unnecessary. In this case, we can undo this previous transformation (e.g., merging back the previous split) to decrease the program size without having the analysis lose precision.

2.2 Program refinement as an optimization problem

A good program refinement is one that maximizes (or sufficiently increases) analysis precision while having minimal size, like $P_{(1,0,0)}$ in the example. Finding such a good program refinement, given any static analysis $\mathcal{A}$ and original program P, is our goal in this paper. As the example suggests, this can be viewed as an optimization problem, where we explore the search space of program *refinements* $\mathbb{G}$ to find a solution.

One question is what sufficiently precise means. Indeed, delaying a join node often leads to a small local improvement of precision on an unimportant program part. To solve this problem, we define a notion of *locations of interest* ($\mathcal{L}$), which are the control locations where we want the analysis precision to improve (typically, it will be the control location of an unproved assertion or analysis alarm). Note that improving precision at different locations of interest may need different program refinements. A question is, when we want to prove several properties, whether we should try to prove all of them simultaneously (one analysis with several locations of interest), or sequentially (several analyses with one location of interest. One lesson learned from our evaluation is that the latter is better, as often the former leads to path explosion.

2.3 Reducing the search space: from program refinement to tuples

One important problem is that the search space of possible program refinements is very large (and, in general, infinite when there are loops). Furthermore,

reasoning about program refinements (e.g. comparing whether a program is a refinement of another) is algorithmically complicated.

To solve both of these problems, we reduce the search space by considering program refinements that can be represented using a n-tuple. For instance, in Fig. 1, we use a triple where each dimension corresponds to the delay of one split-point. This means that there exists a function $\mathcal{H}$ mapping tuples to program refinements. Furthermore, we will require that $\mathcal{H}$ preserves ordering, meaning that if $t_1 < t_2$ (where $<$ is the component-wise ordering between tuples), then $\mathcal{H}(t_2)$ is a refinement of $\mathcal{H}(t_1)$. This makes $\mathcal{H}$ a *homomorphism* Note that this also implies that the size of $\mathcal{H}(t_1)$ is smaller than the size of $\mathcal{H}(t_2)$. For example, in Fig. 1, $P_{(1,0,2)}$ is a refinement of $P_{(1,0,0)}$ and has larger size, because $(1,0,0) < (1,0,2)$.

As the set of tuples in $\mathbb{N}^n$ is unbounded, we still need to bound the search space. For this we use a bound parameter k: i.e., we mandate the components of the tuple to be in the range $[0...k]$ for some bound k, thus making the search space finite as shown by the ellipses in Fig. 3, and we also have $\mathcal{H}([0...k]^n) \subset \mathbb{G}$. Note that if we cannot find a suitable program refinement given a bound k, it is possible to increase the bound to increase the search space, starting from the best refinement we obtained so far (making the search *anytime*, as you can always access the best program refinement when you stop it).

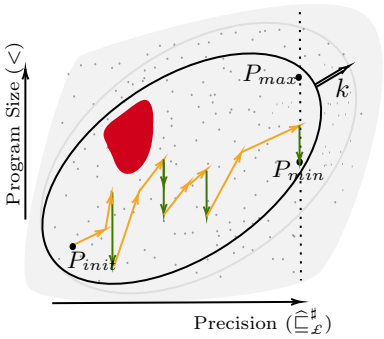


Fig. 3. Optimization Problem

Note that $P_{max} = \mathcal{H}((k,k,...,k))$ is the most precise refinement in this search space, so one may wonder why not just use this program refinement. The problem is that it is also the program refinement with the largest size in the space, and it is often very large due to path explosion, and often impossible to analyze in a reasonable amount of time and memory, as demonstrated by our experimental evaluation. This makes searching for a more precise refinement that limits the size of the programs necessary.

2.4 Ratchet Search: Continuous improvement of precision

Our search strategy should thus try to improve precision while keeping the program size as low as possible, which, as we saw, seem to be conflicting goals. The strategy that we propose for this consists of alternating two phases, one that explores program refinements to improve precision (in orange in Fig. 3), followed by one that minimizes the program while preserving the same precision (in green). We call it the *ratchet search strategy* because, once a precision improvement is found, all the further program refinements that we consider will

Table 1. Summary of properties across different types. t. Note: † indicates that all operators of $\mathcal{A}$ are monotonic

	Types	**Monotonicity**	**Incrementality**
Tuples	$\mathbb{N}^n_{\leq k}$	$t_1 < t_2$	$t_2 = \texttt{Succ}_i(t_1)$
Graphs	$\mathcal{H}([0...k]^n) \subset \mathbb{G}$	$\mathcal{H}(t_2)$ is a refinement of $\mathcal{H}(t_1)$	$delay_i(\mathcal{H}(t_1)) = \mathcal{H}(\texttt{Succ}_i(t_1))$
Fixpoints	$L \to D^\sharp$	$C(\texttt{Fix}_\mathcal{A}(\mathcal{H}(t_2)))\widehat{\sqsubseteq}^\sharp$ † $C(\texttt{Fix}_\mathcal{A}(\mathcal{H}(t_1)))$	$\texttt{fix-delay}_i(\mathcal{H}(t_1), \eta_1) \sqsubseteq^\sharp$ $\texttt{Fix}_\mathcal{A}(\mathcal{H}(t_2))$
Concrete States	$L \to \mathcal{P}(\Sigma)$	$\widehat{\gamma}(C(\texttt{Fix}_\mathcal{A}(\mathcal{H}(t_2)))) \subseteq$ † $\widehat{\gamma}(C(\texttt{Fix}_\mathcal{A}(\mathcal{H}(t_1))))$	$\gamma(\texttt{fix-delay}_i(\mathcal{H}(t_1), \eta_1)) \subseteq$ $\gamma(\texttt{Fix}_\mathcal{A}(\mathcal{H}(t_2)))$

have at least the same precision (we can only move forward in the direction of precision increase).

The key reason why this strategy works is that the component-wise ordering $<$ between tuples not only provides information about program size but also about precision ordering. Specifically, we already saw that if $t_1 < t_2$, then we have that $\mathcal{H}(t_2)$ is a refinement of $\mathcal{H}(t_1)$, and hence $|\mathcal{H}(t_1)| \leq |\mathcal{H}(t_2)|$ (the size of the program corresponding to t_2 is larger). For t_1, t_2 with their corresponding fixpoint maps $\eta_1 : L_1 \to D^\sharp, \eta_2 : L_2 \to D^\sharp$, we simultaneously have that $\eta_2 \widehat{\sqsubseteq}^\sharp_\mathcal{L} \eta_1$ for any given set of locations of interest $\mathcal{L}$ (that is a set of CFG nodes), where $\widehat{\sqsubseteq}^\sharp_\mathcal{L}$ represents the fact that the analysis is more precise on the locations of interest $\mathcal{L}$. Formally, $\eta_2 \widehat{\sqsubseteq}^\sharp_\mathcal{L} \eta_1 \stackrel{\text{def}}{=} \forall l \in \mathcal{L} : C(\eta_2)(l) \widehat{\sqsubseteq}^\sharp C(\eta_1)(l)$.

The left column in Table 1 explains why: if $\mathcal{H}(t_2)$ is a refinement of $\mathcal{H}(t_1)$, and assuming that all the analysis operators of $\mathcal{A}$ are monotonic (which is often the case except for the widening operator, see Sec. 6), then the result of computing a fixpoint on $\mathcal{H}(t_2)$ is at least as precise than computing it on $\mathcal{H}(t_1)$. Here, $C(\eta(\cdot))$ represents a projection of η on the locations of the original program to allow for comparison. Given that γ is also monotonic, this also translates to a (non-measurable) precision improvement in the concrete.

We can consider that the precision order $\widehat{\sqsubseteq}^\sharp_\mathcal{L}$ allows to quotient the tuple space using the "equal precision on $\mathcal{L}$" predicate $=^\sharp_\mathcal{L}$, that derives from $\widehat{\sqsubseteq}^\sharp_\mathcal{L}$. Exploration consists of finding a strictly more precise equivalence class, while minimization consists of finding a minimal tuple within the same equivalence class (note that there may be several incomparable minimal tuples). Note that when the set $\mathcal{L}$ of locations of interests is larger, then the equivalence classes of precision for $=^\sharp_\mathcal{L}$ shrinks, i.e., the minimal programs for a given level of precision are larger. This explains why it is better to focus the refinement on a small set of $\mathcal{L}$ at a time.

Our experiments show that minimization is rarely triggered in practice (situations like the motivating example are quite rare in practice). However, the strategy where we try to keep the graph as small as possible is very important;

in particular, it is very important to rollback on a smaller tuple when we explore increasing a dimension of the tuple that does not lead to an increase in precision, otherwise, the explored refinements suffer from a too large size due to path explosion.

2.5 The GSR algorithm

This ratchet search strategy readily translates into a search algorithm, that we call the Generic and Semantic-Directed Refinement algorithm (GSR for short), given in Algorithm 1.

Algorithm 1: Given initial program P, analysis $\mathcal{A}$, bound k, and $\mathcal{L}$ a set of locations, GSR returns a tuple t_b with fixpoint map η_b that is at least as precise as the analysis on maximum trace-partitioning in $\mathbb{N}^n_{\leq k}$. The code in gray is activated later to do incremental fixpoint computation

```
1  Procedure GSR(P, A, k, £)
2      t_b ← (0, ..., 0), η_b ← Fix_A(P), Ω ← ∅
3      t_c ← t_b, η_c ← η_b
4      while (1) do
5          //old_t_c ← t_c, old_η_c ← η_c
6          t_c ← Next(t_b, k, Ω)
7          if (t_c == None) then break
8          η_c ← Fix_A(H_P(t_c)) //η_c ← Fixpoint_Inc(old_t_c, old_η_c, t_b, η_b, t_c)
9          if η_c ⊑♯_£ η_b then
10             t_b, η_b ← Minimize_A(t_c, η_c)
11         else
12             Ω ← Ω ∪ PruneRegions(t_c, t_b, Ω)
13     return t_b, η_b
```

The algorithm maintains a current best refinement t_b and its corresponding fixpoint map η_b. Then, it uses the `Next` function to try to find a program refinement that improves on precision (this corresponds to the precision improvement phase in orange in Fig. 3). When it finds one, it tries to find a smaller tuple that maintains the same precision using the `Minimize` function (this corresponds to the minimization phase in green in Fig. 3). Another component is the `PruneRegions` part (corresponding to the red area in Fig. 3): indeed sometimes we can learn that refining some dimensions is useless and save this information for later search. We did not experiment on finding a good pruning strategy so we do not expand further on it (we leave this direction of research open for future work).

The implementation of `Minimize` is quite simple: starting from a tuple t_c, we test all the immediate predecessors of t_c, and stop if we cannot find any that maintains the same precision, or repeat from this predecessor if one is found.

This guarantees finding a minimal element in the precision equivalence class. This is a very nice property to perform efficient minimization. As a result, the worst-case minimization complexity is $\mathcal{O}(k \cdot n)$ where n is the number of locations ($\approx$ size of the program), and the minimum complexity is just $\mathcal{O}(n)$. Incremental computation (see subsection Sec. 5) allows doing this very efficiently.

2.6 Tuple exploration strategy

A very important element is the choice for the `Next` function, i.e., the ordering strategy in which we explore tuples. The main problem here is that the search space remains very large: there are k^n n-tuples when we use a bound k and n is the program size.

However, if we restrict the search space to consider only tuples with a single dimension (an index of the tuple) improved, then the search space is much smaller (its size is $n \times k$). Thus, a realizable strategy is to find dimensions that improve precision one at a time, as we did in the motivating example of Fig. 1. This is what our Iterative Deepening Depth-first Delay Search (IDDDS) algorithm does initially. To find the next suitable dimension, it uses an iterative deepening [38] strategy, consisting of exploring improving a dimension using a depth limit that is increased when no improvement is found. Our experiments show that this strategy works very well in practice, allowing us to quickly find precision improvements to the analysis, even with large values of k and n (this allows us to set k to a large value and not worry about this customization parameter, and also to try every split-point instead of having to manually select a subset of suitable split-points).

Unfortunately, this strategy is not *complete*, meaning that there are precision improvements that can only be found if you simultaneously improve on several dimensions, that the above strategy will miss. We found that, for instance, some SV-COMP [10] instances could only be improved when three splits are simultaneously performed. This is why, after IDDDS has completely explored all the possibilities for 1-dimension improvement, it tries all combinations of 2-dimension improvement (the search space for 2-dimension improvements is of size $n \times (n-1) \times k$ if we increase both dimensions simultaneously). It then continues with combinations of 3-dimensions, etc. Unfortunately, our experimental section shows that exploring these combinations times out without finding any improvement in precision. An interesting research direction would consist of finding a good strategy to find which combination of dimensions is worth trying.

As a comparison, we also tried the SDS strategy which uniformly increases all tuple components from $(0, ..., 0)$, $(1, ..., 1)$, and finally to (k, k, k). This resembles common heuristics that apply a fixed delay to every split point [44], or uniformly apply a fixed level of context-sensitivity [49]. This strategy has the benefit of a very small search space (of only k) and works moderately well, but eventually suffers from program size explosion issues, as shown in our experiments.

2.7 Incremental Computation

While dealing with the large search space is important, the time required by each fixpoint computation is another important aspect. We already said that we want to minimize the graph size to avoid program size explosion issues. Another aspect is dealing with redundant computation stemming from the analysis of different variants of the same program.

Our search strategy is based on local improvements (updating tuples by incrementing one dimension at a time). Following this we want to locally update the program graph and its corresponding fixpoint map.

For a successor function $\texttt{Succ}_i : \mathbb{N}^n \to \mathbb{N}^n$ that takes a tuple t_1 and outputs $t_2 = \texttt{Succ}_i(t_1)$ by incrementing i^{th} component by 1, we have a corresponding $\texttt{delay}_i : \overline{\mathbb{G}} \to \overline{\mathbb{G}}$ function (where $\overline{\mathbb{G}} = \{\mathcal{H}(t) | t \in \mathbb{N}^n\}$) that applies the transformation of one-step delay on the paths corresponding to split-point i on the graphs. As shown in Table 1, the homomorphism $\mathcal{H}$ preserves the structure of the $\texttt{Succ}_i$ function: $\texttt{delay}_i(\mathcal{H}(t)) = \mathcal{H}(\texttt{Succ}_i(t))$. Similarly, we define $\texttt{merge}_i : \overline{\mathbb{G}} \to \overline{\mathbb{G}}$ functions that undo the one-step delay and are such that $\texttt{merge}_i(\mathcal{H}(t)) = \mathcal{H}(\texttt{Succ}_i^{-1}(t))$.

The fact that graph and tuple updates correspond through homomorphism allows to map any transformation on the tuples (which can be decomposed as a succession of calls to $\texttt{Succ}_i$ and $\texttt{Succ}_i^{-1}$) to a corresponding transformation on the graph, and is the basis for our incremental computation: the refinements are always modified in-place instead of being recomputed. For instance, to compute the graph $P_{(3,1,2)} = \mathcal{H}(3,1,2)$ starting from the graph for $P_{(2,1,4)} = \mathcal{H}(2,1,4)$, we can use the property $P(3,1,2) = \texttt{delay}_0(\texttt{merge}_2(\texttt{merge}_2(P_{(2,1,4)})))$

Furthermore, for each $\texttt{delay}_i$ and $\texttt{merge}_i$ operations on the graph, we implement a corresponding $\texttt{fix-delay}_i$ and $\texttt{fix-merge}_i$ on fixpoint maps that perform incremental fixpoint computations. For a tuple t_1 with an existing fixpoint map η_1, the fixpoint map for tuple t_2 can be efficiently computed using $\texttt{fix-delay}_i(t_1, \eta_1)$. It only computes fixpoints for the parts of the graph $\mathcal{H}(t_2)$ affected by the $\texttt{delay}_i$ transformation, while reusing previous results of η_1 for the unaffected parts of $\mathcal{H}(t_2)$. In experiments, we demonstrate that thanks to this incrementality, we can test several thousands of refinements in a few seconds.

All this is done by the `Fixpoint_Inc` function (commented out in the definition of the GSR algorithm). Notice that this function takes both the previous candidate fixpoint map $\texttt{old\_}\eta_c$ and the previous best fixpoint map η_b: depending on the tuple, it may be better to start the incremental fixpoint computation from one to the other (i.e., starting over from η_b is a kind of a non-chronological backtracking [50]).

Another important advantage of incremental fixpoint computation is that it allows dealing with non-monotonic abstract operators (in particular, widening is never monotonic [19]). Indeed, we saw that our GSR algorithm relies on the fact that tuple ordering is an approximation of precision ordering as shown in the column 3 of Table 1, which is not true when the abstract operators are not monotonic. Luckily, we can solve this problem by incremental computation, i.e. we can ensure that for any tuple t and any refinement $\texttt{Succ}_i(t)$, the fixpoint

map corresponding to $\texttt{Succ}_i(t)$ will be at least as precise as the fixpoint map η corresponding to t, i.e.: $\texttt{fix-delay}_i(\mathcal{H}(t), \eta) \sqsubseteq^\sharp \texttt{Fix}_\mathcal{A}(\mathcal{H}(t))$. (see Sec. 5). This is achieved simply by intersecting the newly computed fixpoint map for $\texttt{Succ}_i$(t) with the one we had for t, η.

3 Background

3.1 Notations

The set of natural numbers (containing 0) is denoted by $\mathbb{N}$ with the usual ordering $\leq$. The partial order $\preceq$ on $\mathbb{N}^n$, the set of n-tuples of natural numbers, is given by its component-wise extension: for any $t, u \in \mathbb{N}^n$, we say $t \preceq u$ if and only if $t[i] \leq u[i]$ for each index $i \in \{1, \ldots, n\}$. If $t \preceq u$ and $t \neq u$, we write $t \prec u$. For a given $k \in \mathbb{N}$, we denote by $\mathbb{N}^n_{\leq k}$ the set $\{t \in \mathbb{N}^n \mid \forall i \in \{1, \ldots, n\} : t[i] \leq k\}$, that is, all n-tuples in $\mathbb{N}^n$ such that each component is less than or equal to k. The powerset of a set S is denoted by $\mathcal{P}(S)$, and the finite powerset of S consisting of all finite subsets of S is denoted by $\mathcal{P}_{\text{fin}}(S)$. Lastly, we define $\overline{\mathbb{N}}$ as the set $\mathbb{N} \cup \{-\infty\}$ and extend the ordering $\leq$ to include $-\infty$ such that for every $i \in \mathbb{N}$, $-\infty < i$.

3.2 Program Model

We consider a program to be given by a control flow graph (CFG) $P = (L, E, l_0)$, which is a 3-tuple containing a set L of control locations, a set E of constraint-labeled directed edges, and an initial location $l_0 \in L$. We denote the set of all program CFGs as $\mathbb{P}$. A *path* π is a finite sequence of edges $(l_1, e_1, l_2), (l_2, e_2, l_3)$, ..., (l_{k-1}, e_{k-1}, l_k), and we use the notation $l_1 \rightsquigarrow l_k$ to denote the path from l_1 to l_k. We use helper function $\texttt{Loc}(\pi) = \{l_1, ..., l_k\}$ for set of locations of the path.

We can use Bourdoncle's algorithm [13] to decompose a CFG into nested strongly connected components (SCC) or loops, producing a weak topological ordering (WTO). For the sake of simplicity, we assume that CFG is reducible [2]. This implies that every SCC has a single entry location called the *loop-head*. We define the set of all loop-heads as L_{lh}. For any loop-head $l_{lh} \in L_{lh}$, the locations within the loop are denoted by the set $SCC(l_{lh})$. A location $l \notin L_{lh}$ is a *join-point* if it has in-degree ≥ 2. The set of join-points is represented by $L_\sqcup$.

The set of program states is denoted as Σ. The edge constraints are interpreted by the concrete denotational semantics $[\![\cdot]\!] : E \to \Sigma \rightharpoonup \Sigma$ as partial functions over program states.

3.3 Abstract Interpretation

An abstract interpreter [19] is a 6-tuple $\mathcal{A} = (D^\sharp, d_0, [\![.]\!]^\sharp, \sqsubseteq^\sharp, \sqcup, \nabla)$ consisting of an abstract domain $D^\sharp$, or abstract states that form a semi-lattice under the *partial order* $\sqsubseteq^\sharp \subset D^\sharp \times D^\sharp$ with a *bottom* $\bot \in D^\sharp$ (least element under $\sqsubseteq^\sharp$) and the *upper bound (join)* $\sqcup : D^\sharp \times D^\sharp \to D^\sharp$. The state $d_0 \in D^\sharp$ is the initial

abstract state, and the *abstract semantics* $[\![\cdot]\!]^\sharp : E \to D^\sharp \to D^\sharp$ interprets edge constraints as monotone functions over $D^\sharp$. The *widening* $\nabla : D^\sharp \times D^\sharp \to D^\sharp$ is an upper bound operator such that $d_1 \sqcup d_2 \sqsubseteq^\sharp d_1 \nabla d_2$ for all $d_1, d_2 \in D^\sharp$, and repeated applications of ∇ on an increasing sequence of abstract elements converges to a fixed point in a finite number of iterations.

A concretization function $\gamma : D^\sharp \to \mathcal{P}(\Sigma)$ gives the semantics of abstract domain elements. We require $\mathcal{A}$ to be sound, meaning that

$$\forall e \in E, d \in D^\sharp : [\![e]\!](\gamma(d)) \subseteq \gamma([\![e]\!]^\sharp(d)) \tag{1}$$

$$\forall d_1, d_2 \in D^\sharp : \gamma(d_1) \subseteq \gamma(d_1 \sqcup d_2) \wedge \gamma(d_2) \subseteq \gamma(d_1 \sqcup d_2) \tag{2}$$

$$\forall d_1, d_2 \in D^\sharp : d_1 \sqsubseteq^\sharp d_2 \implies \gamma(d_1) \subseteq \gamma(d_2) \tag{3}$$

If $\mathcal{A}$ satisfies the above equations, we can define a function computing a sound over-approximation of the reachable states of an input program $P = (L, E, l_0)$ using the function $\texttt{Fix}_\mathcal{A}$. The result $\eta = \texttt{Fix}_\mathcal{A}(P)$ is in $L \to D^\sharp$ can be computed as a transitive closure of the abstract semantics of edge constraints over a control flow graph. We call a result $\eta : L \to D^\sharp$ a *fixpoint-map* if $\forall l \in L : \gamma(\eta(l))$ includes all the reachable states at location l in P. It is well-known that $\texttt{Fix}_\mathcal{A}$ indeed computes a fixpoint map. Note that a fixpoint map is an invariant but not necessarily an inductive invariant. Indeed our incremental fixpoint computation (Sec. 5) may produce non-inductive invariants.

Finite powerset extension. The finite powerset extension [1,46] of an abstract domain $D^\sharp$ is denoted as $\widehat{D}^\sharp \stackrel{\text{def}}{=} \mathcal{P}_{fin}(D^\sharp)$ with the ordering $\widehat{\sqsubseteq}^\sharp$, also known as the *Hoare powerset extension* ordering. For $\widehat{d_1}, \widehat{d_2} \in \widehat{D}^\sharp$, we have $\widehat{d_1} \widehat{\sqsubseteq}^\sharp \widehat{d_2}$ iff $(\forall d_1 \in \widehat{d_1}. \exists d_2 \in \widehat{d_2}. d_1 \sqsubseteq^\sharp d_2)$. The ordering $\widehat{\sqsubseteq}^\sharp$ on $\widehat{D}^\sharp$ can be lifted point-wise to compare maps $L \to \widehat{D}^\sharp$.

The concretization function $\widehat{\gamma}$ for $\widehat{D}^\sharp$ is defined as $\widehat{\gamma}(S) = \bigcup_{s \in S} \gamma(d))$ for every $S \in \widehat{D}^\sharp$.

3.4 Graph Refinement Notion

We now define the notion of refinement of a graph as illustrated in [44,46].

Definition 1 (Refinement). Given graphs $P = (L, E, l_0)$ and $P_r = (L_r, E_r, l_{0r})$ over the same set of constraints, P_r is a *refinement* of program P iff there exists a map $\tau : L_r \to L$ such that:

1. $\tau(l_{0r}) = l_0$
2. No missing behavior in P_r: Consider locations l, l_r such that $\tau(l_r) = l$. For all $(l, c, l') \in E$, there exists $(l_r, c, l'_r) \in E_r$ such that $\tau(l'_r) = l'$; and,
3. No spurious behavior in P_r: Consider locations l and l_r such that $\tau(l_r) = l$. For all $(l_r, c, l'_r) \in E_r$, there exists $(l, c, l') \in E$ such that $\tau(l'_r) = l'$.

Condition (1) requires the initial location correspondence. Condition (2) ensures that P_r simulates every edge of P, and condition (3) ensures that no spurious edge is added in P_r when compared to P. Hence, for a refinement P_r of P, P and P_r are *bisimilar*.

Definition 2 (Collapsing). Consider a refinement $P_r = (L_r, E_r, l_{0r})$ of $P = (L, e, l_0)$ with map $\tau : L_r \to L$ and a fixpoint map $\eta_r : L_r \to D^\sharp$. The *collapse* of η_r is a map $C(\eta_r) : L \to \widehat{D}^\sharp$ on P such that $\forall l \in L. C(\eta_r)(l) \stackrel{\text{def}}{=} \{\eta_r(l_r) \mid \tau(l_r) = l, l_r \in L_r\}$.

To compare the precision of any two refinements, we first use the collapsing operator to compute for every location l in P, the set of domain objects at duplicated locations corresponding to l in refinements $\mathcal{H}(t_i)$ and $\mathcal{H}(t_j)$. Then for every $l \in \mathcal{L}$, we compare their sets using Hoare powerset extension ordering $\widehat{\sqsubseteq}^\sharp$.

4 Tuples to Refinements: a Homomorphism

This section introduces techniques to manage the large search space. We formalize the notion of tuples to quotient the search space and define a homomorphism that maps each tuple to a graph refinement.

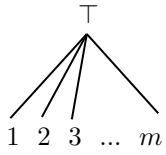
4.1 Tuples

Let L_{sp} be a set of n *split-points*, selected *a priori* based on the edges of the CFG $P = (L, E, l_0)$, and satisfying $L_{sp} \subseteq L_\sqcup \cup L_{lh}$ where $L_\sqcup$ and L_{lh} are disjoint sets. Each split-point in L_{sp} is associated with an index from the set $\{0, \ldots, n-1\}$. Corresponding to each index i, the n-tuple $t \in \mathbb{N}^n$ defines $t[i]$ as the bound associated with the split-point at index i. In the following, we discuss homomorphism from n-tuples $\mathbb{N}^n$ to graph refinements. We use helper function $\texttt{sp}(i)$ to denote the split-point associated with i. We also use the helper function $\texttt{in-degree}(l)$ to denote the number of incoming edges of l in P.

4.2 Graph coloring for graph transformations

In the following we define two sets of colors along with order relations for two graph transformations on the CFGs: 1) separation of join paths; 2) loop unrollings. These colors are used to distinguish different duplicated locations of the original program.

Colors for join-splitting ($\mathcal{C}_S$). Consider a semi-lattice of $m+1$ colors $\mathcal{C}_S = \{1, \ldots, m, \top\}$ with an order relation $\prec_S$ defined such that $\prec_S \stackrel{\text{def}}{=} \{(i, \top) | i \in \{1, ..., m\}\}$ for all $i \in \{1, \ldots, m\}$. For two colors $c, c' \in \mathcal{C}_S$, we have $c = c'$ iff they are the same element. This set is used to manage colors associated with join-points in the CFG.



Colors for loop unrollings ($\mathcal{C}_L$). We consider another set of $m+2$ colors $\mathcal{C}_L = \{1, 2, \ldots, m, \infty, \top\}$ with a relation $\prec_L \stackrel{\text{def}}{=} \{(i, i+1) | i \in \mathcal{C}_L - \{\infty\}\} \cup \{(m, \infty), (\infty, \top)\}$. The colors $1, ..., m$ are used for coloring unrolled locations of the loop to track iterations effectively. Color ∞ is associated with the final loop locations, and $\top$ is for the locations outside of the loop.

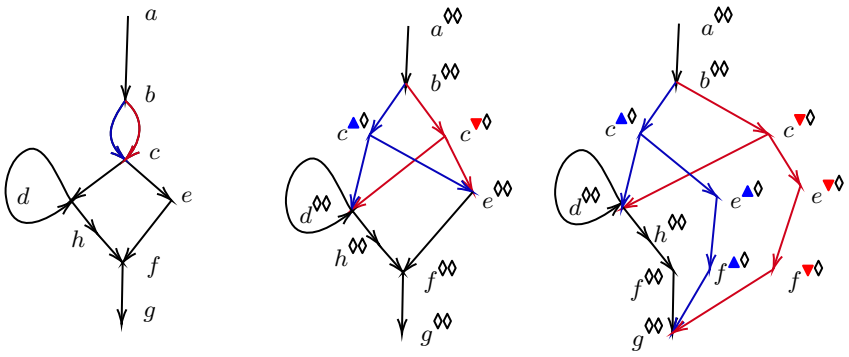


Fig. 4. a); Example initial refinement $R_{(0,0)}$ with tuple size 2 for split-points c and f. b) Delay of join of split-point c by one-step: $R_{(1,0)}$; c) Delay of c by three steps: $R_{(3,0)}$

The set of color words $\mathcal{C}^n$ of length n is defined as follows:

$$\mathcal{C}^n = \{w \mid \forall i \in \{0,\ldots,n-1\} \,:\, (\mathtt{sp}(i) \in L_{\sqcup} \implies w[i] \in C_S \vee \\ \mathtt{sp}(i) \in L_{lh} \implies w[i] \in C_L)\}$$

In the following, we discuss the role of graph coloring in performing and reversing graph transformations.

4.3 Separating Join paths

We now discuss a distance notion to separate paths and then adapt graph coloring to distinguish duplicated locations.

Given the original CFG $P = (L, E, l_0)$, we define a binary relation $\ll$ on location set L, derived from edges E in P. For locations $l, l' \in L$, we have $l \ll l'$ if there exists an edge $(l, e, l') \in E$ and $l, l' \notin L_{lh}$, that are not loop-heads (see background definition). We extend $\ll$ recursively to represent paths of increasing lengths: $\ll^1 \stackrel{\text{def}}{=} \ll$, and for each $d \in \mathbb{N}$, $\ll^{d+1} \stackrel{\text{def}}{=} \ll^d \circ \ll$, where $\circ$ denotes relational composition. The restriction on $\ll$ for loop-heads ensure that in every considered path $\pi : (l_1, e_1, l_2), \ldots, (l_{p-1}, e_{p-1}, l_p)$, the locations $l_1, \ldots, l_p$ are not loop-heads.

Remark 1. While we handle loops through unrolling (Sec. 4.4), we simplify the presentation of homomorphism and do not handle the separation of paths beyond loops. We constrain $\ll$ by not considering loop-heads. A straightforward solution to this is that any one-step separation beyond a loop-head should duplicate the entire loop corresponding and merged before the start of the loop exit-path to preserve homomorphism. In the example shown in Fig. 4c, the separation stops at the loop-head and the loop is not duplicated.

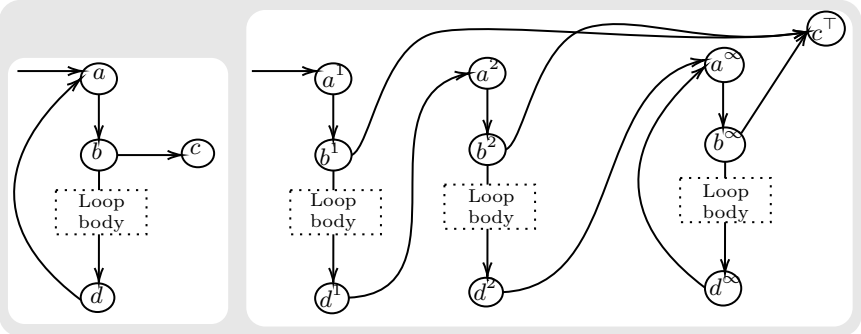


Fig. 5. a) Refinement $S_{(0)}$ with split-point a b) Refinement $S_{(2)}$ after delaying (unrolling) the loop by two steps

The function $d_{max} : L \times L \to \overline{\mathbb{N}}$ represents the maximum path length between two locations, given by $d_{max}(l, l') = \begin{cases} 0 & \text{if } l = l' \\ \max\{d \in \mathbb{N} | l \ll^d l'\} & \text{if such a } d \text{ exists.} \\ -\infty & \text{otherwise} \end{cases}$

Example. Consider the CFG $R_{(0,0)}$ as shown in Fig. 4. Here, $d_{max}(a, f) = 4$, $d_{max}(a, h) = -\infty$, $d_{max}(h, g) = 2$, and $d_{max}(d, g) = -\infty$.

Graph Coloring to separate paths. Consider a join-point $l_\sqcup$ associated with index i and has an in-degree m. Consider color set $\mathcal{C}_S = \{1, ..., m, \top\}$ of $m + 1$ colors. Every location in a refinement is a pair (l, w) such that w is a color word and $\in \mathcal{C}^n$, and $w[i]$ denotes the color associated with a split-point. For join-points, we illustrate using color symbols instead of $\mathcal{C}_S$ by mapping $1 \to \blacktriangle, 2 \to \blacktriangledown, \top \to \lozenge$ for $m = 2$. Since each join-point may have different in-degree in the original program, the maximum colors needed for each such join-point may vary.

For any location (l, w) in a refinement, the colors $\blacktriangle$ or $\blacktriangledown$ for $w[i]$ indicate that separating of paths from join-point $l_\sqcup$ at index i led to the creation of (l, w). In contrast, the color $w[i] = \lozenge$ is given when join-point does not participate in the creation of (l, w).

Example. Consider the example CFG shown in Fig. 1, where the initial refinement has every location labeled with colors $\lozenge\lozenge\lozenge$. Consider three chosen join-points 2, 5, and 9, located at indices 0, 1, and 2 in the tuples respectively. From the initial refinement $P_{(0,0,0)}$, delaying join-point at index 0 by one step results in duplicating location 2 into $2^{\blacktriangle\lozenge\lozenge}$ and $2^{\blacktriangledown\lozenge\lozenge}$, leading to refinement $P_{(1,0,0)}$. Similarly, delaying join-point at index 2 by two steps duplicates the location 9 into $9^{\lozenge\lozenge\blacktriangle}$ and $9^{\lozenge\lozenge\blacktriangledown}$ and location 10 into $10^{\lozenge\lozenge\blacktriangle}$ and $10^{\lozenge\lozenge\blacktriangledown}$.

4.4 Unrolling loops

For a loop-head $l_{lh} \in L_{lh}$, we consider the demanded unrolling [54] of loop for l_{lh} and adapt graph coloring using color set $\mathcal{C}_L = \{1, ..., m, \infty, \top\}$ for locations

to be able to perform/undo an unrolling. For l_{lh} with an unrolling length m, any j^{th} unrolling creates new locations such that every location is associated with a color word w such that $w[i] = j$. The final loop locations and the locations outside of unrolling have the default color $w[i] = \infty$. Note that, by definition of $\ll^i$, join-points inside loops are partitioned only until they reach some $l_{lh} \in L_{lh}$. For all loop-heads, we use a fixed maximum number of colors which is exactly the maximum unrolling length used. In the next section, we show how $\prec_L$ relation by construction allows for unrolling shown in Fig. 5. Any nested loops within the loop are duplicated during the unrolling process.

To reverse an unrolling, all locations corresponding to the most recent unrolled color m are removed, and edges that connected locations with color $m-1$ to those with color m are redirected to the loop-head.

Example. Consider the example show in Fig. 5 with one split-point associated with loop-head. The color words of size 1 is shown in the superscript of each location, and the loop is unrolled twice. All locations created with the new unroll is given the same numeric color, which is the new unroll depth. The final loop locations have the default color ∞.

4.5 From Tuples to Graph Refinements: A Homomorphism

Given program $P = (L, E, l_0)$ and n split-points, we introduce a coloring function $Col : \mathbb{N}^n \to (L \times \mathcal{P}(\mathcal{C}^n))$, associating each n-tuple $t \in \mathbb{N}^n$ with a function that maps locations in L to subsets of $\mathcal{C}^n$.

Through this, we know how each location of the original program is partitioned into multiple locations in refinement and each labeled with a color word $w \in \mathcal{C}^n$. To define $Col(t, l)$ where $t \in \mathbb{N}^n$ and $l \in L$, we use intermediate relations *zpath*, *comp*, *split*, *unroll* which are given by:

$$
\begin{aligned}
\mathit{zpath}(l', l) &\iff \exists \pi : l_1 \rightsquigarrow l : \ l' \notin \texttt{Loc}(\pi) \wedge d_{max}(l', l_1) = -\infty \\
\mathit{split}(t, l, w, i) &\iff l' = \texttt{sp}(i) \wedge l' \in L_{\sqcup} \wedge m = \texttt{in-degree}(l') \wedge \\
&\quad\ d = d_{max}(l', l) \ \wedge (d \geq t[i] \vee d = -\infty \implies w[i] = \top) \wedge \\
&\quad\ (0 \leq d < t[i] \implies (w[i] \in \mathcal{C}_S \wedge (\neg \mathit{zpath}(l', l) \Rightarrow w[i] \neq \top))) \\
\mathit{unroll}(t, l, w, i) &\iff l' = \texttt{sp}(i) \ \wedge \ l' \in L_{lh} \ \wedge \ m = t[i] \wedge \\
&\quad\ ((l \notin \mathit{SCC}(l') \vee m = 0) \implies w[i] = \top) \wedge \\
&\quad\ (l \in \mathit{SCC}(l') \wedge m > 0 \implies w[i] \in \mathcal{C}_L - \{\top\})
\end{aligned}
\tag{4}
$$

We now define $Col(t, l)$ as follows:

$$
\mathit{Col}(t, l) \stackrel{\text{def}}{=} \{ w \in \mathcal{C}^n \ \mid \forall i \in \{0, ..., n-1\} : \ \mathit{split}(t, l, w, i) \vee \mathit{unroll}(t, l, w, i) \}
\tag{5}
$$

Example. Consider $R_{(0,0)}$ with split-points $c, f \in L_{\sqcup}$ as shown in Fig. 4, where only c is delayed as shown by colored edges.

– *zpath*: Consider $R_{(3,0)}$ in Fig. 4c. When a location l is duplicated as (l, w) by delaying split-point i, the constraint *zpath* says when w satisfies $w[i] \neq \Diamond$. For instance when delaying split-point c, both c and e are duplicated as $c^{\blacktriangle\Diamond}, c^{\blacktriangledown\Diamond}$, $e^{\blacktriangle\Diamond}$, and $e^{\blacktriangledown\Diamond}$. However, we do not create locations $c^{\Diamond\Diamond}, e^{\Diamond\Diamond}$. In contrast, for location f, the delay of c not only creates $f^{\blacktriangle\Diamond}, f^{\blacktriangledown\Diamond}$, but also $f^{\Diamond\Diamond}$ with $w[0] = \Diamond$. The reason is that $f^{\Diamond\Diamond}$ is essential for all the paths not interfered during the delay of c.
– *split*(t, l, w, i): Consider the delay of c with distance $t[0] = 3$ as shown in $R_{(3,0)}$. Here, location e has $d_{max}(c, e) = 2$, and since $0 \leq d_{max}(c, e) < 3$. location e will be duplicated with words $\blacktriangle\Diamond, \blacktriangledown\Diamond$. Since the only incoming edge of e is along the delay path, *zpath* ensures that e cannot have a duplicated location with the word $\Diamond\Diamond$. For location d, the delay of c is blocked since d is a loop-head as shown in the figure.

Homomorphism. Given program $P = (L, E, l_0)$ and a set of all program CFGs $\mathbb{P}$, we define *homomorphism* $\mathcal{H}_P^{Col} : \mathbb{N}^n \to \mathbb{P}$ that maps each tuple $t \in \mathbb{N}^n$ to a graph that complies with the constraints of the tuple and the color labeling. When P and Col are clear from the context, we use $\mathcal{H}$ instead of $\mathcal{H}_P^{Col}$.

A graph $P_r = \mathcal{H}_P^{Col}(t)$ composed of $P_r : (L_r, E_r, l_{0r})$ is defined as follows:

1. The set of locations $L_r = \{(l, w) \mid l \in L, w \in Col(t, l)\}$
2. The set of edges E_r is such that $((l_1, w_1), e, (l_2, w_2)) \in E_r$ if and only if $(l_1, e, l_2) \in E$ and one of the following hold for every i
   (a) $\mathtt{sp}(i) \in L_\sqcup \wedge (w_1[i] \preceq_S w_2[i] \vee (w_2[i] \prec_S w_1[i] \implies l_2 = \mathtt{sp}(i)))$
   (b) $\mathtt{sp}(i) \in L_{lh}$ and one of the following holds
      i. $l_2 = \mathtt{sp}(i) \wedge ((w_1[i] = \top \wedge w_2[i] = 1) \vee w_1[i] \prec_L w_2[i] \vee w_1[i] = w_2[i] = \infty)$
      ii. $l_2 \neq \mathtt{sp}(i) \wedge (w_1[i] = w_2[i] \vee w_2[i] = \top)$
3. The initial location $l_{0r} = (l_0, w)$ such that $l_0 \in L$ and $w \in \mathcal{C}^n$ with all entries set to $\top \in \mathcal{C}_S$ for join-points indices and to $\top \in \mathcal{C}_L$ for loop-head indices.
4. Finally, τ map is defined as follows: $\tau(l, w) = l$

5 Incremental Computation

In this section, we formalize how to perform one-step refinement through incremental computation. We first motivate incremental graph transformations and then discuss how to efficiently do incremental fixpoint computation on the transformed graph. Both these methods help make the one-step refinement process efficient thus making the search over a large search space feasible.

Consider the set of n-tuples $\mathbb{N}^n$ and its n successor functions $\mathtt{Succ}_0, ..., \mathtt{Succ}_{n-1}$ on tuples. We define a one-step relation $\mathcal{R}$ as follows: for tuples $t_1, t_2 \in \mathbb{N}^n$, $t_1 \mathcal{R} t_2$ if and only if there exists an i such that the $t_2 = \mathtt{Succ}_i(t_1)$. The ordering $\leq$ on tuples in $\mathbb{N}^n$ is defined as the reflexive and transitive closure of $\mathcal{R}$.

5.1 Incremental Graph Refinement

Consider a left-total relation $\mathcal{R}$ between tuples in the graph. If $\mathcal{R}$ is a function $\mathcal{H}$ (i.e. associate a single graph to every tuple), then we can define for each successor function $\texttt{Succ}_i$ a corresponding function $\texttt{delay}_i$ defined as: $\forall t, \forall i : \texttt{delay}_i(\mathcal{H}(t)) \triangleq \mathcal{H}(\texttt{Succ}_i(t))$. We can also define $\texttt{merge}$ functions defined as: $\forall t, \forall i : t[i] > 0 \Rightarrow \texttt{merge}_i(\mathcal{H}(t)) \triangleq \mathcal{H}(\texttt{Succ}_i^{-1}(t))$. These definitions imply that $\mathcal{H}$ is a homomorphism that preserves the tuple structure. This implies in particular that $\forall i : \texttt{merge}_i \circ \texttt{delay}_i = Id$ (we can undo graph refinements), or that the delay functions are commutative: $\forall i, j : \texttt{delay}_i \circ \texttt{delay}_j = \texttt{delay}_j \circ \texttt{delay}_i$ (meaning that the graph obtained by applying several refinements does not depend on the order in which these refinements are applied).

When applied to the homomorphism $\mathcal{H}$ defined in Sec. 4.5, we obtain the $\texttt{delay}_i$ implementation given in the Appendix. There we have the additional property that $\forall t : \texttt{delay}_i(\mathcal{H}(t))$ is a refinement of $\mathcal{H}(t)$; as a corollary we have that $\mathcal{H}$ is monotonic (i.e. $t_1 < t_2$ implies that $\mathcal{H}(t_2)$ is a refinement of $\mathcal{H}(t_1)$. All the proofs can be found in the Appendix.

5.2 Incremental Fixpoint Computation

We can incrementally refine graphs using the $\texttt{delay}_i$ and $\texttt{merge}_i$ functions, but we also need to do the same to incrementally compute fixpoint maps. For this, we utilize the $\texttt{fix-delay}_i$ and $\texttt{fix-merge}_i$ functions. Consider tuples t and t' such that $t' = \texttt{Succ}_i(t)$ for some i. If we have a fixpoint map η corresponding to t, then $\texttt{fix-delay}_i(\mathcal{H}(t), \eta)$ returns a fixpoint map η' for t', without having to recompute one from scratch as $\texttt{Fix}_{\mathcal{A}}(\mathcal{H}(t'))$. Our method employs the incremental fixpoint computation techniques of Stein et al. [54], recomputing only the parts of the graph $\mathcal{H}(\texttt{Succ}_i(t))$ that are affected by the transformation, while reusing the results of η for the rest. When the analysis operations are monotonic, we maintain the *fixpoint-consistency* property, which ensures that $\texttt{fix-delay}_i(\mathcal{H}(t), \eta) = \texttt{Fix}_{\mathcal{A}}(\mathcal{H}(\texttt{Succ}_i(t)))$ (when the operators are not monotonic, we ensure that $\texttt{fix-delay}_i(\mathcal{H}(t), \eta) \sqsubseteq^\sharp \texttt{Fix}_{\mathcal{A}}(\mathcal{H}(\texttt{Succ}_i(t)))$; this is discussed in Sec. 6). Stein et al. [54] ensures fixpoint consistency by eagerly removing all the abstract states at locations that (transitively) depend on the affected location l, and subsequently recomputes them lazily.

We use a slight variant of Stein et al. [54] to specifically optimize $\texttt{fix-delay}_i$ for GSR without eagerly deleting abstract states at all locations from a point of transformation l. Often happens that even if a location l' is reachable from l, the abstract state at l' does not change (e.g. when a variable was made more precise but is later deallocated). So, we instead *lazily* remove abstract states at the successors of location l' only if the new abstract state d' at l' differs from the old abstract state d. The fixpoint iteration uses the classical worklist algorithm [41] to incrementally compute abstract states. Due to the presence of cycles in CFG, we use Bourdoncle's algorithm [13] to identify widening points and apply widening to enforce convergence.

Algorithm 2: `Fixpoint_Inc`$(t_c, \eta_c, t_b, \eta_b, t'_c)$

Input: t_b, t_c such that $t_b < t_c$, fixpoint maps η_b, η_c

1 **if** $t_c < t'_c$ **then**
2 $\quad t, \eta \leftarrow t_c, \eta_c$;
3 **else**
4 $\quad t, \eta \leftarrow t_b, \eta_b$;
5 $\texttt{diff}_u \leftarrow t'_c - t$;
6 **return** $\bigcirc_{0 \le u \le n} \texttt{fix-delay}_i^{\texttt{diff}_u}(\eta)$;

We now discuss the implementation of the `Fixpoint_Inc` function, which computes a new fixpoint map from existing ones. An important point here is that we only want to apply the $\texttt{fix-delay}_i$ operation, not the $\texttt{fix-merge}_i$ one because only $\texttt{fix-delay}_i$ can force the monotonicity of operators. Thus, we want the difference $\texttt{diff}_u$ between the target tuple t'_c and the starting tuple to only contain positive components. Thus, we test if the previous candidate t_c was smaller than the new one t'_c; if not, we start over the fixpoint computation from the existing best candidate t_b. If $t_c < t'_c$, we have $t_b < t_c < t'_c$, i.e. there are fewer computations to make if we can start from t_c. For example, if we have $t_c = (2, 1, 4)$ and $t'_c = (3, 1, 2)$ and $t_b = (1, 1, 1)$. We don't have $t_c < t'_c$, so we need to start from t_b. In the end we return $\texttt{fix-delay}_0 \circ \texttt{fix-delay}_0 \circ \texttt{fix-delay}_2(\eta_b)$.

To preserve fixpoint-consistency with transformations inside loops, we need to erase and recompute the fixpoint for every location in the loop to compute the precise fixpoint (bottom-up) instead of top-down refining from an imprecise fixpoint [19]. To avoid doing too many fixpoint computations for loops for every $\texttt{fix-delay}_i$ step, we recompute the fixpoint for the locations in the loop only once, when all the delays have been done at the end of each deepening bound in IDDDS.

6 Non-Monotonic Operators

Typical implementations of the abstract operators in practice resort to using sound but non-monotonic operators. In particular, as widening extrapolates the growth of abstract element, it cannot be monotonic, as a more precise argument can result in a new growth to extrapolate. For instance, a typical widening over intervals [19] computes both $[1, 40] \nabla [1, 40] = [1, 40]$ and $[1, 1] \nabla [1, 40] = [1, +\infty]$, which is not monotonic. In the presence of such operators, we no longer have the property that $t_1 < t_2$ implies that the fixpoint results of t_2 is at least as precise as t_1 (see Table 1). Consider the example in Figure 6. Initially, interval analysis after the join at line 2 would compute the value of x as $[1, 60]$. After the assumption $x \le 40$, this interval becomes $[1, 40]$, and within the loop, it continues to stay as $[1, 40]$.

However, when the join ($\sqcup$) at line 2 is delayed by one step leading to the value $x = 60$ being isolated, the only value of x that passes is $[1, 1]$. Once in the loop, the interval expands to $[1, 40]$. However, upon Widening, the result degrades to $[1, +\infty]$ at the loop-head.

```
1 x = 1;
2 if (*) x = 60;
3 assume(x <= 40);
4 while (*) {
5     if (*) {
6         x = 40;
7     }
8 }
```

Fig. 6. Ex. widen.c

Naive solutions to this problem would consist of using abstract domains that have only monotonic operators (i.e., without widening, like the sign, constant propagation, the dependency abstract domain, taint, nullness, etc. domains [19]). These domains are effective because they satisfy the Ascending Chain Condition (ACC) [19], which implies that any increasing sequence of elements eventually stabilizes. In the presence of non-monotonic operators like widening, a naive solution is to discard refinements which result in incomparable/worse precision due to non-monotonicity. In this section, we propose a solution to tackle non-monotonicity without discarding refinements.

A key insight is that incremental computation solves this problem of non-monotonicity of operators and further improves precision. For $t_2 = \texttt{Succ}_i(t_1)$ as shown in Incrementality column in Table 1, we saw $\texttt{fix-delay}_i(\mathcal{H}(t_1), \eta_1) \sqsubseteq^\sharp \texttt{Fix}_\mathcal{A}(\mathcal{H}(t_2))$. We have see that when the operators are monotone, we have the equality. Interestingly, we observe that $\sqsubseteq^\sharp$ holds in the presence of non-monotonicity through *forcing* trick, where $\texttt{fix-delay}_i$ clips the fixpoint computation (especially on loop heads) using intersection $\sqcap$ with the previous fixpoint map η_1 of t_1.

Let d_1, d_2 be the abstract states of a loophead l_{lh} in t_1 and $t_2 = \texttt{Succ}_i(t_1)$ respectively such that $d_2 \not\sqsubseteq^\sharp d_1$. The intuition is that, since d_1 and d_2 are sound-approximation of reachable states at l_{lh}, then $d_1 \sqcap d_2$ continues to be a sound over-approximation at l_{lh} in t_2. $\texttt{fix-delay}_i, \texttt{fix-merge}_i$ are guaranteed to produce better precision when non-monotonic operators produce incomparable results.

More formally, for t_1, t_2 such that $t_1 < t_2$ with fixpoint maps η_{t_1} and η_{t_2}, we define $\mathcal{F}_{delay}$ that computes a new fixpoint map of t_2 from η_{t_1} and η_{t_2} as follows:

$$spawns(w_1, w_2) \stackrel{\text{def}}{=} \forall i \in \{0, ..., n-1\} : (\texttt{sp}(i) \in L_\sqcup \implies w_2[i] \preceq_S w_1[i] \wedge \texttt{sp}(i) \in L_{lh} \implies w_1[i] =_L w_2[i])$$

$$\mathcal{F}_{delay}(\eta_{t_1}, \eta_{t_2})(l, w_2) = \begin{cases} \eta_{t_1}(l, w_1) \sqcap \eta_{t_2}(l, w_2) & \text{if } \exists (l, w_1) : spawns(w_1, w_2) \\ \eta_{t_2}(l, w_2) & \text{otherwise} \end{cases}$$

Similarly, the $\mathcal{F}_{merge}$ operator can be defined in a straightforward way that effectively captures the invariant state. Note that, after forcing operation the invariants obtained are no longer inductive. Hence, care must be taken to ensure the invariants are sound. Having the search exploration path also helps with reproducing these non-inductive invariants.

7 Implementation and Properties of GSR

We have already presented GSR (Algorithm 1) in Sec. 2.5, and we now present a reference implementation of its three components.

Algorithm 3: $\texttt{Next}(t_b, k, \Omega)$

Inputs: Best tuple t_b, max bound k, pruned set Ω.
Globals: Previous explored tuple $\texttt{old\_t}_c$, $set\_dim = \{\}$, set of set_dims $Q = \{\}$, $subsetsize = 0$
Result: A new $t \notin \Omega$ or **None**

1 Restart:
2 $t \leftarrow \texttt{old\_t}_c$;
3 $t' \leftarrow \left(\bigcirc_{i \in set\_dim : t[i] \leq bound} \texttt{Succ}_i \right)(t)$;
4 **if** $t \neq t'$ **then**
5 $\quad t \leftarrow t'$;
6 $\quad$ **if** $t \notin \Omega$ **then return** t;
7 $\quad$ **else goto** Restart;
8 **if** $Q \neq \emptyset$ **then**
9 $\quad$ Remove set_dim from Q;
10 $\quad set\_dim \leftarrow$ Choose from Q;
11 $\quad t \leftarrow t_b$ //Backjumping;
12 $\quad$ **goto** Restart
13 **if** $bound < k$ **then**
14 $\quad bound \leftarrow min(2 * bound, k)$;
15 $\quad Q \leftarrow \{S \subseteq \{0, ..., n-1\} \mid |S| = subsetsize\}$;
16 $\quad$ **goto** Restart
17 **if** $subsetsize < n$ **then**
18 $\quad subsetsize \leftarrow subsetsize + 1$;
19 $\quad bound \leftarrow 2$;
20 $\quad$ **goto** Restart
21 **return None**

Algorithm 4: $\texttt{Minimize}_{\mathcal{A}}(t, \eta)$

Input: Input tuple t and its fixpoint map η
Result: Minimal tuple t and its fixpoint map η

1 **while** $\exists j \in \{0, ..., n-1\} : \eta' = \texttt{fix-merge}_j(t, \eta) \wedge \eta' =^{\sharp}_{\mathcal{L}} \eta$ **do**
2 $\quad t \leftarrow \texttt{Succ}_j^{-1}(t)$;
3 $\quad \eta \leftarrow \eta'$
4 **return** t, η;

Algorithm 5: $\texttt{PruneRegions}(t_c, t_b, \Omega)$

Input: Current and best refinements t_c, t_b such that $t_b \leq t_c$ (requirement), and pruned regions Ω
Result: Set of pruned regions Ω

1 **for** *every* $t \in \Omega$ **do**
2 $\quad$ **if** $t_c > t$ **then**
3 $\quad\quad \Omega \leftarrow$ remove t from Ω;
4 $\quad$ **else if** $t_c \leq t$ **then return** Ω ;
5 $\Omega \leftarrow \Omega \cup \{t_c\}$;
6 **return** Ω;

`Next`. Consider Algorithm 3, which implements two search strategies for selecting the next refinement tuple: Synchronized Delay Search (SDS) and Iterative Deepening Depth-first Delay Search (IDDDS). To support these strategies, we maintain 4 global variables: $subsetsize, bound$, set of subsets Q, and the current subset of dimensions set_dim.

IDDDS begins by incrementally exploring each dimension of the tuple separately up to the initial bound. For instance, starting at $(0, 0, 0)$, it progresses to $(1, 0, 0)$, $(2, 0, 0)$, and cycles through other dimensions like $(0, 1, 0)$ and $(0, 2, 0)$, continuing up to $(0, 0, 2)$. Once the initial bound is reached, the bound is doubled at line 17, allowing for deeper exploration such as $(3, 0, 0)$, $(4, 0, 0)$, $(0, 3, 0)$, and $(0, 4, 0)$. After each dimension has been explored individually up to $(0, 0, k)$, IDDDS expands its focus to include subsets of split-points, applying the same iterative deepening strategy by incrementing the $subsetsize$. Starting with combinations of split-points like $(1, 1, 0)$ and $(2, 2, 0)$, it then increases delays synchronously within these subsets.

For the above example with 3 split-points having indices $\{0, 1, 2\}$, initially $Q = \{\{0\}, \{1\}, \{2\}\}$ is updated at line 15 with $subsetsize = 1$. Then *set_dim* picks each of the subsets from Q and delays all indices within each subset until the first deepening *bound*. Once a subset is explored, `Next` uses backjumping/non-chronological backtracking [50] to backtrack to t_b, and the *set_dim* picks from Q the next subset and all indices of *set_dim* are again synchronously delayed until the *bound*. Once all subsets are explored, Q is empty and then the deepening *bound* is doubled at line 14, and Q is updated again for the same *subsetsize*. Once all the subsets are explored through iterative deepening until $bound = k$, *subsetsize* is incremented and the deepening bound is set to $bound = 2$. The process is again restarted for the updated *subsetsize*. This is repeated until the maximum *subsetsize* where all possible combinations have been explored up to the maximum bound k.

SDS consists of the end of the search performed by IDDDS: by beginning execution with $(subsetsize = n, bound = k, Q = \{\}, set\_dim = \{0, ..., n-1\})$, we enable a traversal that uniformly increases each component of the tuple from $(0, 0, 0)$ to (k, k, k), applying synchronized delays across all dimensions simultaneously.

`Minimize`. The reference implementation shown in Algorithm 4 is quite simple and is the same as described in Sec. 3. It takes a current refinement t_c and tests all the immediate predecessors of t_c using $\texttt{merge}_i$ and $\texttt{fix-merge}_i$, and stops if we cannot find any that maintains the same precision at $\mathcal{L}$, or repeat from this predecessor if we found some. This guarantees finding a minimal element in the precision equivalence class. Note that multiple minimal refinements may exist and the output varies depending on the order of selection of predecessors.

`PruneRegions`. The reference implementation shown in Algorithm 5 adds t_c to Ω which is passed as input so `Next` ensures to not visit a tuple twice. It also performs crucial optimization to keep only maximal elements within the set Ω.

Theorem 1 (GSR Properties). *Let $(t_r, \eta_r) = GSR(P, \mathcal{A}, k, \mathcal{L})$ with reference implementations. The output is*

- **Correct :** It always returns t_r and η_r such that $\eta_r \sqsubseteq^\sharp_{\mathcal{L}} \texttt{Fix}_{\mathcal{A}}(\mathcal{H}(t_r))$
- **Complete :** It always returns a t_{com} with a fixpoint map η_{com} such that $t_{com} \leq t_{max}$ and $\eta_{com} \sqsubseteq^\sharp_{\mathcal{L}} \texttt{Fix}_{\mathcal{A}}(t_{max})$, with $t_{max} = (k, ..., k)$.
- **Minimal :** If operators of $\mathcal{A}$ are monotonic, it returns a t_r, η_r such that $\forall t_i.\, (t_i < t_r \implies \eta_r \widehat{\sqsubseteq}^\sharp_{\mathcal{L}} \texttt{Fix}_{\mathcal{A}}(\mathcal{H}(t_i)))$

Summary. Correctness guarantees precision. In the presence of non-monotonic operators, the resulting η_r depends on the order in which we performed the incremental fixpoint computations during the exploration. If all the operators of $\mathcal{A}$ are monotonic, then GSR with the reference implementations is correct, complete, and minimal. If the operators of $\mathcal{A}$ are not monotonic, then GSR does not preserve minimality. However, GSR is still complete in the presence of non-monotonic operators since it returns a fixpoint map that is at least as precise as the analysis on t_{max}.

8 Experimental Evaluation

The primary goal of this experimental study is to validate the feasibility of our proposal and to evaluate its potential benefits.

Implementation and set up. Our GSR framework is primarily written in OCaml with approximately 10,000 lines of code. Our Incremental analysis logic and $delay_i$ graph transformations operate over an explicit CFG graph representation. Our implementation is parametric in an abstract domain, and the effort required to instantiate the framework to a new abstract domain is comparable to the effort required to do so in a classical abstract interpreter framework. We offer numerical domains such as Constant Propagation, Congruence Domain, and well-known off-the-shelf abstract domains such as Intervals and Octagons from APRON [34]. All experiments are performed on a PC 2.8 GHz and 64GB of RAM, on Ubuntu 22.04 (64-bit).

Standard Configuration. We consider as a split point *every* location l within the CFG that has at least two incoming edges. All the experiments are run with bound $k = 1000$ except the RQ2.a in Sec. 8.1. We choose the Intervals [19] to be our main analysis domain for our experiments (except for RQ1.b). We consider three forms of sensitivity: 1) Control-flow splitting; 2) Loop unrolling; 3) Context-Sensitivity.

Benchmark selection. We take 1126 programs (2924 properties to be checked) from the SVComp software verification benchmark, from four sub-categories of "Reach-safety": control-flow (39 programs, 149 properties), recursive (65 programs, 65 properties), loops (859 programs, 2579 properties), and sequentialized (62 programs, 107 properties). We exclude programs known to be UNSAFE as well as the programs/sub-categories requiring reasoning about arrays, memory, or floats, as our prototype implementation does not support them now. Finally, we also consider 24 variants (24 properties) of $KsHandler\_N.c$, a keyboard shortcut handler function from the Google Closure library highlighted as a hard example for sensitivity in abstract interpretation [37]

Search strategy and baselines. Our search strategy uses IDDDS method. We also consider three baseline strategies: *No-split* represents the fixpoint computation on the original program; *Full-split* applies full trace partitioning up to the allowed bound k; *SDS* mirrors traditional methods like 2-CFA [51,44,45], which analyze all split-points uniformly up to a given depth iteratively.

8.1 Research Questions

We set out to address the following research questions:

RQ1: Feasibility, Efficiency and Genericity. We are interested here in assessing the general feasibility and utility of our approach by **(RQ1.a) Comparison with baselines**, i.e. to understand how the GSR-IDDDS strategy compare in terms of both precision and cost against standard baselines, including no trace partitioning and full trace partitioning. We also seek to demonstrate that our

approach is generic, in that it can be used with different domains (**RQ1.b Domains**) and different kinds of sensitivity (loop unrolling, control-flow splitting, context-sensitivity, and their combination) (**RQ1.c Sensitivity**).

RQ2: Exploration of the Design Space. We then seek to understand better some design points of the method and their practical impact, in order to get insights for further improvements. Especially, we study the **RQ2.a Impact of bound** k, **RQ2.b Impact of incremental solving**, **RQ2.c Impact of proving one property at a time**, **RQ2.d Impact of the minimization phase**, and **RQ2.e Impact of improving several sensitivities at once**.

RQ3: Comparison Against CEGAR Approaches. Finally, we compare our method with the current well-established CEGAR-based refinement tools.

8.2 RQ1: Feasibility, Efficiency and Genericity

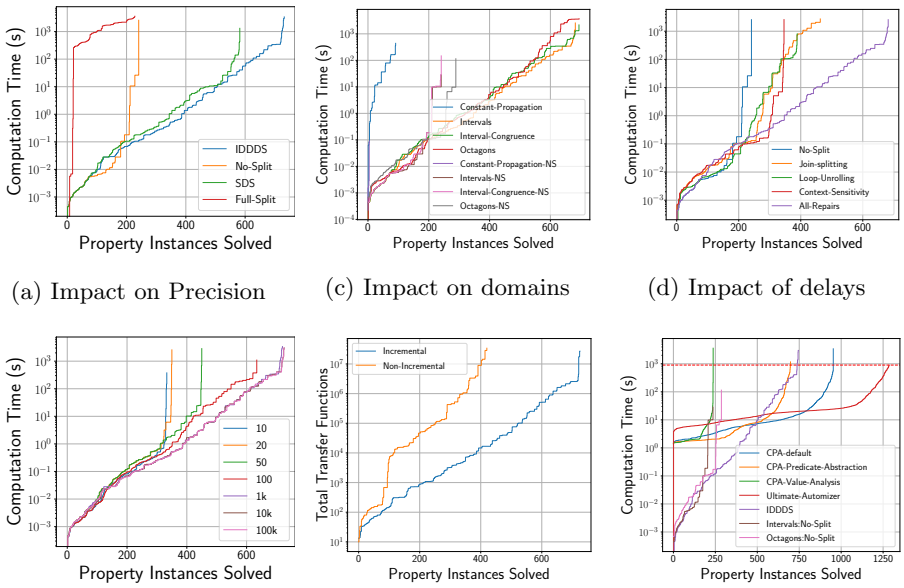
RQ1.a: Comparison with baselines. In Fig.7a, we compared the performance of our IDDDS strategy with three baselines and show the statistics on the total property proved with a timeout of 1 hour. Compared to *No-Split* and *Full-split* strategies (241 and 273 proven properties), IDDDS outperforms them both (732 proven properties). IDDDS also performs better than the *SDS* strategy (582 proven properties). Especially, *SDS* suffers from path explosion on the benchmark sets where there is significant control-flow.

Experiments with IDDDS with bound $k = 1000$ show that large bounds are required for each benchmark category as shown in Fig. 8a, for instance: Control-flow (k=73), Recursive (k=25), Loops (k=101), Sequentialized (k=20), KsHandlers (k=158). Several split-points also need to be delayed for verifying properties for each program. The average number of split-points delayed out of an average total number of split-points are: Control-flow (15/34), Recursive (2/2), Loops (3/11), Sequentialized (2/57), KsHandlers (40/101).

To conclude, we showed that our method indeed improves over all of the No-split, Full-split, and SDS strategies, being able to prove significantly more goals. Also, large bounds k are required on all benchmark categories.

RQ1.b: Domains. Here we study the behavior of our approach on 4 different numerical domains, thereby showing that our approach is independent of the abstract domain being used. Fig.7c shows the clear positive impact of our strategy for each considered domain. For example, for interval domain with 1hr timeout, we go from 241 (No-Split) to 732 successes upon activating splits. For Constant Propagation, we go from 5 to 91. For Octagons, we go from 289 to 695. For Intervals+Congruence, we go from 241 to 733. While the No-Split version Octagons performed better than Intervals No-Split, however, it is interesting to notice that Octogans sometimes time out on programs where Intervals do not due to more expensive domain operations.

By instantiating the algorithm over four numerical domains, we show that our method is generic and brings improvement to different domains.



(a) Impact on Precision (c) Impact on domains (d) Impact of delays

(b) Impact of bound k (f) Impact of incrementality (e) Comparison to CEGAR

Fig. 7. Cactus plots showing the time to completion for solving property instances by different search strategies. The x-axis represents the total number of instances solved. Except for (e) the y-axis (logarithmic scale) corresponds to the time taken in seconds, and as for (f) y-axis (logarithmic scale) corresponds to the total transfer functions required to prove properties. In (c), NS denotes No-Split version

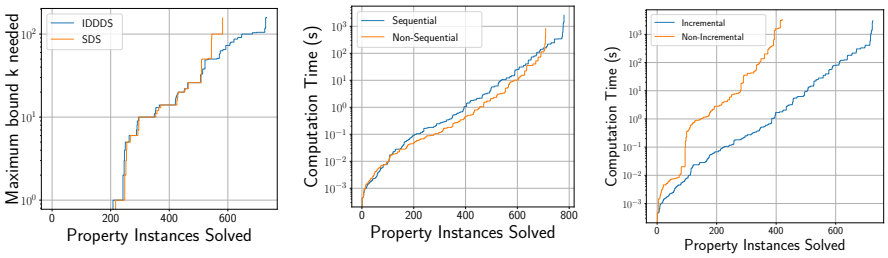
.

RQ1.c: Sensitivity. We show in Fig. 7d that we can handle different kinds of sensitivity (loop unrolling, control-flow splitting, and context-sensitivity), and furthermore, combining all these in a single search instance is very useful. Our experiments showed that a combined method solved 732 instances, surpassing the results of individual techniques: join-splitting with 463 instances, loop-unrolling with 389, and context-sensitivity with 346. Generally, individual strategies explore a restricted search space compared to the combined strategy, missing refinements that could prove properties.

To conclude, we show that we can consider several forms of sensitivity and that combining them proves significantly more properties than using a single kind.

8.3 RQ2: Exploration of the Design Space

RQ2.a: Impact of Bound k. We show in Fig.7b the impact of bound k for various values $k = 10, 20, 50, 100, 1000, 10000, 100000$. First, it is interesting to see that we can afford very large values of k through our search algorithm IDDDS. Interestingly, extra large $k \gg 1000$ does not produce new timeouts compared to the default $k = 1000$, and the time taken to prove properties remains almost the same as shown in the plot irrespective of the bound. Moreover, results show



(a) Maximum bound k needed

(b) Impact of proving one property at a time

(e) Impact of Incrementality (Time)

Fig. 8. a) Cactus plot illustrates the maximum bound k needed to prove properties for all programs. b,c) Cactus plots showing the time to completion for solving property instances by different search strategies: the x-axis represents the total number of instances solved; the y-axis (log scale) corresponds to the time taken in seconds.

indeed that large bounds > 100 are essential for the considered properties, with the optimal bound (in terms of time vs. success) being 158. With the timeout of 1hr, 333 successes for k =10, 350 for k =20, 450 for k =50, 634 for k =100, 732 for k =1000, 10000, and 100k.

To conclude, we show that our search algorithm can indeed afford large bound 'k' without affecting the time taken to prove properties and that such large bounds are indeed necessary on the considered benchmarks.

RQ2.b: Incremental Computation. We used the number of computed abstract transfer functions to measure the impact of incremental computation (which is proportional to the time taken by the analysis). Results in Fig.7f demonstrate the important impact of incremental fixpoint computation, achieving a significant speedup (188x) on transfer functions and fewer timeouts. For a timeout of 1hr and bound $k = 1000$, for solving 2926 instances, we have 732 successes with Incrementality. Moreover, without incrementally we solved only 450 instances while timing out on the rest.

To conclude, we show that the IDDDS algorithm with Incremental computation outperforms the Non-incremental version by several orders of magnitude.

RQ2.c: Impact of proving one property at a time. Here, we assess whether it is preferable to prove all the properties simultaneously (which could remove redundant computations), or to perform a different search for each property, proving them sequentially (with the hope of smaller program refinements for each property). We observe in Fig. 8b that with timeout 1hr and $k = 1000$, while sequential strategy proved 732 instances, non-sequential only solved 660 instances. For these timed-out programs, each property to be proved requires different $\texttt{delay}_i$ operation(s), and hence the only refinement that can prove all properties at once requires all such $\texttt{delay}_i$ operations and leads to path explosion. This is especially observed in Control-flow and KsHandlers categories where there are several properties and several split-points.

To conclude, we show that proving one property at a time works better than proving them all at once, as it avoids path explosion.

RQ2.d: Impact of the minimization. In all the experiments that we performed on the SVComp benchmark, the minimization phase was never able to reduce the program size during the intermediate step. We did check that this minimization worked on the example in Figure 1, as well as in another experiment using *extended tuples* on a challenging example [37]. Normal tuples $\mathbb{N}^n$ solve examples in KsHandlers when proving one property at a time, yet we saw minimization was necessary when proving all properties at the same time.

To conclude, we show that the minimization phase only rarely triggers in practice, but is important on some complex examples.

RQ2.e: Impact of improving several sensitivities at once. In our experiments, we never observed a property that, during the IDDDS search, was proved with $subsetsize = 2$ without being first proved by $subsetsize = 1$ (i.e., that requires several tuple dimensions to be incremented simultaneously). This is good news, as the search space when limiting the search to $subsetsize = 1$ is much smaller than with 2,3, etc. However, these examples exist in theory and in practice: in the Sequentialized category of SVComp, we performed a manual check and saw that programs require $subsetsize = 3$ to verify the property – it cannot be verified with $subsetsize = 1, 2$. An interesting future work is in picking the right choice of locations of interest $\pounds$, as it helps retain delays that could lead to improvements at selected locations and helps avoid needing $subsetsize > 1$.

To conclude, even if there exists programs where several dimensions have to be improved simultaneously, many properties can be proved by limiting the search to improving only one dimension at a time, considerably reducing the search space.

8.4 RQ3: Comparison Against CEGAR Approaches

We consider the two mature and well-optimized software model-checkers Ultimate Automizer and CPA-Checker, overall winners of the SVComp-2024 competition. CPA-Checker includes a different model-checking algorithms like Predicate Abstraction (CPA-PA) [6], Value Analysis by CEGAR [11] (CPA-VA), k-induction and others through its default configuration (CPA). Ultimate Automizer (UA) is based on Trace Abstraction [30].In this experiment, we consider these 4 variants, CPA-PA, CPA-VA, CPA (default), UA.

Results are shown in Fig.7e, and uniquely proved instances are reported in Table 2 as #U (GSR-IDDDS proves the instances and the competitor does not) and #N (the competitor proves the instance and GSR-IDDDS does not). As expected, our early prototype cannot beat the best SVComp competitors, that have been there for several years. Yet, several significant facts can be pointed out. First, our prototype without splitting performs very bad proving only 241 properties, while adding IDDDS dramatically improves its performance by proving 724 properties. Model-checkers CPAChecker proves 953 properties and UA proves 1286 properties. Indeed, our approach is competitive for a time out of 10s (realistic setting in some verification scenarios), and it can still beat CPAChecker with pure predicate abstraction (698 successes) or pure Value Analysis (238 successes) for an overall time out of 900s (SVComp time out). Finally, our method is able to prove some instances that the other approaches cannot.

Table 2. Number of unique instances solved with respect to the state-of-the-art model-checkers. #U: uniquely solved by our method GSR-IDDDS, #N: uniquely solved by competitor against GSR-IDDDS (1126 programs and 2924 properties)

	vs. CPA-PA		vs. CPA-VA		vs. CPA		vs. UA	
Properties	#U	#N	#U	#N	#U	#N	#U	#N
2924	53	22	480	2	11	241	34	591

8.5 Conclusion and Lessons Learned

These experiments demonstrate that our framework can be implemented in a reasonably efficient way for different domains and different sensitivities, and that it already provides a clear value on standard benchmarks Furthermore, these experiments yield interesting practical insights into our general method. In particular, we discovered that:

L1 Incremental computation makes the searching approach feasible (as most fixpoint recomputation only requires a few recomputation steps)

L2 While in theory it is sometimes needed to simultaneously modify several points of the program, improving program points one after the other is generally sufficient and considerably reduces the search space

L3 The iterative deepening strategy allows quickly identifying interesting program improvements without getting stuck, and makes the choice of a bound parameter k unimportant – hence no need for paramter fine-tuning

L4 Combining different kinds of program refinement (delaying joins, loop unrolling, and improved context-sensitivity) is important. The precision in many programs can only improve when those are done simultaneously

L5 Decreasing the program graph size as soon as possible is very important – notably by backtracking on useless splits. On the other hand, our merging algorithm is experimentally rarely useful

L6 When considering different properties, it is better to consider them in isolation (one refinement per property) rather than proving them all at once.

9 Related Work

Completeness in Abstract Interpretation. Giacobazzi et al. first explored abstraction refinement in abstract interpretation [25], but their approach did not offer a direct algorithm for refining abstractions. In contrast, our method can be seen as a specialized way of performing domain refinement, and we provide insight into automation. Classes of complete programs for a given abstraction have been recently studied in a pioneering line of work [24,14,15]. However, all these works seek only to evaluate completeness. In contrast, we progressively repair analysis towards completeness with respect to an abstract domain. Bruni et al. recently proposes [16] to progressively repair its local incompleteness for any given domain. In contrast, our repair strategies are domain-independent as long as we are given implementations for operators $[\![.]\!], \sqcup, [\![.]\!]^*, \nabla$, etc.

Sensitivity through disjunctions. Generic sensitivity techniques based on control-flow are proposed to improve precision through disjunctions in static

analyses such as flow-sensitivity[18], path-sensitivity [32,7,22,23], value-sensitivity [33], trace-sensitivity [42,46,29], views [37], and dynamic partitioning [12]. However, these frameworks are not semantic-directed. Diverse forms of context-sensitivity [47,43,52,35] are proposed to improve precision. The path sensitivity work by Das et al. [22] provides a criterion for which places to split by encoding the property into an automaton while providing a heuristic to merge paths.

Finally, some semantic-directed refinement techniques do exist, yet specialized to particular domains: Li et al [39] for shape analysis, Liang et al [40] for points-to analysis, and other refinement-based approaches [28,53] for pointer analysis. to client queries, but these are specialized for pointer analyses. Bardin et al. [9] also proposes a refinement specialized to binary-level CFG recovery.

Other Refinement Techniques. Efficient algorithmic solutions were first introduced in model-checking by Clarke et al. [17,8,31] as iterative refinements through CEGAR. But these techniques are specialized and should be manually adapted to each domain [6,11,30,8]. Zhang et al [56] present a CEGAR-based technique for finding the optimum abstraction, a cheapest abstraction that proves the query.

Program transformations as a way to improve analysis results has a long history. Some works are based on explicit transformations, such as hot paths isolation [3], loop re-writing techniques [5,48,26,21,27]. Vechev et al. [55] combines both abstraction and program refinement for synchronization synthesis.

10 Conclusion

In this work, we tackle the challenge of automatically tuning trace partitioning for a given program and property, casting the problem as an optimization task and solving it through search-based methods. We identify key difficulties and propose technical solutions to address them (e.g., a new tuple representation of program refinements). Experiments validate the feasibility of our approach.

Future Work. We see several directions for improvements. First, as the framework is domain-independent, integrating new domains poses no theoretical difficulties, and our implementation would clearly benefit from memory-oriented domains. Second, we could try to explore the search space faster through parallelization, as different refinements can be explored in parallel and independently-improved tuples can be easily joined (by taking the maximum of every component to accelerate refinement). Finally, new split-points could be injected on demand, opening our technique to more general sites defined from semantic information (e.g., value set approximation) while keeping a finite branching per split.

Acknowledgments

This work was supported in part by the National Research Agency (grant ANR-22-CE39-0014-03) and France 2030 (grants ANR-22-PECY-0005 and ANR-22-PECY-0007). We are particularly grateful to Rishika Gupta, Alakh Dhruv Chopra, and Ranadeep Biswas, for the support, discussions, and comments.

References

1. Abramsky, S., Gabbay, D.M., Maibaum, T.S.E. (eds.): Handbook of Logic in Computer Science (Vol. 3): Semantic Structures. Oxford University Press, Inc., USA (1995)
2. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison-Wesley series in computer science / World student series edition, Addison-Wesley (1986), https://www.worldcat.org/oclc/12285707
3. Ammons, G., Larus, J.R.: Improving data-flow analysis with path profiles. In: Davidson, J.W., Cooper, K.D., Berman, A.M. (eds.) Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada, June 17-19, 1998. pp. 72–84. ACM, Montreal, Canada (1998). https://doi.org/10.1145/277650.277665, https://doi.org/10.1145/277650.277665
4. Baier, D., Beyer, D., Chien, P., Jankola, M., Kettl, M., Lee, N., Lemberger, T., Rosenfeld, M.L., Spiessl, M., Wachowitz, H., Wendler, P.: Cpachecker 2.3 with strategy selection - (competition contribution). In: TACAS (3). Lecture Notes in Computer Science, vol. 14572, pp. 359–364. Springer (2024)
5. Balakrishnan, G., Sankaranarayanan, S., Ivancic, F., Gupta, A.: Refining the control structure of loops using static analysis. In: Chakraborty, S., Halbwachs, N. (eds.) Proceedings of the 9th ACM & IEEE International conference on Embedded software, EMSOFT 2009, Grenoble, France, October 12-16, 2009. pp. 49–58. ACM, Grenoble, France (2009). https://doi.org/10.1145/1629335.1629343, https://doi.org/10.1145/1629335.1629343
6. Ball, T., Majumdar, R., Millstein, T.D., Rajamani, S.K.: Automatic predicate abstraction of C programs. In: Burke, M., Soffa, M.L. (eds.) Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA, June 20-22, 2001. pp. 203–213. ACM, Utah, USA (2001). https://doi.org/10.1145/378795.378846, https://doi.org/10.1145/378795.378846
7. Ball, T., Rajamani, S.K.: Automatically validating temporal safety properties of interfaces. In: Dwyer, M.B. (ed.) Model Checking Software, 8th International SPIN Workshop, Toronto, Canada, May 19-20, 2001, Proceedings. Lecture Notes in Computer Science, vol. 2057, pp. 103–122. Springer, Toronto, Canada (2001). https://doi.org/10.1007/3-540-45139-0_7, https://doi.org/10.1007/3-540-45139-0_7
8. Ball, T., Rajamani, S.K.: The SLAM project: debugging system software via static analysis. In: Launchbury, J., Mitchell, J.C. (eds.) Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002. pp. 1–3. ACM, Portland, OR, USA (2002). https://doi.org/10.1145/503272.503274, https://doi.org/10.1145/503272.503274
9. Bardin, S., Herrmann, P., Védrine, F.: Refinement-based CFG reconstruction from unstructured programs. In: Jhala, R., Schmidt, D.A. (eds.) Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6538, pp. 54–69. Springer, Austin, TX, USA (2011). https://doi.org/10.1007/978-3-642-18275-4_6, https://doi.org/10.1007/978-3-642-18275-4_6

10. Beyer, D.: Competition on software verification and witness validation: SV-COMP 2023. In: Sankaranarayanan, S., Sharygina, N. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Paris, France, April 22-27, 2023, Proceedings, Part II. Lecture Notes in Computer Science, vol. 13994, pp. 495–522. Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_29, https://doi.org/10.1007/978-3-031-30820-8_29
11. Beyer, D., Löwe, S.: Explicit-state software model checking based on CEGAR and interpolation. In: Cortellessa, V., Varró, D. (eds.) Fundamental Approaches to Software Engineering - 16th International Conference, FASE 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7793, pp. 146–162. Springer, Rome, Italy (2013). https://doi.org/10.1007/978-3-642-37057-1_11, https://doi.org/10.1007/978-3-642-37057-1_11
12. Bourdoncle, F.: Abstract interpretation by dynamic partitioning. J. Funct. Program. **2**(4), 407–423 (1992). https://doi.org/10.1017/S0956796800000496, https://doi.org/10.1017/S0956796800000496
13. Bourdoncle, F.: Efficient chaotic iteration strategies with widenings. In: Bjørner, D., Broy, M., Pottosin, I.V. (eds.) Formal Methods in Programming and Their Applications, International Conference, Akademgorodok, Novosibirsk, Russia, June 28 - July 2, 1993, Proceedings. Lecture Notes in Computer Science, vol. 735, pp. 128–141. Springer (1993). https://doi.org/10.1007/BFb0039704, https://doi.org/10.1007/BFb0039704
14. Bruni, R., Giacobazzi, R., Gori, R., Garcia-Contreras, I., Pavlovic, D.: Abstract extensionality: on the properties of incomplete abstract interpretations. Proc. ACM Program. Lang. **4**(POPL), 28:1–28:28 (2020). https://doi.org/10.1145/3371096, https://doi.org/10.1145/3371096
15. Bruni, R., Giacobazzi, R., Gori, R., Ranzato, F.: A logic for locally complete abstract interpretations. In: 36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021, Rome, Italy, June 29 - July 2, 2021. pp. 1–13. IEEE, Rome, Italy (2021). https://doi.org/10.1109/LICS52264.2021.9470608, https://doi.org/10.1109/LICS52264.2021.9470608
16. Bruni, R., Giacobazzi, R., Gori, R., Ranzato, F.: Abstract interpretation repair. In: Jhala, R., Dillig, I. (eds.) PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022. pp. 426–441. ACM, CA, USA (2022). https://doi.org/10.1145/3519939.3523453, https://doi.org/10.1145/3519939.3523453
17. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings. Lecture Notes in Computer Science, vol. 1855, pp. 154–169. Springer, Chicago, IL, USA (2000). https://doi.org/10.1007/10722167_15, https://doi.org/10.1007/10722167_15
18. Cousot, P.: Semantic foundations of program analysis. In: Program flow analysis: theory and applications, pp. 303–342. Prentice Hall, new jersey (1981)
19. Cousot, P.: Principles of Abstract Interpretation. MIT Press, Cambridge, Massachusetts (2021)

20. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Graham, R.M., Harrison, M.A., Sethi, R. (eds.) Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977. pp. 238–252. ACM, California, USA (1977). https://doi.org/10.1145/512950.512973, https://doi.org/10.1145/512950.512973
21. Cyphert, J., Breck, J., Kincaid, Z., Reps, T.W.: Refinement of path expressions for static analysis. Proc. ACM Program. Lang. **3**(POPL), 45:1–45:29 (2019). https://doi.org/10.1145/3290358, https://doi.org/10.1145/3290358
22. Das, M., Lerner, S., Seigle, M.: ESP: path-sensitive program verification in polynomial time. In: Knoop, J., Hendren, L.J. (eds.) Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002. pp. 57–68. ACM, Berlin, Germany (2002). https://doi.org/10.1145/512529.512538, https://doi.org/10.1145/512529.512538
23. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for java. In: Knoop, J., Hendren, L.J. (eds.) Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002. pp. 234–245. ACM, Berlin, Germany (2002). https://doi.org/10.1145/512529.512558, https://doi.org/10.1145/512529.512558
24. Giacobazzi, R., Logozzo, F., Ranzato, F.: Analyzing program analyses. In: Rajamani, S.K., Walker, D. (eds.) Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015. pp. 261–273. ACM, Mumbai, India (2015). https://doi.org/10.1145/2676726.2676987, https://doi.org/10.1145/2676726.2676987
25. Giacobazzi, R., Ranzato, F., Scozzari, F.: Making abstract interpretations complete. J. ACM **47**(2), 361–416 (2000). https://doi.org/10.1145/333979.333989, https://doi.org/10.1145/333979.333989
26. Gulwani, S., Jain, S., Koskinen, E.: Control-flow refinement and progress invariants for bound analysis. In: Hind, M., Diwan, A. (eds.) Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009. pp. 375–385. ACM, Dublin, Ireland (2009). https://doi.org/10.1145/1542476.1542518, https://doi.org/10.1145/1542476.1542518
27. Gulwani, S., Zuleger, F.: The reachability-bound problem. In: Zorn, B.G., Aiken, A. (eds.) Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010. pp. 292–304. ACM, Toronto, Ontario (2010). https://doi.org/10.1145/1806596.1806630, https://doi.org/10.1145/1806596.1806630
28. Guyer, S.Z., Lin, C.: Client-driven pointer analysis. In: Cousot, R. (ed.) Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings. Lecture Notes in Computer Science, vol. 2694, pp. 214–236. Springer, CA, USA (2003). https://doi.org/10.1007/3-540-44898-5_12, https://doi.org/10.1007/3-540-44898-5_12
29. Handjieva, M., Tzolovski, S.: Refining static analyses by trace-based partitioning using control flow. In: Levi, G. (ed.) Static Analysis, 5th Inter-

national Symposium, SAS '98, Pisa, Italy, September 14-16, 1998, Proceedings. Lecture Notes in Computer Science, vol. 1503, pp. 200–214. Springer, Pisa, Italy (1998). https://doi.org/10.1007/3-540-49727-7_12, https://doi.org/10.1007/3-540-49727-7_12

30. Heizmann, M., Hoenicke, J., Podelski, A.: Refinement of trace abstraction. In: Palsberg, J., Su, Z. (eds.) Static Analysis, 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5673, pp. 69–85. Springer, CA, USA (2009). https://doi.org/10.1007/978-3-642-03237-0_7, https://doi.org/10.1007/978-3-642-03237-0_7
31. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Launchbury, J., Mitchell, J.C. (eds.) Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002. pp. 58–70. ACM, Portland,OR, USA (2002). https://doi.org/10.1145/503272.503279, https://doi.org/10.1145/503272.503279
32. Holley, L.H., Rosen, B.K.: Qualified data flow problems. IEEE Trans. Software Eng. **7**(1), 60–78 (1981). https://doi.org/10.1109/TSE.1981.234509, https://doi.org/10.1109/TSE.1981.234509
33. Jeannet, B.: Dynamic partitioning in linear relation analysis: Application to the verification of reactive systems. Formal Methods Syst. Des. **23**(1), 5–37 (2003). https://doi.org/10.1023/A:1024480913162, https://doi.org/10.1023/A:1024480913162
34. Jeannet, B., Miné, A.: Apron: A library of numerical abstract domains for static analysis. In: Bouajjani, A., Maler, O. (eds.) Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5643, pp. 661–667. Springer, Grenoble (2009). https://doi.org/10.1007/978-3-642-02658-4_52, https://doi.org/10.1007/978-3-642-02658-4_52
35. Kastrinis, G., Smaragdakis, Y.: Hybrid context-sensitivity for points-to analysis. In: Boehm, H., Flanagan, C. (eds.) ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013. pp. 423–434. ACM, Seattle, USA (2013). https://doi.org/10.1145/2491956.2462191, https://doi.org/10.1145/2491956.2462191
36. Kildall, G.A.: A unified approach to global program optimization. In: Fischer, P.C., Ullman, J.D. (eds.) Conference Record of the ACM Symposium on Principles of Programming Languages, Boston, Massachusetts, USA, October 1973. pp. 194–206. ACM Press, Massachusetts, USA (1973). https://doi.org/10.1145/512927.512945, https://doi.org/10.1145/512927.512945
37. Kim, S., Rival, X., Ryu, S.: A theoretical foundation of sensitivity in an abstract interpretation framework. ACM Trans. Program. Lang. Syst. **40**(3), 13:1–13:44 (2018). https://doi.org/10.1145/3230624, https://doi.org/10.1145/3230624
38. Korf, R.E.: Depth-first iterative-deepening: An optimal admissible tree search. Artif. Intell. **27**(1), 97–109 (1985). https://doi.org/10.1016/0004-3702(85)90084-0, https://doi.org/10.1016/0004-3702(85)90084-0
39. Li, H., Berenger, F., Chang, B.E., Rival, X.: Semantic-directed clumping of disjunctive abstract states. In: Castagna, G., Gordon, A.D. (eds.) Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Lan-

guages, POPL 2017, Paris, France, January 18-20, 2017. pp. 32–45. ACM, Paris, France (2017). https://doi.org/10.1145/3009837.3009881, https://doi.org/10.1145/3009837.3009881

40. Liang, P., Tripp, O., Naik, M.: Learning minimal abstractions. In: Ball, T., Sagiv, M. (eds.) Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011. pp. 31–42. ACM, Austin, TX, USA (2011). https://doi.org/10.1145/1926385.1926391, https://doi.org/10.1145/1926385.1926391
41. Martin, F.: Generating program analyzers. Ph.D. thesis, Saarland University, Saarbrücken, Germany (1999), http://scidok.sulb.uni-saarland.de/volltexte/2004/203/index.html
42. Mauborgne, L., Rival, X.: Trace partitioning in abstract interpretation based static analyzers. In: Sagiv, S. (ed.) Programming Languages and Systems, 14th European Symposium on Programming,ESOP 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3444, pp. 5–20. Springer, Edinburgh, UK (2005). https://doi.org/10.1007/978-3-540-31987-0_2, https://doi.org/10.1007/978-3-540-31987-0_2
43. Milanova, A.L., Rountev, A., Ryder, B.G.: Parameterized object sensitivity for points-to analysis for java. ACM Trans. Softw. Eng. Methodol. **14**(1), 1–41 (2005). https://doi.org/10.1145/1044834.1044835, https://doi.org/10.1145/1044834.1044835
44. Rival, X.: Understanding the origin of alarms in astrée. In: Hankin, C., Siveroni, I. (eds.) Static Analysis, 12th International Symposium, SAS 2005, London, UK, September 7-9, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3672, pp. 303–319. Springer, London, UK (2005). https://doi.org/10.1007/11547662_21, https://doi.org/10.1007/11547662_21
45. Rival, X., Mauborgne, L.: The trace partitioning abstract domain. ACM Trans. Program. Lang. Syst. **29**(5), 26 (2007). https://doi.org/10.1145/1275497.1275501, https://doi.org/10.1145/1275497.1275501
46. Sankaranarayanan, S., Ivancic, F., Shlyakhter, I., Gupta, A.: Static analysis in disjunctive numerical domains. In: Yi, K. (ed.) Static Analysis, 13th International Symposium, SAS 2006, Seoul, Korea, August 29-31, 2006, Proceedings. Lecture Notes in Computer Science, vol. 4134, pp. 3–17. Springer, Seoul, Korea (2006). https://doi.org/10.1007/11823230_2, https://doi.org/10.1007/11823230_2
47. Sharir, M., Pnueli, A., et al.: Two approaches to interprocedural data flow analysis. New York University. Courant Institute of Mathematical Sciences . . . , New York (1978)
48. Sharma, R., Dillig, I., Dillig, T., Aiken, A.: Simplifying loop invariant generation using splitter predicates. In: Gopalakrishnan, G., Qadeer, S. (eds.) Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6806, pp. 703–719. Springer, Snowbird, UT, USA (2011). https://doi.org/10.1007/978-3-642-22110-1_57, https://doi.org/10.1007/978-3-642-22110-1_57
49. Shivers, O.G.: Control-flow analysis of higher-order languages or taming lambda. Carnegie Mellon University (1991)
50. Silva, J.P.M., Sakallah, K.A.: GRASP - a new search algorithm for satisfiability. In: Rutenbar, R.A., Otten, R.H.J.M. (eds.) Proceedings of the 1996 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 1996, San Jose, CA,

USA, November 10-14, 1996. pp. 220–227. IEEE Computer Society / ACM (1996). https://doi.org/10.1109/ICCAD.1996.569607, https://doi.org/10.1109/ICCAD.1996.569607

51. Smaragdakis, Y., Balatsouras, G.: Pointer analysis. Found. Trends Program. Lang. **2**(1), 1–69 (2015). https://doi.org/10.1561/2500000014, https://doi.org/10.1561/2500000014
52. Smaragdakis, Y., Bravenboer, M., Lhoták, O.: Pick your contexts well: understanding object-sensitivity. In: Ball, T., Sagiv, M. (eds.) Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011. pp. 17–30. ACM, USA (2011). https://doi.org/10.1145/1926385.1926390, https://doi.org/10.1145/1926385.1926390
53. Sridharan, M., Bodík, R.: Refinement-based context-sensitive points-to analysis for java. In: Schwartzbach, M.I., Ball, T. (eds.) Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006. pp. 387–400. ACM, Canada (2006). https://doi.org/10.1145/1133981.1134027, https://doi.org/10.1145/1133981.1134027
54. Stein, B., Chang, B.E., Sridharan, M.: Demanded abstract interpretation. In: Freund, S.N., Yahav, E. (eds.) PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021. pp. 282–295. ACM, Canada (2021). https://doi.org/10.1145/3453483.3454044, https://doi.org/10.1145/3453483.3454044
55. Vechev, M.T., Yahav, E., Yorsh, G.: Abstraction-guided synthesis of synchronization. In: Hermenegildo, M.V., Palsberg, J. (eds.) Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010. pp. 327–338. ACM, Madrid, Spain (2010). https://doi.org/10.1145/1706299.1706338, https://doi.org/10.1145/1706299.1706338
56. Zhang, X., Naik, M., Yang, H.: Finding optimum abstractions in parametric dataflow analysis. In: Boehm, H., Flanagan, C. (eds.) ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013. pp. 365–376. ACM, Seattle, WA (2013). https://doi.org/10.1145/2491956.2462185, https://doi.org/10.1145/2491956.2462185