Identifying Minimal Changes in the Zone Abstract Domain

Kenny Ballou[0000−0002−6032−474X] and Elena Sherman[0000−0003−4522−9725]

Boise State University
kennyballou@u.boisestate.edu,elenasherman@boisestate.edu

Abstract. Verification techniques express program states as logical formulas over program variables. For example, symbolic execution and abstract interpretation encode program states as a set of linear integer inequalities. However, for real-world programs these formulas tend to become large, which affects scalability of analyses. To address this problem, researchers developed complementary approaches which either remove redundant inequalities or extract a subset of inequalities sufficient for specific reasoning, *i.e.,* formula slicing. For arbitrary linear integer inequalities, such reduction approaches either have high complexities or over-approximate. However, efficiency and precision of these approaches can be improved for a restricted type of logical formulas used in relational numerical abstract domains. While previous work investigated custom efficient redundant inequality elimination for Zones states, our work examines custom semantic slicing algorithms that identify a minimal set of changed inequalities in Zones states.
The client application of the minimal changes in Zones is an empirical study on comparison between invariants computed by data-flow analysis using Zones, Intervals and Predicates numerical domains. In particular, evaluations compare how our proposed algorithms affect the precision of comparing Zones vs. Intervals and Zones vs. Predicates abstract domains. The results show our techniques reduce the number of variables by more than 70% and the number of linear inequalities by 30%, comparing to those of full states. The approach refines the granularity of comparison between domains, reducing incomparable invariants between Zones and Predicates from 52% to 4%, and increases equality of Intervals and Zones, invariants from 27% to 71%. Finally, the techniques improve the comparison efficiency by reducing total runtime for all subject comparisons for Zones and Predicates from over four minutes to a few seconds.

Keywords: Abstract domains · Abstract interpretation · Static analysis · Program analysis

1 Introduction

Many verification techniques express a program state as a logical formula over program variables. For example, symbolic execution uses a logical formula to

[0]Author generated copy

describe a path constraint; in abstract interpretation, relational domains such as Zones or Polyhedra use a set of linear integer inequalities to describe program invariants. While expressive, this logical representation can become quite difficult to handle efficiently. For example, when symbolic execution traverses deep paths, or when relational domains encode numerous program variables. The increase in formula size causes verification tasks to run out of memory or timeout. Thus, to improve scalability, verification techniques need to efficiently handle these large logical formulas.

To overcome this predicament, researchers consider two complementary approaches. The first one focuses on eliminating the number of redundant constraints using techniques such as Motzkin-Chernikova-Le Verge [6,21] algorithm for linear integer inequalities. Previous work, for example, used it to minimize path constraints in symbolic execution [22].

The second approach focuses on identifying a minimal set of constraints necessary to reason about a specific task, for example, identifying a set of linear inequalities affected by state change. Green [33] performs the slicing operation on a path constraint formula to determine a set of linear inequalities affected by a newly added constraint. Identifying relevant constraints reduces the query size sent to an SMT solver to check for satisfiability.

However, these minimization techniques assume a general format of linear inequalities, which for a logical formula over linear integer inequalities of restricted types, may incur complexity cost or obtain a non-optimal solution. In abstract interpretation, most popular abstract relational domains such as Zones [24] or Octagons [26] restrict the type of linear inequalities they encode. Researches noted that Motzkin-Chernikova-Le Verge algorithm has a high complexity and in some cases exponential complexity [34] when applied to eliminate redundant linear inequalities. Leveraging the efficient encoding for the Zones domain, Larsen *et al.* [19] developed a more efficient algorithm which removes redundant constraints, with cubic complexity with respect to the number of program variables.

The slicing technique proposed in Green uses syntax-based rules to compute transitive dependencies of constraints. While sound, this approach might over-approximate the set of affected linear inequalities. Applying a precise "semantic-based" slicing for a general linear inequality is a difficult problem. However, as this work shows for Zones domain, it reduces to quadratic complexity. In this work we propose several specialized algorithms for computing a minimal changed set of linear inequalities for the Zones domain. For efficient encodings and operations, relational numerical domains use rewriting rules [28] to convert linear constraints into a canonical form. We also identify challenges such a canonical representation causes in identifying minimal changes in an abstract state.

We evaluate our approach in the context of a data-flow analysis (DFA) framework [18], where abstract interpretation computes invariants over program variables. Researchers in areas such as program verification [5,35] or program optimization [1,17] use the computed invariants to accomplish their respective goals.

The goal is to improve the precision of comparing invariants of Zones against ones of Interval and Predicate abstract domains by comparing only the part of

Zone state that changed. The importance of empirically evaluating domains has been suggested previously [25] since domains differ in their expressiveness and efficiency, and thus, finding an optimal domain is an important problem.

Evaluating our techniques to study the difference between incomparable domains, *e.g.,* Zones and Predicate domains [14], allows us to determine its effect on decreasing the number of incomparable comparisons results. For example, Zones can compute more precise values for some variables at the beginning of a method, but later on, Predicates domain computes more precise values for another set of variables. If analyses are compared using the entire state of each, then results would be incomparable for the later part. However, using our approach, the comparison would indicate that the Predicates domain computes more precise invariants in the latter part of the program.

Our main contributions for this work are:

- A problem definition and a collection of efficient algorithms to identify minimal changes for the Zones abstract domain.
- A demonstration of the effectiveness of our techniques at increasing precision when comparing Zones to comparable and incomparable domains. Similarly, demonstration that our techniques improve efficiency of domain comparison.

2 Background and Motivation

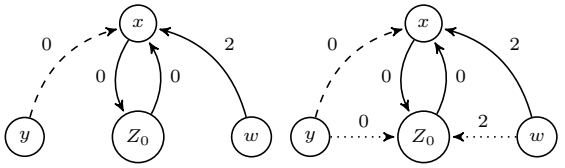
We illustrate problems with finding the minimal changed set of linear inequalities for the Zones domain on a code example in Figure 1a and focus on changes to the abstract state after taking the true branch in statement 4, *i.e.,* changes to the incoming state of 4 to the outgoing state of the true branch of state 4.

To better conceptualize the idea of the minimal changed state, we first consider abstract states computed by an analyzer over the Intervals [8] domain. The incoming state is $x \mapsto [0, 0], w \mapsto (-\infty, 2], y \mapsto \top$, where the interval for x comes from line 2; w comes from taking the true branch on line 3; and y, at this point, is unbounded. After applying the transfer function for the true branch of line 4, the analyzer updates the value of y to $(-\infty, 0]$ without affecting values of x and w. To identify changed variables, the analyzer simply checks the difference in updated variable values between the two states since changes to one variable do not induce changes in other variables.

2.1 Finding a Minimal Subset in Relational Domains

For a relational numerical domain, the analyzer produces the formula $x = 0 \wedge w - x \leq 2$ for the incoming state to line 4, the absence of the y variable means it is a free variable. Interpreting the true branch of 4, the analyzer introduces: $y - x \leq 0$, resulting in a new outgoing state: $x = 0 \wedge w - x \leq 2 \wedge y - x \leq 0$. Here, the minimal subset of inequalities contains $x = 0$ and $y - x \leq 0$. Indeed, only these two inequalities are sufficient to reason about the changed part of the state.

```
1 int example(int w, int y) {
2   int x = 0;
3   if (w <= x + 2) {
4     if (y <= x) {
5       assert y <= 0;
6     }
7   }
8   return x;
9 }
```



(a) Example program (b) Zone state (c) Fully closed Zone state

Fig. 1: Example program and two equivalent Zones

Such a minimal subset is not easily identifiable. Green's slicing technique over-approximates it by including all inequalities through a syntax-based transitive closure. However, since x has not changed and is only used in $y - x \leq 0$ to restrict y's values, *no changes occurred with respect to* w, and hence, the $w - x \leq 2$ inequality remains the same. In our work, we show algorithms for reasoning about such "semantic-based" slicing for the Zones domain. Before we provide an overview of our approach, we present a brief background on the Zones domain.

2.2 Zones Domain

The Zones abstract domain expresses specific relations between program variables. Zones limits its relations to unitary difference inequalities such as $x-y \leq b$, and interval inequalities such as $x \leq b$ and $x \geq b$, where $b \in \mathbb{Z}$. Equality relations are rewritten into a pair of two inequalities. For example, an $x = y + b$ relation results in $x - y \leq b$ and $y - x \leq -b$. To encode an interval value such as $x = [1, 2]$, Zones use the following inequalities $x \geq 1$ and $x \leq 2$. To encode interval valued variables as unitary difference constraints, Zones introduces a special "zero" variable, denoted here as Z_0, and rewriting rules. These rules change the inequalities for the interval $[1, 2]$ into $x - Z_0 \leq 2$ and $Z_0 - x \leq -1$ where the value of Z_0 always equals 0. In this way, the Zones domain represents all inequality constraints in a uniform $x - y \leq b$ template.

Inequalities in the Zones domain have an isomorphic representation as a weighted, directed graph, which is efficiently encoded as a 2D matrix [9,24]. In this graph, variables are nodes, the source and the sink of an edge identify variables in the first and the second positions of the difference template, respectively, and the weight is the coefficient b.

To illustrate this representation, consider the graph in Figure 1b, which encodes the constraints from the running example. The solid lines denote the inequalities of the incoming state to statement 4; and the dashed line is the additional inequality after interpreting the true branch of line 4. Here, the edge from x to Z_0 is for $x - Z_0 \leq 0$ and the edge from Z_0 to x is for $Z_0 - x \leq 0$ of the rewritten $x = 0$ term. The edge from w to x represents $w - x \leq 2$. Similarly, the dashed edge from y to x represents the additional $y - x \leq 0$ term.

2.3 Finding Affected Inequalities

In graph terms, the problem reduces to finding affected edges when an edge is changed in the state graph. Naively, all edges reachable from the changed variables are affected, *i.e.,* a connected component (with undirected edges).

However, we should exclude nodes connected only through Z_0 as a part of the connected component since it might introduce *spurious* dependencies. Consider, the state also has an edge from a variable w to Z_0 with weight 2, which encodes inequality $w \leq 2$ represented as $w - Z_0 \leq 2$ in the canonical unitary difference form. If Z_0 is treated as a regular variable, then the technique would include $w - Z_0 \leq 2$ in the set of affected inequalities. Our approach considers Z_0 a singularity and stops at Z_0 when it transitively computes predecessors R^- and successors R^+ of a node, for example, to identify connected components.

To improve on the connected component approach, our algorithms reason about the directions of the edges in the graph. The outgoing edges from one node to another means the second one restricts the first. In our example, w has an outgoing edge to x, meaning x restricts w. The only changes that can be propagated to w from an additional inequality, if x's value would be further restricted, manifests as a new or updated outgoing edge. However, after adding $y - x \leq 0$ inequality (the dashed edge), the only new outgoing transitions are associated with y. Thus, in order to determine the affected inequalities we transitively compute the predecessors and successors of y, *i.e.,* $R^-(y) = \{y\}$ and $R^+(y) = \{x, y, Z_0\}$. The union of the two sets results in the subgraph of dependent variables to y. In the next section we present formal algorithms and prove their correctness.

2.4 Dealing with Spurious Inequalities

Relational numerical domains possess a powerful ability to infer new relations between variables. In the Zones domain, it happens through computing transitive closures for each variable. For example, since w can reach Z_0, one can establish that $w \leq 2$. Later, when the x variable gets reassigned, $w \leq 2$ remains in place. Since inequalities describing a bounded region are not unique, Zones require a canonical representation to enable operations such as equality comparisons. In order to determine whether two Zones are equal, as needed in DFA's fixed-point algorithms, these states should be *fully closed*. That is, all inferred, transitive constraints should be explicitly stated. Figure 1c shows the fully closed version of the same graph as in Figure 1b, where dotted lines depict inferred edges.

Applying the previously described technique identifies three inequalities $y \leq x$, $x = 0$ and $y \leq 0$. Clearly, the first two constraints do not add any additional information to the third. In fact, after closing, the newly added edge (y, x) becomes what we call a *spurious* edge since it falsely implies that y and x make up a relational constraint, while in fact, since x is a constant, this is not the case.

Therefore, before finding affected inequalities, we identify such spurious edges and remove them from the graph. The algorithm checks for a given connected component containing changed variables whether an edge between two nodes

connected to Z_0 carries additional information, otherwise it removes the direct edge. In our example, the connected component containing potentially changed variables are all nodes with an edge to Z_0.

In our example, the algorithm removes edges (y, x) and (w, x). Now, node x is no longer reachable from y, making $R^+(y) = \{y, Z_0\}$. For a fully closed state the algorithm determines a single affected inequality: $y \leq 0$. Thus, removing spurious constraints helps with identifying smaller, truly connected components.

3 Finding the Minimal Changed Set of Inequalities

In this section, we first formally define the problem of finding a minimal changed set of inequalities from Zones. We continue with a series of algorithms starting with spurious constraint elimination, followed by different minimization approaches for arbitrary and fully closed Zones.

3.1 Problem Definition

Let $\mathcal{N}_1$ and $\mathcal{N}_2$ be sets of inequalities in the initial and updated states, respectively. An inequality n of a Zone state can be uniquely identified by its first n_1 and the second n_2 variables in the unitary difference formula template. Let u be the node representing n_1 and similarly $w = n_2$, then the corresponding edge n in the state graph has u as its source and w as its target nodes.

For a given variable v, we define its dependent inequalities in $\mathcal{N}_1$ as

$$S_1 = \{n \in \mathcal{N}_1 \mid n_1 \in R^-_{\mathcal{N}_1}(v) \vee n_2 \in R^+_{\mathcal{N}_1}(v)\}$$

where $R^-_{\mathcal{N}_1}(v) = P(v) \cup \{v\}$, and similarly, $R^+_{\mathcal{N}_1}(v) = S(v) \cup \{v\}$. S_2 is dually constructed for $\mathcal{N}_2$.

Let dv be a nonempty set of updated variables that change $\mathcal{N}_1$ to $\mathcal{N}_2$. Then, the problem of finding the minimal changed set of inequalities is equivalent to finding the smallest $S \subseteq \mathcal{N}_2$ such that

$$S \cap \{n \in \mathcal{N}_2 \mid \forall v \in dv : n_1 \in R^-_{\mathcal{N}_2}(v) \vee n_2 \in R^+_{\mathcal{N}_2}(v)\} \neq \emptyset \text{ and } S_2 \setminus_{id} S \Leftrightarrow S_1 \setminus_{id} S$$

Here, the first term ensures that the set S includes only affected inequalities. The second term ensures that the set of remaining inequalities after removing updated ones should be logically equivalent between the initial and updated states. The equality in the set minus operation is determined by inequality IDs, *i.e.,* n_1 and n_2. That is, if two inequalities have the same order of variables, then they are considered equivalent for set operations.

3.2 Minimization Algorithms

A straightforward solution to identify such an S would be to compare $\mathcal{N}_1$ and $\mathcal{N}_2$ directly, which would require an exhaustive search. Instead, our approaches only take $\mathcal{N}_2$ as a graph Z, a set of updated variables dv, and a set of updated edges

Algorithm 1 Minimal Changed Set

```
1: function MinChangedSet(Z, dv, de, Method)
2:     G ← RemoveSpurious(Z)
3:     switch Method do
4:         case CC
5:             G ← ConnectedComponents(G, dv)
6:         case NN
7:             G ← NodeNeighbors(G, dv)
8:         case MN
9:             G ← MinimalNeighbors(G, de)
10:    return G
11: end function
```

de. A DFA framework can directly provide the sets *dv* and *de* when it invokes a transfer function. Using these three input values and a choice of minimization $Method$, the pseudocode in Algorithm 1 computes the smallest set of changed inequalities which it returns as a graph G.

On line 2, the algorithm invokes `RemoveSpurious` on the updated state Z. The purpose of this method is to remove spurious dependencies in Z inferred through Z_0, thus creating smaller connected components in G. Next, on line 3, the algorithm switches on $Method$. The rest of the algorithm computes minimal changed sets based on the method selected: CC approach– `ConnectedComponents`; NN approach– `NodeNeighbors`, and lastly MN– `MinimalNeighbors`. These algorithms approximate S differently and have different computational complexity, which are varied based on the Zones representation. Gs of these algorithms create a total order on the number of inequalities in the reduced state $G_{CC} \preceq_{|E|} G_{NN} \preceq_{|E|} G_{MN}$. The runtime complexity of Algorithm 1 is $O(n^2)$,[1] which is dominated by the quadratic complexity of `RemoveSpurious`.

Spurious Connections The goal of the spurious constraint removal step is to deal with inferences through Z_0. Our approach for identifying spurious edges is a special case of the reduction proposed by Larsen *et al.* [19]. That is, an edge between two nodes can be removed from a Zone if the weight between them is greater or equal to any path between them. Instead of applying this reduction to the entire state, our algorithm considers only the path *through* Z_0. This reduces the runtime complexity from $O(n^3)$ to $O(n^2)$. We define a spurious directed edge between s and t variables when $(s,t) \geq (s,Z_0) + (Z_0,t)$.

Algorithm 2 details the steps for this spurious edge removal. The algorithm determines candidate pairs by selecting variables with connections to or from Z_0, line 1. Nodes not connected to the zero node can be excluded because any edge is, by definition, non-spurious. It iterates over the candidate node pairs, line 9 and for each pair, it checks the spurious edge criterion on line 10. If the criterion is satisfied, the edge is removed, line 11. The correctness of the algorithm comes

[1]The average is usually less due to sparsity in graphs.

Algorithm 2 Removal of spurious dependences in G

```
1: function RemoveSpurious(G)
2:     C ← {}
3:     for s ∈ V(G) do
4:         if (s, Z_0) ≠ ⊤ ∨ (Z_0, s) ≠ ⊤ then
5:             C ← C ∪ {s}
6:         end if
7:     end for
8:     for s ∈ C do
9:         for t ∈ C do
10:            if s ≠ t ∧ (s, t) ≥ (s, Z_0) + (Z_0, t) then
11:                (s, t) ← ⊤
12:            end if
13:        end for
14:    end for
15:    return G
16: end function
```

from the spurious edge criterion: it never removes an edge inferred by Z_0 that is not redundant. Furthermore, spurious edges represent redundant constraints, therefore, removing them does not cause precision loss.

Connected Components For an arbitrary Zone, we can safely over-approximate all affected inequalities from dv by identifying a connected component containing dv. Note, that changed variables are always in one component, since an update to a Zone creates an edge between them. Identifying dv's connected component reduces to discovering the undirected, reachable nodes of each $v \in dv$ in the spurious reduced Zone. The `CC` algorithm is a modified depth-first search algorithm, with $O(n^2)$ runtime complexity. In the beginning, the algorithm marks the special Z_0 node as visited, thus, preventing discovery of new paths through it, lest we undo the reduction of Algorithm 2.

The CC algorithm is the same for arbitrary, non-closed Zones and fully closed Zones. The resulting sets of changed variables, however, may differ. The set of inequalities of fully closed Zones are minimized but often more connected. However, more connections enables more spurious reductions, therefore, leading to smaller connected components.

Node Neighbors The Node Neighbors (NN) algorithm for an arbitrary Zone state is presented in Algorithm 3. In essence, the algorithm searches for the successor and predecessor of each changed variable, line 3. `ForwardReachable` returns the set of all reachable successor variables for each changed variable, $v \in dv$, using a typical depth-first search. `BackwardReachable` is similarly defined for reachable predecessor variables. In both cases, the zero variable receives special consideration. That is, we specially treat the zero variable as a sink with no outgoing edges during traversal.

The complexity of NN is $O(4n^2)$ because there are at most 2 changed variables from the DFA framework, and we do DFS twice per variable.

Algorithm 3 Algorithm for node neighbor selection for arbitrary Zones.

```
1: function NodeNeighbors(G, dv)
2:     variables ← {}
3:     for v ∈ dv do
4:         variables ← variables ∪ ForwardReachable(G, v)
5:         variables ← variables ∪ BackwardReachable(G, v)
6:     end for
7:     return variables
8: end function
```

Algorithm 4 Minimize Changed Variables Algorithm

```
1: function MinNeighbors(G, de)
2:     variables ← {}
3:     for (s, t) ∈ de do
4:         if s = Z0 then
5:             variables ← variables ∪ {t}
6:         else if t = Z0 then
7:             variables ← variables ∪ {s}
8:         else if s ≠ Z0 ∧ t ≠ Z0 then
9:             variables ← variables ∪ {s}
10:        end if
11:    end for
12:    return NodeNeighbors(G, variables)
13: end function
```

When given a fully closed Zone, the NN complexity is reduced. The form of a fully closed Zones makes explicit all transitively related variables. That is, the set of successors and predecessors of a variable in a fully closed Zone is equivalent to the local neighborhood of the variable, $R_G^-(v) \cup R_G^+(v) = N_G^{\pm}(v)$. Therefore, NN for the fully closed Zones simply returns the inequalities incident to the neighbor set of the requested variable.

The complexity and accuracy of finding a minimized state can be reduced for Zones in fully closed canonical form, where all dependencies are explicit. Thus, to identify affected inequalities by dv, we need to find incoming and outgoing edges of dv since they are potentially affected by those updates. The NN algorithm takes dv and a reduced state graph G, and retrieves all incoming and outgoing edges of dv, then uses them to identify its neighbors. The local subgraphs for each $v \in dv$ represent the identified changed inequalities. The runtime complexity of NN is linear $O(n)$, since it only considers each associated edge of dv.

The correctness of NN relies on the dependency information encoded in the successors and predecessors set. If a variable u is not in the $R_G^-(v) \cup R_G^+(v)$ set for v, then v does not depend on or relate to u. Furthermore, since u is not in this dependency set, it has not changed from the previous state. Therefore, the variable u can be removed from the dependent inequality set returned by NN.

Minimal Neighbors The previous CC and NN minimization algorithms assume that all updated variables, dv, modify inequalities within a Zone state, however, that may not always be the case. An updated variable might not induce changes to the state. The Minimal Neighbors (MN) technique improves upon this over-approximation by considering the set of updated edges de in a Zone state. DFA framework can provide this information when processing a statement, *e.g.,* an assignment or a conditional statement. Notice, that sources and targets of an edge in de are always in dv, but additional computation is required to identify de.

Algorithm 4 shows the pseudocode for identifying the changed variables among updated edges. Specifically, the algorithm takes as input G produced by `RemoveSpurious` and a set of updated edges, de. It starts by iterating over the set of updated edges, line 3. For each edge, the algorithm checks whether the source or target is the zero node, Z_0. If so, then the other node from the edge pair is added to the `variables` set, lines 5 and 7. This case handles when updates are on intervals because we must always include the non Z_0 variable in the filtered set of changed variables.

If neither source nor target is Z_0, then the source of the edge is added to the `variables` set, line 9, since this corresponds to the target variable restricting the source node, while the former remains unchanged for that edge. Finally, on line 12, the algorithm invokes NN procedure on G and the computed changed set of variables, and returns the minimized graph. The runtime complexity of MN is equivalent to NN: $O(4n^2)$. Similarly, if G is fully closed, the complexity is $O(n)$.

Below we provide a proof sketch that shows that for a given G from `RemoveSpurious` algorithm and an updated edge (s, t), the variable of its target, t does not change if the edges (t, s) and (Z_0, s) do not exist. That is, the target variable is never added to the variable set in Algorithm 4.

Proof. Given G with an updated edge (s, t) corresponding to the additional constraint $s - t \leq c$, for some $c \in \mathbb{Z}$. For the purpose of contradiction, let us assume t has changed as a result of the update from $s - t \leq c$. This means either there exists a path from t to s or that there exists a path (Z_0, s). But existence of such paths violates our assumption that $t, Z_0 \notin R_G^-(s)$. Therefore, t remains unchanged by the addition of the edge (s, t).

3.3 Widening and Merges

A few situations require special treatment for state updates in DFA. First, widening and merge points in the CFG of the analysis may induce more changed variables. Second, conditional transfers tend to modify more than a single variable, *e.g.,* two variables in three address form. Therefore, to ensure accurate and minimal comparisons, our techniques and comparisons must handle these situations. However, this is easily accomplished by each of our techniques since the parameter, dv for Algorithm 1 is a set of changed variables.

4 Experiments Methodology

To determine if the proposed minimization algorithms are efficient and effective, we evaluate them using subject programs within an existing DFA analyzer. For each subject program, we compute invariants at each statement for each abstract domain. Over the corpus of methods, we compute 4529 total invariants. The invariants are stored as logical formulas in SMT format. We run analysis on three domains: Zones, Intervals and Predicates, and compare the first with the last two using queries to an SMT solver. Since previous research demonstrates advantages of using fully closed Zone states [4], our experiments evaluate minimization algorithms for that canonical representation.

To evaluate the efficacy of CC, NN and MN, each after computing spurious constraint reduction, we compute the reduction in the number of variables and inequalities in the SMT formula of Zone invariants over the preceding techniques in $\preceq$, *i.e.,* CC vs. full state (FS), NN vs. CC, and MN vs. NN. That is, we compute the percentage of change per program statement and then average them over all methods. We use percentage, and not absolute values since the number of variables changes from statement to statement. Similarly, we compute the average percentage change for inequalities. Since program branches compute possibly different sized sets of variables and inequalities, we take the maximum number of variables and inequalities between the two.

Using the invariants computed for Zones, Intervals, and Predicates, we entail the invariants to compare the precision of Zones vs. Intervals and Zones vs. Predicates for each minimization algorithm. The results for Interval's invariants are classified as less precise $\prec$ or equal $=$. Predicates extend these categories to include more precise $\succ$ or incomparable $\prec\succ$ to Zones.

Subject Programs Subject programs consist of 127 Java methods used in previous research [3,31]. Methods from the DFA benchmark suite [31] were extracted from a wide range of open-source projects and have a high number of integer operations. The subject programs range from 4 to 412 Jimple instructions, a three address intermediate representation.

The EQBench suite [3] consists of method pairs for testing differential symbolic execution tools [2,30]. We sampled only original methods and excluded renamed equivalent methods.

Experimental Setup We execute each of the analyses on a single GNU/Linux machine, running Linux kernel version `5.15.89`, equipped with an AMD Ryzen Threadripper 1950X 16-Core Processor and 64 GB of system memory. We use an existing DFA static analysis tool [31] implemented in the Java programming language. The analysis framework uses Soot [29] version `4.2.1`. Similarly, we use Z3 [27], version `4.8.17` with Java bindings to compare SMT expressions from the abstract domain states. Finally, we use Java version 11 to execute the analyses, providing the following JVM options: `-Xms4g`, `-Xmx32G`, `-XX:+UseG1GC`, `-XX:+UseStringDeduplication`, and `-XX:+UseNUMA`.

Implementation We use the reduction from Larsen *et al.* [19] to create an equivalent, but reduced invariant expression at each program point. We combine the output states via logical entailment to compare Zones to Intervals and

to Predicates. The set of variables in a minimized Zone state determines what variables are extracted from the corresponding full states of Intervals and Predicates. After entailment, we use Z3, using the linear integer arithmetic (LIA) logic, to decide model behavior of each domain. Using the GNU `time` [13] command, version 1.9, we capture the walk-clock execution time of Z3.

Evaluations Intervals and Zones perform widening operations after two iterations over widening program points. We do not preform narrowing for either domain because narrowing is program specific. The lack of narrowing does not affect our results since we are evaluating techniques for identifying minimal subsets of changes, not techniques for improving precision.

We use a generic disjoint Predicate domain, which does not affect generality of the results. The Predicate domain's elements are influenced by Collberg *et al.'s* [7] study on Java programs and numerical constants. Consequently, the domain elements use several of the most common integer constants found in Java programs. The specific Predicate domain used in this study consists of the following set of disjoint elements: $\{(-\infty, -5]$, $(-5, -2]$, -1, 0, 1, $[2, 5)$, $[5, +\infty)\}$.

5 Evaluation Results and Discussions

To empirically evaluate efficiency and effectiveness of the state minimization algorithms, we answer the following research questions:

RQ1 How well do the minimization algorithms reduce the size of a Zone state and improve runtime of domain comparisons?

RQ2 How do the minimization algorithms affect categorization of domain comparison results?

5.1 Impact of Minimization on State Size and Comparison Efficiency

Table 1 contains data for efficiency evaluation, split over the two benchmark suites. The table shows the average percentage reduction in vertices and edges in Zones, comparing to the preceding minimization algorithm (columns 2–4); and as a reduction in total runtime for all comparisons between Zones and Interval states (column 5) and between Zones and Predicates (column 6).

We aggregate the relative change in vertices and edges over all subject methods for a more tractable comparison. We use the percentage change to answer the first part of **RQ1** related to state sizes. The data show a large reduction in the number of vertices between FS and after applying CC algorithm. The number of edges features a similarly significant, though less dramatic reduction since they are compared without spurious constraints. On average, we see small reductions in the rest of the comparisons. The difference in vertex reductions versus edge reductions is due to the reduced number of vertices required, contrasted with edge sparsity arising from widening and merge operations which affects all representations. However, as the small reduction of edges between MN and NN

DFA Subject Programs					
State Type	**vs.**	**↓ Δ % V**	**↓ Δ % E**	**∼ Inter, sec.**	**∼ Pred, sec.**
FS	-	-	-	4.03	265.91
CC	FS	70.37	29.47	1.41	4.09
NN	CC	0.02	0.01	1.41	4.04
MN	NN	0.10	0.05	1.35	4.05
EQBench Subject Programs					
FS	-	-	-	0.79	5.56
CC	FS	43.0	2.1	0.63	0.87
NN	CC	0.0	0.0	0.58	0.9
MN	NN	0.13	0.13	0.58	0.9

Table 1: Average percentage changes in V and E between each technique (columns 2–4), and average total runtime of state comparisons (columns 5,6).

shows that after removing more vertices from the subgraph, we remove more edges as well.

The EQBench results mirror the reduction of edges and variables. In the EQBench benchmark suite, we see no reduction between CC to NN. However, our final approach does remove vertices and edges from the previous techniques of CC and NN. This reduction is attributable to the bisection enabled by our final technique which further reduces sparse graphs based on the semantics of the changed constraint.

Addressing the second part of **RQ1**, we compare the average total runtime of comparisons over the corpus of subject programs. Columns 5 and 6 of Table 1 show the total runtime to execute all domain comparisons, averaged over 5 executions, and broken down by benchmark suite. Between FS and CC, we see dramatic reductions in total time. As expected the remaining techniques show small improvements in comparison time due to the minor reductions of vertices and edges shown in columns 2–4. The increase in time for Zone and Predicate comparison for the EQBench subject programs is attributable to execution variance. Overall, our minimization algorithms reduce the size of Zone states and, in turn, improve the efficiency of domain comparisons.

5.2 Impact on Domain Comparison

Comparable domains We compare Zones and Intervals invariants to answer **RQ2**. We break down the results by benchmark suites in Table 2. Columns 2 and 3 show the precision summary of invariants between Zones and Intervals.

In the DFA suite, using FS to compare each domain, Zones compute more precise invariants for approximately 3/4 of the total number of invariants. However, the ratio drops significantly to less than a third, (31%), when using the CC technique. Our final technique MN lowers the percentage of invariants where Zones are more precise to about 30% of all computed invariants. Furthermore, our techniques demonstrate the preponderance of invariants where Zones and

DFA Subject Programs						
State	**$\succ$ Intervals**	**= Intervals**	**$\succ$ Pred**	**= Pred**	**$\prec$ Pred**	**$\prec\succ$ Pred**
FS	2898	1002	1464	237	167	2032
CC	1194	2706	1324	1930	473	173
NN	1191	2709	1322	1933	473	172
MN	1164	2736	1305	1960	473	162
EQBench Subject Programs						
FS	374	255	307	135	46	141
CC	131	498	217	322	72	18
NN	131	498	217	322	72	18
MN	131	498	217	322	72	18

Table 2: Summary of comparison between Zones and Intervals(2, 3), and between Zones and Predicates (4–7).

Intervals are equivalent. We see similar results when considering the methods of the EQBench suite. Using the full state to compare Zones and Intervals, we see Zones compute a majority of more precise invariants, about 59%. However, using any one of our minimization techniques moves the proportion of more precise invariants to 21%. We attribute the lack of further reduction with later techniques to the preponderance of non-integer operations in the EQBench suite.

Incomparable Domains Additionally, with respect to **RQ2**, we compare Zones to Predicates to evaluate whether our techniques minimize the number of incomparable invariants computed between the two domains. Since Zones and Predicates are incomparable, we consider all comparison categories denoted here as: $\succ$ *Pred* for Zones more precise than Predicates; $=$ *Pred* for Zones equivalent to Predicates; $\prec$ *Pred* for Zones less precise than Predicates; and $\prec\succ$ *Pred* for Zones and Predicates being incomparable.

Columns 4–6 of Table 2 summarizes the distribution of relative precision for Zones and Predicates over the computed invariants of the subject programs. Unlike Zones, Predicates cannot represent arbitrary integer constants; Predicates are limited to the *a priori* chosen predicate elements. However, Predicates can represent disjoint ranges of values which Zones, and other numerical domains, cannot. As such, when using FS, we see a high percentage of invariants fall into either $\succ$ *Pred* and $\prec\succ$ *Pred*. The $\succ$ *Pred* makes up about 38% of invariants. Similarly, $\prec\succ$ *Pred* weighs in at $\sim$ 52% of invariants.

When, applying CC, the percentage of incomparable invariants drops significantly to 4%. $\prec$ *Pred* comprise 12% of invariants, up from 4%. Similarly, the percentage of $\succ$ *Pred* drops from 38% to 34%. Finally, equality between the two domains significantly increased from 6% to about 49%. These trends continue for each technique. Each technique shifts the distributions of invariants from $\succ$ *Pred* and $\prec\succ$ *Pred* to $=$ *Pred* and $\prec$ *Pred*. In MN state, Zones are more precise for about 33% of invariants, down from 38%; Zones are equal to Predicates for about 50% of invariants, up from 6%; and Zones and Predicates are incomparable for about 4% of invariants, down from 52%. For each technique, Zones less precise

than Predicates remained at 12%, up from 4% compared to FS, between the techniques. Considering the EQBench methods, we observe similar results for Zones and Predicates. Using the CC technique, we see a significant shift in the distribution of invariants. However, we do not see any further distribution shifts in this program set. We attribute this to the fact that the EQBench methods consist of many non-integer operations.

Clearly, our techniques reduce the percentage of incomparable invariants, enabling, for example, more accurate comparison between Zones and incomparable domains, such as Predicates. While not the goal of this study, the comparable results confer merit to previous research which anecdotally mentions: the majority of computed invariants are interval valued [10,16]. This improved accuracy would be especially valuable in adaptive analysis approaches where a heuristic decides which abstract domain to utilize for a specific block of code.

5.3 Threats to Validity

External Threats The subject programs from previous research [3,31] were extracted from real, open-source projects, each with a high number of integer operations. The EQBench suite consists of predominately numerical programs but demonstrate the generalizability of our results. Other concerns include the choice of Predicate elements and lack of narrowing which could influence the precision counts between Zones and Predicates and between Zones and Intervals, respectively. However, since we examine only the trend of the different categories, the exact precision does not affect our conclusions.

Internal Threats To mitigate internal threats to validity, we developed a large test suite, 703 unit tests, to ensure our implementation is correct. The test suite contains numerous tests which check the consistency of the partial order over Zones and Intervals. Furthermore, we developed and manually verified tests to check comparison between Zones and Predicates. Specifically, the test suite contains manually verified tests which use real subject programs to test the correctness of the analyses and their comparisons.

6 Related Work

We have mentioned our spurious reduction technique is based on the work by Larsen *et al.* [19]. Their algorithm removes all redundant constraints without removing variables, reducing the overall number of linear integer inequalities but it does not reduce the number of variables. Along similar lines, Giacobazzi *et al.* [12] proposed techniques for abstract domain compression for complete finite abstract domains. That is, it reduces the number of constraints within the logical formula without altering the approximation of the abstract domain. Our techniques extract subsets of the state for specific verification tasks.

Our approach, Connected Components (CC), resembles the slicing technique of the Green solver interface [33] and split normal form introduced by Gange *et al.* [10,11]. Our approach differs from Green in application and restriction to

Zone constraints. Slicing can extract connected constraints by what variables are present in the set of constraints. However, we can exclude transitive relations between variables because within Zones, not all variables are modified by the introduction of a new constraint.

We base our methodology on previous work on new abstract domains which provide a comparison of the new domain against other known similar or comparable domains. These comparisons can be categorized into two predominate strategies. The first, domains are compared via a known set of properties over benchmark programs [10,15,16,20,23]. The second, domains are compared via logical entailment of the invariants computed at program points [25,31]. In the first case, the comparison is straightforward. In the latter case, as this work demonstrates, the precision between two domains can depend on the set of invariants used to perform the comparison.

7 Conclusion and Future Work

We proposed several techniques which identify a minimal set of changed inequalities for the Zones abstract domain. Our techniques improve upon existing techniques such as Green's slicing [33] technique which further reduces the number of dependent variables within a changed set of inequalities. We empirically evaluated our techniques and showed improvements of efficacy and efficiency. Concretely, the changed subgraph of Zones is equivalent to Intervals in more than 70% of computed invariants, a result commented on but never demonstrated in previous research [10,16]. Moreover, our techniques significantly reduced the incomparable invariants found when comparing two incomparable domains, resulting in a clearer picture of the relative precision between the two domains. Furthermore, the reduction in variables improved the efficiency of domain comparison, reducing average total runtime of incomparable domain comparisons by 98%. While evaluated within the context of DFA frameworks, we presented general algorithms which, we believe, are applicable in other areas of formal methods such as model checking and symbolic execution.

Future Work We intend to extend this work to include additional relational domains. Specifically, enabling comparison between two relational domains, such as Zones and Octagons, is an interesting avenue to pursue. Since the techniques improve accuracy in comparison between domains they could be beneficial for adaptive static analysis techniques which selectively use the best abstraction. We believe this work also opens up possibilities of comprehensive studies which empirically validate several abstract domains and their partial ordering. Specifically, it would be interesting to see comprehensive comparisons between Predicate domains and Zones.

Acknowledgments

The work reported here was supported by the U.S. National Science Foundation under award CCF-19-42044.

References

1. Abate, C., Blanco, R., Ciobâcă, c., Durier, A., Garg, D., Hriţcu, C., Patrignani, M., Tanter, E., Thibault, J.: An extended account of trace-relating compiler correctness and secure compilation. ACM Transactions on Programming Languages and Systems **43**(4), 1–48 (Dec 2021). https://doi.org/10.1145/3460860
2. Badihi, S., Akinotcho, F., Li, Y., Rubin, J.: Ardiff: scaling program equivalence checking via iterative abstraction and refinement of common code. Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (11 2020). https://doi.org/10.1145/3368089.3409757, `http://dx.doi.org/10.1145/3368089.3409757`
3. Badihi, S., Li, Y., Rubin, J.: Eqbench: A dataset of equivalent and non-equivalent program pairs. 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR) (May 2021). https://doi.org/10.1109/msr52588.2021.00084, `http://dx.doi.org/10.1109/MSR52588.2021.00084`
4. Ballou, K., Sherman, E.: Incremental transitive closure for zonal abstract domain. NASA Formal Methods p. 800–808 (2022). https://doi.org/10.1007/978-3-031-06773-0_43
5. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation - PLDI '03 (2003). https://doi.org/10.1145/781131.781153
6. Chernikova, N.: Algorithm for finding a general formula for the non-negative solutions of a system of linear inequalities. USSR Computational Mathematics and Mathematical Physics **5**(2), 228–233 (1965). https://doi.org/10.1016/0041-5553(65)90045-5, `https://www.sciencedirect.com/science/article/pii/0041555365900455`
7. Collberg, C., Myles, G., Stepp, M.: An empirical study of java bytecode programs. Software: Practice and Experience **37**(6), 581–641 (2007). https://doi.org/10.1002/spe.776
8. Cousot, P., Cousot, R.: Static determination of dynamic properties of programs. In: Proceedings of the Second International Symposium on Programming. pp. 106–130. Dunod, Paris, France (1976)
9. Dill, D.L.: Timing assumptions and verification of finite-state concurrent systems. Lecture Notes in Computer Science p. 197–212 (1990). https://doi.org/10.1007/3-540-52148-8_17
10. Gange, G., Ma, Z., Navas, J.A., Schachte, P., Søndergaard, H., Stuckey, P.J.: A fresh look at zones and octagons. ACM Transactions on Programming Languages and Systems **43**(3), 1–51 (Sep 2021). https://doi.org/10.1145/3457885
11. Gange, G., Navas, J.A., Schachte, P., Søndergaard, H., Stuckey, P.J.: Exploiting sparsity in difference-bound matrices. Lecture Notes in Computer Science p. 189–211 (2016). https://doi.org/10.1007/978-3-662-53413-7_10
12. Giacobazzi, R., Mastroeni, I.: Domain compression for complete abstractions. Verification, Model Checking, and Abstract Interpretation p. 146–160 (Dec 2002). https://doi.org/10.1007/3-540-36384-x_14
13. Gordon, A.: Gnu time, `https://www.gnu.org/software/time/`
14. Graf, S., Saidi, H.: Construction of abstract state graphs with pvs. Lecture Notes in Computer Science p. 72–83 (1997). https://doi.org/10.1007/3-540-63166-6_10

15. Gurfinkel, A., Chaki, S.: Boxes: A symbolic abstract domain of boxes. Lecture Notes in Computer Science p. 287–303 (2010). https://doi.org/10.1007/978-3-642-15769-1_18
16. Howe, J.M., King, A.: Logahedra: A new weakly relational domain. Lecture Notes in Computer Science p. 306–320 (2009). https://doi.org/10.1007/978-3-642-04761-9_23
17. Katz, S.: Program optimization using invariants. IEEE Transactions on Software Engineering **SE-4**(5), 378–389 (Sep 1978). https://doi.org/10.1109/tse.1978.233858
18. Kildall, G.A.: A unified approach to global program optimization. Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages - POPL '73 (1973). https://doi.org/10.1145/512927.512945
19. Larsen, K., Larsson, F., Pettersson, P., Yi, W.: Efficient verification of real-time systems: Compact data structure and state-space reduction. In: Proceedings Real-Time Systems Symposium. pp. 14–24. IEEE Comput. Soc (1997). https://doi.org/10.1109/real.1997.641265
20. Laviron, V., Logozzo, F.: Subpolyhedra: A (more) scalable approach to infer linear inequalities. Verification, Model Checking, and Abstract Interpretation p. 229–244 (2008). https://doi.org/10.1007/978-3-540-93900-9_20
21. Le Verge, H.: A Note on Chernikova's algorithm. Research Report RR-1662, INRIA (1992), `https://hal.inria.fr/inria-00074895`
22. Lloyd, J., Sherman, E.: Minimizing the size of path conditions using convex polyhedra abstract domain. ACM SIGSOFT Software Engineering Notes **40**(1), 1–5 (Feb 2015). https://doi.org/10.1145/2693208.2693244, `http://dx.doi.org/10.1145/2693208.2693244`
23. Logozzo, F., Fähndrich, M.: Pentagons: A weakly relational abstract domain for the efficient validation of array accesses. Science of Computer Programming **75**(9), 796–807 (9 2010). https://doi.org/10.1016/j.scico.2009.04.004
24. Miné, A.: A new numerical abstract domain based on difference-bound matrices. Lecture Notes in Computer Science p. 155–172 (2001). https://doi.org/10.1007/3-540-44978-7_10
25. Miné, A.: Weakly Relational Numerical Abstract Domains (12 2004), `https://pastel.archives-ouvertes.fr/tel-00136630`
26. Miné, A.: The octagon abstract domain. Higher-Order and Symbolic Computation **19**(1), 31–100 (3 2006). https://doi.org/10.1007/s10990-006-8609-1
27. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. Lecture Notes in Computer Science p. 337–340 (2008). https://doi.org/10.1007/978-3-540-78800-3_24
28. Nötzli, A., Reynolds, A., Barbosa, H., Niemetz, A., Preiner, M., Barrett, C., Tinelli, C.: Syntax-guided rewrite rule enumeration for smt solvers. In: Janota, M., Lynce, I. (eds.) Theory and Applications of Satisfiability Testing – SAT 2019. pp. 279–297. Springer International Publishing, Cham (2019)
29. OSS, S.: Soot (2020), `https://soot-oss.github.io/soot/`
30. Person, S., Dwyer, M.B., Elbaum, S., Păsăreanu, C.S.: Differential symbolic execution. Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering - SIGSOFT '08/FSE-16 (2008). https://doi.org/10.1145/1453101.1453131, `http://dx.doi.org/10.1145/1453101.1453131`
31. Sherman, E., Dwyer, M.B.: Exploiting domain and program structure to synthesize efficient and precise data flow analyses (t). 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE) (11 2015). https://doi.org/10.1109/ase.2015.41

32. Tange, O.: Gnu parallel 20221222 ('chatgpt') (Dec 2022). https://doi.org/10.5281/zenodo.7465517, https://doi.org/10.5281/zenodo.7465517, GNU Parallel is a general parallelizer to run multiple serial command line programs in parallel without changing them.
33. Visser, W., Geldenhuys, J., Dwyer, M.B.: Green: Reducing, reusing and recycling constraints in program analysis. Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (Nov 2012). https://doi.org/10.1145/2393596.2393665, http://dx.doi.org/10.1145/2393596.2393665
34. Yu, H., Monniaux, D.: An efficient parametric linear programming solver and application to polyhedral projection. In: Chang, B.Y.E. (ed.) Static Analysis. pp. 203–224. Springer International Publishing, Cham (2019)
35. Zhu, H., Magill, S., Jagannathan, S.: A data-driven chc solver. Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (Jun 2018). https://doi.org/10.1145/3192366.3192416