# Repositioning of Static Analysis Alarms

Tukaram Muske
Tata Consultancy Services
India
t.muske@tcs.com

Rohith Talluri
Tata Consultancy Services
India
rohith.talluri@tcs.com

Alexander Serebrenik
Eindhoven University of Technology
The Netherlands
a.serebrenik@tue.nl

## ABSTRACT

The large number of alarms reported by static analysis tools is often recognized as one of the major obstacles to industrial adoption of such tools.

We present repositioning of alarms, a novel automatic postprocessing technique intended to reduce the number of reported alarms without affecting the errors uncovered by them. The reduction in the number of alarms is achieved by moving groups of related alarms along the control flow to a program point where they can be replaced by a single alarm. In the repositioning technique, as the locations of repositioned alarms are different than locations of the errors uncovered by them, we also maintain traceability links between a repositioned alarm and its corresponding original alarm(s). The presented technique is tool-agnostic and orthogonal to many other techniques available for postprocessing alarms.

To evaluate the technique, we applied it as a postprocessing step to alarms generated for 4 verification properties on 16 open source and 4 industry applications. The results indicate that the alarms repositioning technique reduces the alarms count by up to 20% over the state-of-the-art alarms grouping techniques with a median reduction of 7.25%.

## CCS CONCEPTS

• **Theory of computation → Program analysis**; • **Software and its engineering → Formal software verification**;

## KEYWORDS

Static analysis, static analysis alarms, data flow analysis, anticipable conditions, available conditions, alarms repositioning

**ACM Reference Format:**
Tukaram Muske, Rohith Talluri, and Alexander Serebrenik. 2018. Repositioning of Static Analysis Alarms. In *Proceedings of 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'18)*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3213846.3213850

## 1 INTRODUCTION

Static analysis tools have shown promise in automated detection of code anomalies and programming errors [3, 4, 6, 30, 32]. In practice, due to approximations used during analysis [5, 16, 21, 28] the tools often generate a large number of alarms, i.e., warning messages notifying the tool-user about potential errors. A high percentage of these alarms are false positives, i.e., alarms that do not represent an error. To partition the alarms into false positives and true errors, postprocessing of the alarms, often manual, is inevitable [12, 14]. Therefore, the large number of alarms generated and cost involved in partitioning them manually have been recognized as major concerns in adoption of static analysis tools [5, 7, 16, 19].

One of the approaches for effective handling of the alarms [11, 26] consists in grouping of related alarms and representing each group as a single alarm. However, state-of-the-art grouping approaches [13, 20, 24, 31] sometimes fail to group alarms related by the same causes or variables, e.g., when the related alarms belong to different branches of an *if* statement. This limitation is further illustrated in Section 2. To overcome this limitation, we present repositioning of alarms, a postprocessing technique that moves the alarms up or down the program control flow without affecting the errors uncovered. The primary goal of the repositioning is:

| To reduce the number of alarms reported without affecting the errors uncovered. |
|---|

Furthermore, since traditional static analysis tools report alarms at the locations where run-time errors are likely to occur, the user has to traverse the code back to the causes of an alarm to identify whether the alarm represents an error or not [12, 18, 23]. Given the large size and complexity of industrial source code [23], this traversal can be a daunting task. To reduce these code traversals we therefore, through repositioning, also aim:

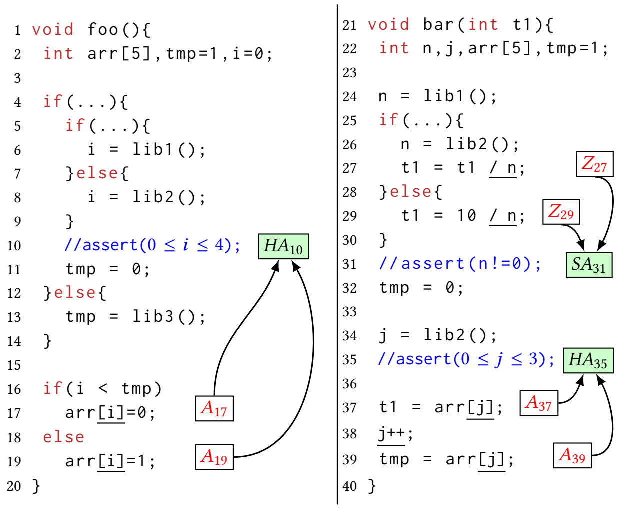
| To report alarms closer to their causes. |
|---|

We implement alarms repositioning by propagating the alarm conditions—checks performed by the analysis tools—first in the backward direction and later in the forward direction. The propagation of conditions is through computation of *anticipable* and *available* conditions respectively using data flow analyses [17, 27, 29].

Our technique is tool-agnostic and orthogonal to many other techniques available for postprocessing the alarms. The technique is suitable for alarms reported by analysis tools that compute flow of values of the program variables during the analysis, e.g., Polyspace Code Prover [1] and Frama-C [9]. We do not consider alarms reported based on structural information or local bug patterns [15].

We performed empirical evaluation of the proposed technique using 33,162 alarms generated by a commercial static analysis tool on 16 open source and 4 industry applications. The open source applications were selected from the benchmarks used to evaluate earlier alarms grouping techniques [20, 31]. The industry applications were embedded systems belonging to the automotive domain. Before performing repositioning, the input alarms were processed using state-of-the-art grouping techniques [20, 24, 31].

*ISSTA'18, July 16–21, 2018, Amsterdam, Netherlands*
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-5699-2/18/07…$15.00
https://doi.org/10.1145/3213846.3213850



```
1  void foo(){
2   int arr[5],tmp=1,i=0;
3
4   if(...){
5     if(...){
6       i = lib1();
7     }else{
8       i = lib2();
9     }
10    //assert(0 ≤ i ≤ 4);
11    tmp = 0;
12  }else{
13    tmp = lib3();
14  }
15
16  if(i < tmp)
17    arr[i]=0;
18  else
19    arr[i]=1;
20 }
```

```
21 void bar(int t1){
22  int n,j,arr[5],tmp=1;
23
24  n = lib1();
25  if(...){
26    n = lib2();
27    t1 = t1 / n;
28  }else{
29    t1 = 10 / n;
30  }
31  //assert(n!=0);
32  tmp = 0;
33
34  j = lib2();
35  //assert(0 ≤ j ≤ 3);
36
37  t1 = arr[j];
38  j++;
39  tmp = arr[j];
40 }
```

(1) *lib1*, *lib2*, and *lib3* are library functions whose code is not available for static analysis and their return-type is *signed int*.
(2) Ellipsis (...) indicates the code omitted for simplifying the example.

**Figure 1: Alarm examples with their repositioning**

We observed that the proposed repositioning of alarms reduces alarms count by up to 20% over the grouping techniques, with a median reduction of 7.25% and the average reduction being 6.47%. Evaluating potential benefits of repositioning in reducing the code traversals during manual partitioning of alarms, due to reporting the alarms closer to their cause points, is out of scope of this paper.

*The key contribution of the paper is a novel and empirically evaluated postprocessing technique that reduces alarms by repositioning.*

*Paper outline.* Section 2 presents an informal overview of the repositioning technique using a motivating example. Section 3 describes repositioning of alarms formally, while Sections 4 and 5 describe two data flow analyses used to reposition the alarms. Section 6 discusses our experimental evaluation. Section 7 presents related work, and Section 8 concludes.

## 2 INFORMAL DISCUSSION

Consider the *C* code example in Figure 1 adapted from a real-life embedded system. The code is simplified considerably but it is still sufficiently rich to present the alarms repositioning technique. It includes two functions, *foo* and *bar*, independent of each other. Analyzing the code using a static analysis tool such as Polyspace Code Prover [1] or Frama-C [9] generates six alarms of two commonly checked categories of runtime errors: array index out of bounds (AIOB) and division by zero (ZD). The alarms are generated because values returned by the calls to library functions are treated as unknown by the analysis tool. The alarms are reported at the locations where run-time errors are likely to occur. We use notations $A_n$ and $Z_n$, respectively, to denote an alarm at line *n* corresponding to AIOB and ZD. We also refer to these tool generated alarms as the *original alarms* and to their locations as the *original locations*.

Several techniques aiming at reduction of the number of alarms group them based on similarity or correlation [26]. Those techniques [20, 24, 31] achieve the reduction by (1) identifying a *dominant* alarm $\phi_1$ for an alarm $\phi_2$, i.e., $\phi_1$ such that $\phi_2$ is always *false* whenever $\phi_1$ is *false*; and (2) grouping $\phi_1$ and $\phi_2$ together. These grouping techniques are, however, unable to group the alarms $A_{17}$ and $A_{19}$ in Figure 1, because the alarms are reported in the two different branches of the *if* statement at line 16 and hence neither of them can be identified as a dominant alarm for the other. In this case, both $A_{17}$ and $A_{19}$ are reported as dominant alarms. Similarly, $Z_{27}$ and $Z_{29}$ do not get grouped together because they are reported in the two different branches of the *if* statement at line 25. Furthermore, these two alarms are caused by different reasons: the assignments at lines 26 and 24 respectively.

Gehrke et al. [13] use propagation of the alarm conditions to reduce the number of alarms. First, alarm conditions from the original alarms, propagated backward, are used to insert alarms at new locations. Later, alarm conditions of the inserted alarms, propagated forward, are used to remove the original alarms. However, as alarms are inserted when no more upward propagation of the conditions is possible, the number of alarms reported finally can increase. For example, for $A_{17}$ and $A_{19}$ the approach inserts three new alarms: two at locations immediately after lines 6 and 8, and one immediately before line 13. Also, applying this approach to $Z_{27}$ and $Z_{29}$ does not help in reducing the alarms count.

Our repositioning approach, described next using examples, is motivated by the work of Gehrke et al. [13], and overcomes its limitations and the limitations of grouping techniques [20, 24, 31].

**Example 1 (Hoisting of alarms):** Consider the alarms $A_{17}$ and $A_{19}$: both are AIOB alarms and based on the same variable *i*.

Given an alarm $\phi$, *alarm condition cond*($\phi$) is such that when it evaluates to *true* (resp. *false*), $\phi$ is a safe (resp. erroneous) program point. For example, *cond*($A_{17}$) and *cond*($A_{19}$) are $i \geq 0$ && $i \leq 4$.

As $A_{17}$ and $A_{19}$ have the same alarm condition and the same reasons (causes) for their generation at lines 6 and 8, they can be merged together and the new alarm after merging can be repositioned at line 10 where the paths coming from these causes meet for the first time. The alarm after the repositioning is shown as an assertion $HA_{10}$, where *cond*($HA_{10}$) is $i \geq 0$ && $i \leq 4$. During the repositioning, the effect of the *else* branch at line 12 is ignored, because (1) $i = 0$ if the *else* branch at line 12 is taken and the alarms $A_{17}$ and $A_{19}$ are safe due to this value; and (2) we are not interested in scenarios in which the alarms are guaranteed to be safe.

We refer to the repositioning of an alarm to a point earlier in the code as *hoisting*, and the alarm after the repositioning is referred to as the *hoisted alarm*. The hoisting of $A_{17}$ and $A_{19}$ at line 10 is safe, because *cond*($HA_{10}$) $\Leftrightarrow$ (*cond*($A_{17}$) $\wedge$ *cond*($A_{19}$)). Thus, reporting $HA_{10}$ instead of $A_{17}$ and $A_{19}$ is sufficient for error detection.

Note that the hoisting of $A_{17}$ and $A_{19}$ is also possible and safe at line 15, however we prefer the hoisting at line 10, because the alarm reported at line 10 is closer to its causes at lines 6 and 8: recall that we try to reduce backward code traversals performed during manual inspections of the original alarms. For example, inspecting an alarm at line 15 (or $A_{17}$ or $A_{19}$) requires traversing the code from line 15 backwards either via the "then" branch to assignments in lines 6 and 8, or via the "else" branch at line 12 back to the assignment at line 2. Inspecting the hoisted alarm $HA_{10}$ eliminates the need of inspecting the "else" branch. The gain achieved due to eliminating such code traversals, can be even bigger when the original and hoisted alarms belong to different functions.

Note that the possible hoisting of $A_{17}$ and $A_{19}$ closest to their causes, is immediately after the assignments at lines 6 and 8. However, doing so results in two new alarms and it does not allow us to reduce the number of alarms. Thus, we prefer hoisting of $A_{17}$ and $A_{19}$ at line 10 and this hoisting is optimal considering the two alarms repositioning goals. Such alarms repositioning not only reduces the number of alarms by one but also reduces code traversals, to some extent, performed during the manual inspections.

We stress that, in absence of either $A_{17}$ or $A_{19}$, the other alarm cannot be safely hoisted to a location earlier in the code (e.g. to line 15), as the hoisting is not *outcome preserving*: the hoisted alarm can represent an error while the original alarm being a safe point.

**Example 2 (Sinking of alarms):** The hoisting of alarms achieves both the repositioning goals together. However, it may not always help to merge alarms. For example, the original alarms $Z_{27}$ and $Z_{29}$ are candidates for repositioning as they have the same alarm condition and also they appear in the different branches of the *if* statement at line 25. They cannot be merged and hoisted before the *if* statement at line 25 as doing so misses capturing the effect of the cause point at line 26. In such cases, repositioning them later in the code at line 31 allows to merge them together while capturing the effect of both the cause points at lines 24 and 26. This repositioning helps to reduce the alarms count by one. The alarm after the repositioning is shown as $SA_{31}$, and $cond(SA_{31})$ is $n \neq 0$. This repositioning is safe because $cond(SA_{31}) \Leftrightarrow (cond(Z_{27}) \wedge cond(Z_{29}))$.

We refer to this type of repositioning down the control flow as *sinking of alarms*, and the alarm after sinking is referred to as *sunk alarm*. Since there can exist multiple program points for safe sinking of alarms (as it is the case at line 31 onwards for alarms $Z_{27}$ and $Z_{29}$), we choose the program point for alarms sinking as the highest program point where paths coming from the alarms meet for the first time (i.e., at line 31 for the alarms $Z_{27}$ and $Z_{29}$). Note that although the sinking reduces the number of alarms by one (as per the primary goal), however the sunk alarm $SA_{31}$ is further away from the alarm causes. Thus, we perform sinking of alarms only if it reduces more alarms than their hoisting. □

**Example 3:** Consider $A_{37}$ and $A_{39}$ alarms that are based on the same variable $j$. The grouping techniques fail to identify any one of them as a dominant alarm for the other due to the increment operation at line 38, resulting in reporting of both the alarms as dominant alarms. We observe that both the alarms can be safely merged into a single alarm repositioned at line 35, denoted by $HA_{35}$. Note that $cond(HA_{35})$, i.e. $j \geq 0$ && $j \leq 3$, is such that $cond(HA_{35}) \Leftrightarrow (cond(A_{37}) \wedge cond(A_{39}))$. In this scenario, the repositioning allowed us to reduce the alarms number by one. Furthermore, it also eliminates inspecting the second alarm which requires considering the effect of the increment operation at line 38. The saving achieved can be considerable if code at the three lines, 37-39, appear in different functions. □

*Traceability.* In the repositioning technique, as the locations of the repositioned alarms are different than the locations of the errors detected by them, we also maintain traceability links between a repositioned alarm and its corresponding original alarm(s). These links will be explored by users only when a repositioned alarm is found to uncover an error during manual inspection and a correction is needed at its corresponding original alarm program point(s).

## 3 TECHNIQUE OVERVIEW

This section describes our alarm repositioning technique. We first recapitulate the notions related to the control flow graph (CFG).

### 3.1 Background: Control Flow Graph

A control flow graph (CFG) [2] of a program is a directed graph $\langle \mathrm{N}, \mathrm{E} \rangle$, where N is a set of nodes representing the program statements (like assignments and controlling conditions); and E is a set of edges where an edge $(n_1, n_2)$ represents a possible flow of program control from $n_1 \in \mathrm{N}$ to $n_2 \in \mathrm{N}$ without any intervening node. A CFG has two distinguished nodes *Start* and *End*, representing the entry and exit of the corresponding program, respectively. Except for the *Start* and *End* nodes, we assume that there is one-to-one correspondence between the CFG nodes and their corresponding statements in the program. The program statements are assumed not to cause side effects. For a given node $n$, we use $pred(n)$ (resp. $succ(n)$) to denote predecessors (resp. successors) of $n$ in the graph.

We write $entry(n)$ and $exit(n)$ to denote the *entry* and *exit* of a node $n$, i.e., the program points *just before* and *immediately after* the execution of statement corresponding to the node $n$, respectively. The entry/exit of a node is assumed not to be shared with entry or exit of any other node even though they may relate to the same program point. Henceforth, we use $n_m$ to denote the node of a program statement at line $m$ when a code example is referred.

### 3.2 Definitions

Similarly to the available and anticipable expressions [17, 27], we define available and anticipable alarm conditions. We use notation $\phi_p$ to denote an original alarm $\phi$ reported at a program point $p$.

Definition 1 (Available Alarm Conditions). *An alarm condition c is available at a program point p, if every path from the program entry to p contains an alarm $\phi_q$ with c as its alarm condition, and the point q is not followed by a definition of any operand of c on any path from q to p.* □

In Figure 1, condition $n \neq 0$ is available alarm condition at all program points after $entry(n_{32})$ due to the alarms $Z_{27}$ and $Z_{29}$.

Definition 2 (Anticipable Alarm Conditions). *An alarm condition c is* anticipable *at a program point p, if every path from p to the program exit contains an alarm $\phi_q$ with c as its alarm condition, and the point q is not preceded by a definition of any operand of c on any path from p to q.* □

In Figure 1, condition $i \geq 0$ && $i \leq 4$ is anticipable at $entry(n_{13})$, $exit(n_6)$, and $exit(n_8)$ due to the alarms $A_{17}$ and $A_{19}$.

Definition 3 (Safe Repositioning of alarms). *A repositioning (hoisting or sinking) of a set of alarms S to a program point p is said to be safe if $c_p \Leftrightarrow \wedge_{\phi \in S} cond(\phi)$, where $c_p$ is the repositioned alarm condition.* □

## 3.3 Repositioning Technique

To achieve the repositioning discussed in the previous section, we design a two step static analysis technique as described next.

*3.3.1 Step 1 (Intermediate Repositioning).* In the first step, alarm condition of every original alarm $\phi$ reported at $p$ is safely hoisted at the *highest hoisting point along every path* that reaches $p$. The highest hoisting point on a path is identified as the program point $q_h$ such that $cond(\phi)$ is anticipable at $q_h$ but the same condition is no longer anticipable at any program point just before $q_h$. Thus, this step results in hoisting an alarm condition closer to its cause points but also in multiple hoistings of the same condition. For example, the alarm condition $i \geq 0$ && $i \leq 4$ of alarms $A_{17}$ and $A_{19}$ in Figure 1 gets hoisted at two locations: $exit(n_6)$ and $exit(n_8)$. This step discards the third hoisting possible at $entry(n_{13})$ as the condition hoisted at this point always evaluates to *true*. Note that the repositioning obtained after this step is not final and it requires refinement. Thus, we refer to it as *intermediate repositioning*. Section 5 describes the first step in detail.

*3.3.2 Step 2 (Repositioning Refinement).* This step refines the intermediate repositioning by merging the alarm conditions that are candidates for sinking. In this step, *available alarm conditions* are computed from the alarm conditions hoisted in the intermediate repositioning. For every available alarm condition $c$ computed at a program point $p$, we also compute exactly one program point $p_d$ to be associated with $c$. The associated point $p_d$ is the highest program point among the program points where $c$ is available and their nodes dominate the node of $p$. The point $p_d$ is used to reposition $c$ during the final repositioning.

To compute the associated point $p_d$ for an available condition $c$, initially $c$ is associated with the single program point at which it gets generated: a hoisting location from the intermediate repositioning. Later, at the first *meet-program point* $p_m$ where $c$ is observed to have two or more different program points associated with it, the association of $c$ is updated to $p_m$. With this operation we guarantee that, for every condition $c$ available at a program point $p$, there exists only one program point associated with $c$ and the node of the associated point dominates the node of $p$.

For example in Figure 1, $c := i \geq 0$ && $i \leq 4$ gets generated as available alarm condition at the hoisting locations $exit(n_6)$ and $exit(n_8)$ in the intermediate repositioning example (refer Section 3.3.1). At its generation point $exit(n_6)$ (resp. $exit(n_8)$), $c$ gets associated with the same program point. As $entry(n_{11})$ is first meet point where $c$ is available and also has those two program points $exit(n_6)$ and $exit(n_8)$ associated with it, the association of $c$ is changed to $entry(n_{11})$. The location $entry(n_{11})$ associated with $c$ is used later to reposition the condition $c$ finally, described in Section 5.

## 4 INTERMEDIATE REPOSITIONING

This section describes computation of anticipable alarm conditions through backward data flow analysis, and performing the intermediate repositioning using the analysis results.

## 4.1 Anticipable Alarm Conditions Analysis

Given the CFG of a program and original alarms set $\Phi$, this analysis computes alarm conditions of the original alarms as *anticipable alarm conditions* (antconds). For every antcond $c$ computed at any program point, this analysis also computes the input alarms, $\Phi' \subseteq \Phi$, which contribute to anticipability of $c$ at that point. We refer to these alarms $\Phi'$ as the *related original alarms* (rel-alarms) of the condition $c$. The rel-alarms are used to compute traceability links between a *repositioned condition* and its corresponding original alarm(s). Henceforth in the paper, we use repositioned condition to refer to an alarm repositioned in the intermediate or final repositioning, to distinguish it from the original alarms.

*4.1.1 Notations.* Let P be the set of all program points and V be the set of variables in the program. Let C be the set of all conditions that can be formed using program variables, constants, and arithmetic and logical operators. We use tuple $\langle c, \phi \rangle$ to denote an antcond $c \in \mathrm{C}$ along with one of its rel-alarms $\phi \in \Phi$. Thus, the values computed by this backward data flow analysis (*antconds analysis*) at a program point are given by a subset of $\mathrm{L}_b = \mathrm{C} \times \Phi$. For a given set $S \subseteq \mathrm{L}_b$, we define the following:

(1) $condsIn(S) = \{c \mid \langle c, \phi \rangle \in S\}$, returns all antconds in $S$; and
(1) $tuplesOf(c, S) = \{\langle c, \phi \rangle \mid \langle c, \phi \rangle \in S\}$, returns all tuples of a given antcond $c$ in $S$.

*4.1.2 Lattice.* The antconds analysis computes subsets of $\mathrm{L}_b$ flow-sensitively at every program point $p \in \mathrm{P}$. The lattice of these values computed is $\langle B, \sqcap_B \rangle$, where $B$ is the powerset of $\mathrm{L}_b$. As we intend to compute antconds with their corresponding rel-alarms, the meet $\sqcap_B$ is defined as the following: Given $X, Y \in B$,

$$X \sqcap_B Y = \bigcup_{c \in (\ condsIn(X)\ \cap\ condsIn(Y)\ )} tuplesOf(c, X) \ \cup\ tuplesOf(c, Y) \qquad (1)$$

*4.1.3 Data Flow Equations.* Figure 2 shows data flow equations of the antconds analysis in intraprocedural setting: handling of the call nodes is not shown for simplicity of the formalization. We use $AntIn_n$ and $AntOut_n$, in Equations 3 and 2, to denote antconds computed by the analysis at the entry and exit of a node $n$ respectively.

Equation 6 shows processing of every alarm $\phi$ reported for the statement of a node $n$, to generate $cond(\phi)$ as an anticipable condition. Equation 7 denotes that the alarm condition $cond(\phi)$ is not generated as an antcond when it is *implied by* an antcond $c_{in}$ flowing in at the node $n$. However in this case, the alarm $\phi$ is associated with $c_{in}$. Equation 8 denotes computation of antconds transitively at a node $n$. The equation assumes function $wprecond(n, c_p)$ to return the weakest precondition for (1) the statement of a given node $n$, and (2) a postcondition $c_p$. The $\oplus$ is used to denote an arithmetic operator in $\{+, -, /, *\}$. For simplicity, only a few cases of the statements associated with the node $n$ are shown in Equation 8.

## 4.2 Intermediate Repositioning of Alarms

Recall the intermediate repositioning (Section 3.3.1) is implemented by hoisting alarm condition of every alarm *temporarily* at the highest program point in every path reaching to the alarm program point. The highest hoisting point is identified as the point before which the alarm condition is no longer anticipable. We distinguish between the two cases of identifying the highest hoisting points.

**Case 1:** An alarm condition $c$ anticipable at $entry(n)$ is not anticipable at $exit(m)$ when $m$ is a predecessor of $n$ and also a branching

Let $m, n \in \mathrm{N}$; $c, c' \in \mathrm{C}$; $\phi, \phi' \in \Phi$; $u, v \in \mathrm{V}$; $t \in \mathit{Constants}$; and $X, Y \in B$.

$$\mathit{AntOut}_n = \begin{cases} \emptyset & n \text{ is } \mathit{End} \text{ node} \\ \bigsqcap^{B}_{m \in succ(n)} \mathit{AntIn}_m & \text{otherwise} \end{cases} \tag{2}$$

$$\mathit{AntIn}_n = \mathit{Gen}_n(\mathit{AntOut}_n) \cup (\mathit{AntOut}_n \setminus \mathit{Kill}_n(\mathit{AntOut}_n)) \tag{3}$$

$$\mathit{Kill}_n(X) = \left\{ \langle c, \phi \rangle \in X \;\middle|\; \begin{array}{c} n \text{ contains a definition of an} \\ \text{operand of } c \end{array} \right\} \tag{4}$$

$$\mathit{Gen}_n(X) = \mathit{Gen'}_n(X) \cup \mathit{DepGen}_n(\mathit{Kill}_n(X)) \tag{5}$$

$$\mathit{Gen'}_n(X) = \begin{cases} \{ \mathit{process}(\phi, \mathit{condsIn}(X)) \} & \begin{array}{l} n \text{ has an alarm } \phi \\ \text{reported for it} \end{array} \\ \emptyset & \text{otherwise} \end{cases} \tag{6}$$

$$\mathit{process}(\phi, Y) = \begin{cases} \langle c, \phi \rangle & c \in Y,\ c \Rightarrow \mathit{cond}(\phi) \\ \langle \mathit{cond}(\phi), \phi \rangle & \text{otherwise} \end{cases} \tag{7}$$

$$\mathit{DepGen}_n(X) = \begin{cases} \left\{ \begin{array}{c} \langle \mathit{wprecond}(n, c), \phi \rangle \\ \mid \langle c, \phi \rangle \in X \end{array} \right\} & \begin{array}{l} n\text{: } u{=}v; \\ \text{or } n\text{: } u{=}v \oplus t; \\ \text{or } n\text{: } u{=}t \oplus v; \end{array} \\ \emptyset & \text{otherwise} \end{cases} \tag{8}$$

**Figure 2: Data flow equations of the antconds analysis.**

node (i.e. the statement of $m$ is a controlling condition). This case occurs when $c$ is not anticipable through one of the branches coming out of $m$, other than the branch having node $n$. In this case, the antcond $c$ is hoisted at the *entry*($n$). Alternatively, the antconds to be hoisted at *entry*($n$) are given by

$$\mathit{Hoist}_{entry(n)} = \mathit{condsIn}(\mathit{AntIn}_n) \setminus \bigcap_{m \in pred(n)} \mathit{condsIn}(\mathit{AntOut}_m) \tag{9}$$

Condition $i \geq 0$ && $i \leq 4$ of $A_{17}$ (and $A_{19}$) in Figure 1 is anticipable at the *entry*($n_{13}$) but not at the *exit*($n_4$), and $n_4$ is predecessor of $n_{13}$. Thus, the condition is hoisted at the *entry*($n_{13}$). □

**Case 2:** A condition $c$ anticipable at *exit*($n$) is not anticipable at the *entry*($n$) when (1) the node $n$ contains a definition of an operand of $c$ i.e. anticipability of $c$ is killed by $n$, and (2) the node $n$ does not generate any antcond transitively from $c$. In this case, we hoist the condition $c$ at the *exit*($n$). That is, the alarm conditions to be hoisted at *exit*($n$) are given by

$$\mathit{Hoist}_{exit(n)} = \left\{ c \;\middle|\; \begin{array}{c} c \in \mathit{condsIn}(\mathit{Kill}_n(\mathit{AntOut}_n)), \\ \mathit{DepGen}_n(\{c\}) = \emptyset \end{array} \right\} \tag{10}$$

As an example of the above hoisting case, the alarm condition of $A_{17}$ (and $A_{19}$), $i \geq 0$ && $i \leq 4$, is anticipable at the *exit*($n_6$) but not at the *entry*($n_6$) and $n_6$ does not generate anticipable alarm conditions transitively. Thus, the condition gets hoisted at the *exit*($n_6$). Similarly, this condition also gets hoisted at the *exit*($n_8$). □

### 4.2.1 *Discarding Redundant Hoistings.*

Recall hoisting of alarm condition of $A_{17}$ and $A_{19}$ at the *entry*($n_{13}$) (Section 2), where the hoisted alarm condition $i \geq 0$ && $i \leq 4$ always holds at the hoisting location due to the value 0 assigned to $i$ at line 2. We deem this hoisting to be redundant and discard it. Formally, equations 11 and 12 define the *non-redundant hoistings*, where *alwaysTrue*($c, p$) is *true* only if the condition $c$ always holds at $p$.

$$\overline{\mathit{Hoist}}_{entry(n)} = \left\{ c \;\middle|\; \begin{array}{c} c \in \mathit{Hoist}_{entry(n)}, \\ \mathit{alwaysTrue}(c, \mathit{entry}(n)) \neq \mathit{true} \end{array} \right\} \tag{11}$$

$$\overline{\mathit{Hoist}}_{exit(n)} = \left\{ c \;\middle|\; \begin{array}{c} c \in \mathit{Hoist}_{exit(n)}, \\ \mathit{alwaysTrue}(c, \mathit{exit}(n)) \neq \mathit{true} \end{array} \right\} \tag{12}$$

### 4.2.2 *Algorithm.*

The intermediate repositioning of alarms—hoisting at the *entry* and *exit* of all the nodes—is performed by processing every node $n \in \mathrm{N}$ using the equations 11 and 12.

### 4.2.3 *Computing Traceability Links.*

For a given antcond $c$ in $S \subseteq \mathrm{L}_b$, we define function $\mathit{relAlarms}(c, S) = \{\phi \mid \langle c, \phi \rangle \in S\}$ to return the rel-alarms of $c$. The traceability links are generated from a hoisted condition $c$ in $\overline{\mathit{Hoist}}_{entry(n)}$ (resp. $\overline{\mathit{Hoist}}_{exit(n)}$) to its corresponding rel-alarms given by $\mathit{relAlarms}(c, \mathit{AntIn}_n)$ (resp. $\mathit{relAlarms}(c, \mathit{AntOut}_n)$).

### 4.2.4 *Intermediate Repositioning Example.*

Following is the intermediate repositioning obtained for alarms for Figure 1.

(i) $i \geq 0$ && $i \leq 4$ is hoisted at the *exit*($n_6$) and *exit*($n_8$), with both $A_{17}$ and $A_{19}$ as its rel-alarms.
(ii) $n \neq 0$ is hoisted at the *exit*($n_{26}$) (resp. *entry*($n_{29}$)) with $Z_{27}$ (resp. $Z_{29}$) as its rel-alarm.
(iii) $j \geq 0$ && $j \leq 4$ is hoisted at the *exit*($n_{34}$) with $A_{37}$ as its rel-alarm. Furthermore, due to the increment operation at line 38 and transitivity, $j \geq -1$ && $j \leq 3$ also gets hoisted at the *exit*($n_{34}$) with $A_{39}$ as its rel-alarm.

The two conditions hoisted in each of the cases (i) and (ii) belong to different branches of an *if* statement and are candidates for merging (sinking) during the refinement step. In the case (iii), the two conditions repositioned at the same point get merged into a single condition during the refinement step.

## 5 REFINEMENT OF INTERMEDIATE REPOSITIONING

In this section we describe the computation of available alarm conditions (avconds) using forward data flow analysis, and obtaining the final repositioning using the analysis results.

### 5.1 Computing Final Repositioning

Let $c$ be an avcond computed at a program point $p$ from the conditions hoisted in the intermediate repositioning, and $p_r$ be the single program point associated with $c$. Recall the discussion in Section 3.3.2: the location $p_r$ associated with $c$ is computed as the highest program point at which $c$ is available and whose node dominates the node of $p$. We refer to this program point $p_r$ as *repositioning location* of $c$. As we wish to compute avconds transitively, an avcond $c$ at a point $p$ can be transformed version of a condition $c_r$ available at $p_r$. That is, $c$ is transitively computed from $c_r$ along a path from $p_r$ to $p$. We refer to this $c_r$ at $p_r$ as the *repositioning condition* of $c$. Thus, due to the transitivity in avconds computation, we compute the repositioning location $p_r$ and condition $c_r$ for every avcond identified at any program point. These values, $c_r$ and $p_r$, are used later to implement the final repositioning.

We stress that,

- An avcond computed at any point is with exactly one repositioning location $p_r$ and one repositioning condition $c_r$.
- Computing repositioning condition $c_r$ for an avcond $c$ is not required if avconds are not to be computed transitively (because in this case, $c_r$ is same as $c$).
- The associated values, $c_r$ and $p_r$, for an avcond are computed depending on the hoisted conditions and locations in the intermediate repositioning, rather than as dependent on the original alarms input for repositioning.

## 5.2 Available Alarm Conditions Analysis

*5.2.1 Notations.* Let P be the set of all program points and C be the set of all conditions that can be formed using the program variables, constants, and arithmetic and logical operators. We use a function $\mathrm{f} : \mathrm{C} \to \mathrm{C} \times \mathrm{P}$, that maps an avcond $c \in \mathrm{C}$ to its associated repositioning condition $c_r \in \mathrm{C}$ and location $p_r \in \mathrm{P}$. We write the condition $c$ with the associated values as tuple $\langle c, c_r, p_r \rangle$. Thus, the forward analysis (*avconds analysis*) at a program point $p$ computes a subset of $\mathrm{L}_f$, where $\mathrm{L}_f = \{\langle c, c', q\rangle \mid c \in C_p,\ \mathrm{f}(c) = \langle c', q\rangle\}$ and $C_p$ is the set of avconds at $p$.

For a given set $S \subseteq \mathrm{L}_f$ we define:

- $condsIn(S) = \{c \mid \langle c, c_r, p_r\rangle \in S\}$ returns all avconds in $S$.
- $repCond(c, S) = c_r \mid \langle c, c_r, p_r\rangle \in S$, returns the repositioning condition of a given avcond $c$ in $S$.
- $repLoc(c, S) = p_r \mid \langle c, c_r, p_r\rangle \in S$, returns the repositioning location of a given avcond $c$ in $S$.

*5.2.2 Lattice.* As avconds analysis computes subsets of $\mathrm{L}_f$ flow-sensitively at every program point $p \in \mathrm{P}$, we denote the lattice of these values by $\langle F = 2^{\mathrm{L}_f}, \sqcap_F \rangle$. We use ${}^{n}\sqcap_F$ to denote the meet of data flow values at the *entry* of a join node $n$. The meet operation is as shown below (Equation 13), and it is idempotent, commutative, and associative. For simplicity of the equation, we have assumed that the join node $n$ corresponding to a meet operation is known when the meet is performed.
Given $X, Y \in F$ :

$$X \ {}^{n}\sqcap_F \ Y = \bigcup_{c \in (\ condsIn(X)\ \cap\ condsIn(Y)\ )} \{\ mergeInfo(c, entry(n), X, Y)\ \} \tag{13}$$

$$mergeInfo(c, p_m, X, Y) = \begin{cases} \langle c, c_r, p_r\rangle & \langle c, c_r, p_r\rangle \in X,\ \langle c, c_r', p_r'\rangle \in Y,\ p_r = p_r' \\ \langle c, c, p_m\rangle & \text{otherwise} \end{cases}$$

At a meet point $entry(n)$, the above meet operation updates the repositioning condition and location of an avcond $c$ respectively to $c$ and $entry(n)$, only if the repositioning locations of $c$ flowing-in via the two different paths at the meet point are different. In the other case, the values associated with $c$ remain unchanged.

*5.2.3 Data Flow Equations.* Figure 3 shows data flow equations of the avconds analysis that computes avconds transitively in intraprocedural setting. $AvIn_n$ and $AvOut_n$ denote avconds computed with their associated values, respectively, at the entry and exit of a node $n$ (equations 14 and 15). Figure 4 illustrates the computing of $AvOut_n$. Equations 16 and 17 indicate that an avcond is generated

Let $m, n \in \mathrm{N}$; $u, v \in \mathrm{V}$; $c, c', c_r \in \mathrm{C}$; $p_r \in P$;
$t \in Constants$; and $X, Y \in F$.

$$AvIn_n = \begin{cases} \emptyset & n \text{ is } Start \text{ node} \\ \displaystyle {}^{n}\bigsqcap_{F}{}_{\ m \in pred(n)} AvOut_m & \text{otherwise} \end{cases} \tag{14}$$

$$AvOut_n = Gen_{exit(n)} \ \cup\ AvOut'_n \tag{15}$$

$$Gen_{exit(n)} = \{\ \langle c, c, exit(n)\rangle \mid c \in \overline{Hoist}_{exit(n)}\ \} \tag{16}$$

$$AvOut'_n = (AvIn'_n \setminus Kill_n(AvIn'_n)) \ \cup\ DepGen(AvIn'_n)$$

$$AvIn'_n = AvIn_n \ \cup\ Gen_{entry(n)}(AvIn_n)$$

$$Gen_{entry(n)}(X) = \left\{ \langle c, c, entry(n)\rangle \ \middle|\ \begin{array}{c} c \in \overline{Hoist}_{entry(n)}, \\ \langle c', c_r, p_r\rangle \in X,\ \ c' \nRightarrow c \end{array} \right\} \tag{17}$$

$$Kill_n(X) = \left\{ \langle c, c_r, p_r\rangle \in X \ \middle|\ \begin{array}{c} n \text{ contains a definition} \\ \text{of an operand of } c \end{array} \right\} \tag{18}$$

$$DepGen_n(X) = \begin{cases} \left\{ \begin{array}{c} \langle postcond(n, c), c_r, p_r\rangle \\ \mid \langle c, c_r, p_r\rangle \in X \end{array} \right\} & \begin{array}{l} n\text{: } u{=}v; \\ \text{or } n\text{: } u{=}v \oplus t; \\ \text{or } n\text{: } u{=}t \oplus v; \end{array} \\ \emptyset & \text{otherwise} \end{cases} \tag{19}$$

**Figure 3: Data flow equations of the avconds analysis.**

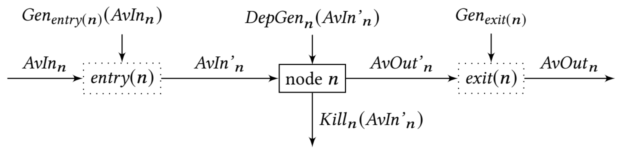


**Figure 4: Processing a node in the avconds analysis**

for every condition hoisted in the intermediate repositioning. The Equation 17 does not generate an avcond for $c \in \overline{Hoist}_{entry(n)}$ when some other avcond in $AvIn_n$ implies $c$. However, such implication handling is not needed in Equation 16, because the antconds are hoisted at the exit of a node only when the node stops anticipability of those conditions.

Equation 19 describes transitive computation of the avconds. It assumes function $postcond(n, c_p)$ to return the *strongest postcondition* for the statement of a given node $n$, and a given precondition $c_p$. Note that this equation does not update the repositioning condition and location of any of the available alarm conditions. Thus, the associated values of an avcond $c$ are updated only when $c$ is generated (equations 16 and 17), or the meet operation is performed.

## 5.3 Computing Traceability Links

For simplicity of the analysis formalization, we separately formalize another forward data flow analysis required to compute the avconds along with their corresponding rel-alarms. This analysis computes subsets of $\mathrm{C} \times \Phi$ flow-sensitively at every program point and has meet operation similar to Equation 1. Figure 5 presents data flow equations of this analysis. We use $fwdIn_n$ and $fwdOut_n$, respectively, to denote the avconds computed at the entry and exit of a node $n$. At any program point, the avconds computed by this analysis will

Let $m, n \in \mathrm{N}$; $u, v \in \mathrm{V}$; $c, c' \in \mathrm{C}$; $\phi, \phi' \in \Phi$; $t \in \mathit{Constants}$; and $X, Y \in B$.

$$\mathit{fwdIn}_n = \begin{cases} \emptyset & n \text{ is } \mathit{Start} \text{ node} \\ \bigsqcap_{B\; m \in pred(n)} \mathit{fwdOut}_m & \text{otherwise} \end{cases}$$

$$\mathit{fwdOut}_n = \mathit{fwdOut}'_n \cup \mathit{Gen}_{exit(n)}$$

$$\mathit{fwdOut}'_n = (\mathit{fwdIn}'_n \setminus \mathit{Kill}_n(\mathit{fwdIn}'_n)) \cup \mathit{TransGen}(\mathit{fwdIn}'_n)$$

$$\mathit{fwdIn}'_n = \mathit{fwdIn}_n \cup \mathit{Gen}_{entry(n)}(\mathit{fwdIn}_n)$$

$$\mathit{Gen}_{entry(n)}(X) = \begin{cases} \{\langle c, \phi\rangle\} & c \in \overline{\mathit{Hoist}_{entry(n)}},\ \langle c', \phi'\rangle \in X \\ & c' \not\Rightarrow c,\ \phi \in \mathit{relAlarms}(c, \mathit{AntIn}_n) \\ \{\langle c', \phi\rangle\} & c \in \overline{\mathit{Hoist}_{entry(n)}},\ \langle c', \phi'\rangle \in X \\ & c' \Rightarrow c,\ \phi \in \mathit{relAlarms}(c, \mathit{AntIn}_n) \end{cases}$$

$$\mathit{Kill}_n(X) = \left\{ \langle c, \phi\rangle \in X \;\middle|\; \begin{array}{c} n \text{ contains a definition} \\ \text{of an operand of } c \end{array} \right\}$$

$$\mathit{TransGen}_n(X) = \begin{cases} \left\{ \begin{array}{c} \langle \mathit{postcond}(n, c), \phi\rangle \\ \mid \langle c, \phi\rangle \in X \end{array} \right\} & \begin{array}{l} n\text{: } u{=}v; \\ \text{or } n\text{: } u{=}v \oplus t; \\ \text{or } n\text{: } u{=}t \oplus v; \end{array} \\ \emptyset & \text{otherwise} \end{cases}$$

$$\mathit{Gen}_{exit(n)} = \{ \langle c, \phi\rangle \mid c \in \overline{\mathit{Hoist}_{exit(n)}},\ \phi \in \mathit{relAlarms}(c, \mathit{AntOut}_n) \}$$

**Figure 5: Computing avconds with related original alarms.**

be same as the avconds computed by the analysis in Section 5.2, with the only difference in the information computed for them.

## 5.4 Final Repositioning Algorithm

Algorithm 1 performs the final repositioning using the avconds.

*5.4.1 Step 1 (Computing Repositioning Locations).* We first identify *avconds to be utilized* for final repositioning. For every avcond $c$ identified, (1) the repositioning condition $c_r$ of $c$ is repositioned at the repositioning location $p_r$ of $c$, and (2) traceability link is created from the repositioned condition $c_r$ to each of the rel-alarms of $c$.

The avconds to be utilized are identified by processing every node $n$ in the program using Equations 20 and 21. These equations respectively compute the avconds that are no longer *available* immediately after entry and exit of a node $n$. These equations are on similar lines of the two cases in Section 4.2. The processing of every node through the two equations ensures the following: (1) each avcond $c$ generated at a program point $p$ gets utilized for repositioning along every path starting at $p$ and ending at the program exit except when it transitively results into some other avcond, and (2) the utilization along any such path is only once and it occurs at the last program point on the path where $c$ is available.

$$\mathit{Conds}_{entry(n)} = \mathit{condsIn}(\mathit{Kill}_n(\mathit{AvIn}'_n)) \tag{20}$$

$$\mathit{Conds}_{exit(n)} = \mathit{condsIn}(\mathit{AvOut}_n) \setminus \bigcap_{s \in succ(n)} \mathit{condsIn}(\mathit{AvIn}_s) \tag{21}$$

As a special case, the algorithm utilizes every condition $c \in \mathit{condsIn}(\mathit{entry}(\mathit{End}))$ for repositioning, because a few avconds like

**Algorithm 1** Algorithm for Final Repositioning

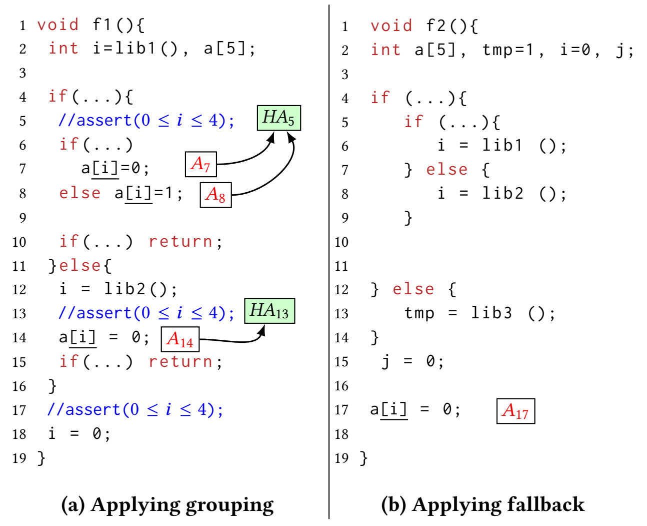
```
procedure PERFORMFINALREPOSITIONING
    for node $n \in \mathrm{N}$ do
        for condition $c \in \mathit{Conds}_{entry(n)}$ do
            reposition($c$, $\mathit{AvIn}'_n$, $\mathit{fwdIn}'_n$);
        for condition $c \in \mathit{Conds}_{exit(n)}$ do
            reposition($c$, $\mathit{AvOut}_n$, $\mathit{fwdOut}_n$);

    /* Special case for the program exit node */
    for condition $c \in \mathit{condsIn}(\mathit{entry}(\mathit{End}))$ do
        reposition($c$, $\mathit{AvIn}_{End}$, $\mathit{fwdIn}_{End}$);

    for point $p \in \mathrm{P}$ do
        Simplify conditions repositioned at $p$.

    /* Postprocessing of repositioned conditions */
    Let $C_R$ be the set of all simplified repositioned conditions.
    Postprocess $C_R$ to eliminate redundancy through grouping.
    Postprocess the non-redundant conditions for fallback.

procedure REPOSITION($c$, $X$, $Y$)
    Reposition $\mathit{repCond}(c, X)$ at $\mathit{repLoc}(c, X)$ with its traceability
    link to every alarm $\phi' \in \{\phi \mid \langle c, \phi\rangle \in Y\}$.
```

$n \neq 0$ in Figure 1 can reach the program end point, but not get computed by any of the equations 20 and 21 for any point.

*5.4.2 Step 2 (Simplifying Repositioned Conditions).* Next every program point is processed to simplify the conditions repositioned at that point. The simplification is performed on conjunction of the repositioned conditions that are related by the same variables. The traceability links for a condition resulting after the simplification are obtained by merging traceability links of the conditions that got simplified. For example, using the avconds at the program end point (line 40) in Figure 1, Step 1 repositions $j \geq 0$ && $j \leq 4$ and $j \geq -1$ && $j \leq 3$ at the $\mathit{exit}(n_{34})$ with their links respectively to $A_{37}$ and $A_{39}$. After the simplification step, these two conditions result in $j \geq 0$ && $j \leq 3$ with its traceability links to both $A_{37}$ and $A_{39}$.

*5.4.3 Step 3 (Postprocessing for Redundancy Elimination).* The repositioning resulting after the previous simplification step may have some redundancy. As an example, consider the code in Figure 6a that has three AIOB alarms reported at lines 7, 8, and 14. The conditions repositioned after Step 2 are at the $\mathit{entry}(n_6)$, $\mathit{exit}(n_{12})$, and $\mathit{entry}(n_{18})$. These three repositioned conditions are shown as assertions at lines 5, 13, and 17, respectively. In this case, the repositioning performed does not reduce the overall alarms count.

Observe that the condition repositioned at $\mathit{entry}(n_{18})$ (line 17) is redundant in presence of the two other repositioned conditions: the other two conditions act as *dominant alarms* for this condition. To improve the repositioning by eliminating such redundancy, (1) we postprocess the repositioned conditions by applying the grouping techniques [20, 24, 31], and (2) discard the repositioned conditions that are identified as *followers.* Applying this postprocessing step to the three repositioned conditions discards the redundant repositioned condition, and reduces the overall alarms count by one.



```
1  void f1(){
2   int i=lib1(), a[5];
3
4   if(...){
5    //assert(0 ≤ i ≤ 4);   HA5
6    if(...)
7      a[i]=0;           A7
8    else a[i]=1;        A8
9
10   if(...) return;
11  }else{
12   i = lib2();
13   //assert(0 ≤ i ≤ 4);  HA13
14   a[i] = 0;  A14
15   if(...) return;
16  }
17  //assert(0 ≤ i ≤ 4);
18  i = 0;
19 }
```

(a) Applying grouping

```
1  void f2(){
2  int a[5], tmp=1, i=0, j;
3
4  if (...){
5     if (...){
6        i = lib1 ();
7     } else {
8        i = lib2 ();
9     }
10
11
12 } else {
13    tmp = lib3 ();
14 }
15  j = 0;
16
17 a[i] = 0;     A17
18
19 }
```

(b) Applying fallback

**Figure 6: Examples to illustrate postprocessing of the repositioned conditions.**

*5.4.4 Step 4 (Postprocessing for Fallback).* In certain cases, the repositioning obtained after Step 3 may increase the number of alarms. This occurs due to (1) the two repositioning goals impacting each other (as illustrated below in certain scenarios), or (2) transitive computation of the antconds and avconds.

Consider the code example in Figure 6b. This example is crafted to illustrate impact of the two repositioning goals on each other. The example has one alarm $A_{17}$. Condition of this alarm, $i \geq 0$ && $i \leq 4$, is not avcond at the $entry(n_{15})$. Indeed, the condition hoisted at the $entry(n_{13})$ gets discarded via Equation 11 during the intermediate repositioning, and due to this discarding the condition is not avcond at the $entry(n_{13})$, and hence at the $entry(n_{15})$. Thus, repositioning of $A_{17}$ results in the two conditions repositioned at the $exit(n_6)$ and $exit(n_8)$ (the repositioned conditions are not shown in Figure 6b).

We discard redundant conditions during the intermediate repositioning to report the alarms closer to their cause points. However, for this example, the discarding increases the alarms count. In the absence of this discarding, the condition gets repositioned only at the $entry(n_{15})$, without increasing the alarms count. This indicates the two repositioning goals impact each other, i.e., reporting alarms closer to their causes might increase the number of alarms.

Hence, we postprocess the repositioned conditions resulting after Step 3. In the postprocessing, we identify situations when the number of alarms increases and revert (fallback approach), i.e., we report original alarms instead of the repositioned ones. As shown by our experimental evaluation, discussed in the next section, fallback is rarely required[1]. □

*Final Repositioning Example:* Applying Algorithm 1 to alarms in Figure 1 results in the following final repositioning. For this example, the postprocessing steps 3 and 4 do not change (improve) the repositioning obtained after Step 2.

(1) $i \geq 0$ && $i \leq 4$ is repositioned at the $entry(n_{11})$ with its traceability links to $A_{17}$ and $A_{19}$, where the repositioning is identified by Equation 21 when applied to the $exit(n_{11})$.
(2) $n \neq 0$ is repositioned at the $entry(n_{32})$ with its traceability links to $Z_{27}$ and $Z_{29}$, where the repositioning is identified when $entry(End)$ is processed as the special case.
(3) $j \geq 0$ && $j \leq 3$ is repositioned at the $exit(n_{34})$ with its traceability links to $A_{37}$ and $A_{39}$. □

Theorem 5.1. *Given a set of alarms Φ, the repositioning of Φ resulting from Algorithm 1 is safe.*

Theorem 5.2. *For any given set of alarms, Algorithm 1 always terminates.*

Theorem 5.3. *For any given set of alarms, Algorithm 1 never increases the number of alarms after repositioning.*

Theorem 5.4. *For a given set of dominant alarms Φ, Algorithm 1 performs sinking of an alarm $\phi \in \Phi$ only if there is an impending reduction in number of overall alarms.*

Due to lack of space, proofs of the above theorems are provided in extended version of the paper, available at http://www.win.tue.nl/~aserebre/ISSTA2018.pdf.

## 6 EMPIRICAL EVALUATION

To determine the practicality and effectiveness of alarms repositioning technique, we performed an empirical evaluation measuring *the reduction in the number of alarms.*

### 6.1 Experimental Setup

*6.1.1 Implementation.* We implemented alarms repositioning on top of analysis framework of a commercial static analysis tool (CSAT). The analysis framework supports analysis of C programs, and allows to implement data flow analyses using function summaries. We implemented limited versions of the both antconds analysis and avconds analysis in inter-functional setting, by solving the data flow analyses in bottom-up order only. In the antconds analysis, we propagated the conditions anticipable at the function-entry to its caller only if the function is called from a single place. In the avconds analysis, all the conditions available at the function-exit are propagated to the caller irrespective of the call invocations of the function. This implementation may result in repositioning an original alarm at multiple locations, and for such cases we resort to the fallback approach.

*6.1.2 Selection of Applications and Alarms.* For the evaluation purpose, we selected in total 20 applications shown in Table 1: 16 open source and 4 industry applications. All these applications were analyzed using CSAT on a machine with i7 2.5GHz processor and 16GB RAM. The open source applications are selected from the benchmarks used for evaluating the grouping techniques [20, 31]: the first 8 applications are from the study performed by Zhang et al. [31] and the next 8 are from the study by Lee et al. [20]. The remaining benchmarks from these studies either were not available or could not be compiled/analyzed using CSAT. The industry applications selected are embedded systems from the automotive domain. All the applications selected are written in *C*.

[1]In fact, there existed only 20 instances that required fallback during repositioning of 33,162 alarms in practice.

Table 1: Experimental Results showing reduction in number of alarms due to their repositioning.

| Application | Size (KLOC) | # Input | # Output | % Reduction | Redundant conds | Fallbacks | Timing Analysis (seconds) | | | Inter-Functional | | Reasons for stopping | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | Original | Repositioning | % OverHead | Mergings | Repos. | Func entry | Branches | Definitions |
| acpid-1.0.8 | 1.7 | 5 | 4 | 20.00 | 0 | 0 | 4.5 | 2.6 | 57.0 | 0 | 0 | 1 | 2 | 1 |
| spell-1.0 | 2.0 | 17 | 17 | 0.00 | 0 | 0 | 15.2 | 4.7 | 30.9 | 0 | 0 | 3 | 9 | 5 |
| barcode-0.98 | 4.9 | 580 | 540 | 6.90 | 0 | 0 | 54.2 | 12.4 | 23.0 | 0 | 0 | 2 | 363 | 175 |
| antiword-0.37 | 27.1 | 748 | 689 | 7.89 | 6 | 0 | 941.7 | 116.1 | 12.3 | 1 | 35 | 93 | 454 | 142 |
| sudo-1.8.6 | 32.1 | 2548 | 2407 | 5.53 | 28 | 0 | 2618.9 | 451.5 | 17.2 | 30 | 54 | 303 | 971 | 1133 |
| uucp-1.07 | 73.7 | 263 | 244 | 7.22 | 4 | 0 | 455.3 | 42.2 | 9.3 | 0 | 0 | 17 | 134 | 93 |
| ffmpeg-0.4.8 | 83.7 | 18523 | 17557 | 5.22 | 169 | 12 | 2059.2 | 535.4 | 26.0 | 85 | 610 | 1252 | 10263 | 6042 |
| sphinxbase-0.3 | 121.9 | 908 | 887 | 2.31 | 24 | 0 | 162.0 | 48.1 | 29.7 | 5 | 41 | 39 | 662 | 186 |
| archimedes-0.7.0 | 0.8 | 2251 | 2146 | 4.66 | 6 | 0 | 27.4 | 8.6 | 31.5 | 1 | 15 | 5 | 1078 | 1063 |
| polymorph-0.4.0 | 1.3 | 10 | 8 | 20.00 | 0 | 0 | 5.3 | 2.1 | 39.5 | 0 | 0 | 1 | 6 | 1 |
| nlkain-1.3 | 2.5 | 89 | 88 | 1.12 | 0 | 0 | 5.0 | 1.9 | 37.2 | 0 | 0 | 0 | 59 | 29 |
| stripcc-0.2.0 | 2.5 | 88 | 77 | 12.50 | 8 | 0 | 17.5 | 2.8 | 16.1 | 0 | 0 | 2 | 49 | 26 |
| ncompress-4.2.4 | 3.8 | 64 | 58 | 9.38 | 0 | 0 | 5.9 | 3.4 | 57.2 | 0 | 0 | 1 | 33 | 24 |
| barcode-0.96 | 4.2 | 440 | 408 | 7.27 | 0 | 0 | 39.3 | 11.3 | 28.7 | 0 | 0 | 2 | 285 | 121 |
| combine-0.3.3 | 10.0 | 454 | 407 | 10.35 | 9 | 0 | 46.7 | 12.5 | 26.8 | 0 | 2 | 9 | 295 | 103 |
| gnuchess-5.05 | 10.6 | 1600 | 1503 | 6.06 | 40 | 0 | 86.4 | 19.9 | 23.0 | 10 | 38 | 90 | 782 | 631 |
| industryApp 1 | 3.4 | 326 | 266 | 18.40 | 0 | 0 | 20.9 | 8.8 | 42.0 | 0 | 20 | 7 | 148 | 111 |
| industryApp 2 | 18.0 | 163 | 162 | 0.61 | 1 | 0 | 44.4 | 11.4 | 25.8 | 0 | 5 | 14 | 112 | 36 |
| industryApp 3 | 18.1 | 1111 | 1007 | 9.36 | 1 | 0 | 72.6 | 21.8 | 30.0 | 0 | 31 | 16 | 800 | 191 |
| industryApp 4 | 30.9 | 2974 | 2541 | 14.56 | 1 | 11 | 1253.5 | 76.4 | 6.1 | 44 | 592 | 167 | 1675 | 699 |
| Total | 453.2 | 33162 | 31016 | 6.47 | 297 | 23 | 7935.9 | 1393.9 | 17.6 | 176 | 1443 | 2024 | 18180 | 10812 |

Table 2: Reduction in alarms error category-wise.

| | #Input | #Output | %Reduction |
|---|---|---|---|
| AIOB | 3464 | 3221 | 7.02 |
| ZD | 985 | 975 | 1.02 |
| OFUF | 24843 | 23564 | 5.15 |
| UIV | 3914 | 3607 | 7.84 |

We selected alarms corresponding to four commonly checked categories of run-time errors: division by zero (ZD), array index out of bounds (AIOB), integer overflow underflow (OFUF), and uninitialized variables (UIV). The alarms selected were postprocessed using the state-of-the-art alarms grouping techniques [20, 24, 31], and we considered only the dominant alarms as input to the repositioning. The grouping technique was employed before repositioning, because as indicated in Sections 1 and 2; we aim at overcoming the limitations of the grouping techniques. Due to the possible side effects caused by function calls, the alarms having *function calls* in their alarm conditions are excluded from grouping [24]. Thus, we also have excluded them from input to the repositioning.

## 6.2 Evaluation Results

Table 1 presents the number of alarms before and after the repositioning, and the percentage of alarms reduced: columns *#Input*, *#Output*, and *%Reduction* respectively. The percentage of reduced alarms ranges between 0 and 20%, with the median reduction 7.25% and the average reduction 6.47%. The average reduction on open source applications is 5.41% as compared to the 13.07% on the industry applications. In a follow-up study we will study the reasons for higher reduction on the industry applications.

Table 1 also details the improvements resulting from the post-processing of repositioned conditions (steps 3 and 4 in Section 5.4). Column *Redundant conds* of Table 1 presents the number of repositioned conditions identified as *followers*, i.e., redundant conditions, by the grouping technique in Step 3 (Section 5.4.3). It indicates that around 1% of the repositioned conditions computed by Step 2 are identified as redundant by Step 3. Column *Fallbacks* presents the number of instances, 23, where fallback got applied (Section 5.4.4). This indicates that the fallback gets applied rarely in practice. Our manual analysis of these instances showed that (a) three instances were due to the kind of interfunctional implementation we had for avconds/antconds computation; (b) 18 instances were because of the two repositioning goals impacting each other; and (c) the other 2 cases were due to computing the conditions transitively.

To compute the performance overhead incurred by the repositioning, we compared the time for repositioning (column *Repositioning*) to the time to analyze the code for the categories selected (column *Original*). On average, the repositioning added performance overhead of 17.6% while it reduced the alarms count by 7.25%.

To investigate which run-time categories are benefited the most through repositioning, we performed evaluation by repositioning alarms in each category separately. The evaluation results in Table 2 shows that reduction percentages for AIOB, OFUF, and UIV are comparable and are lowest for ZD. Our analysis of the results for ZD showed that the division operations mostly appear in one of the *if* branches only. In such cases, repositioning is unable to merge such an alarm with another.

## 6.3 Results Discussion and Future Work

The reduction in alarms due to repositioning, 7.25%, is on top of the alarms reduction obtained through the grouping techniques

[20, 24, 31]. Thus, the reduction indicates failure of the grouping techniques to merge those many alarms. Recall that the repositioning technique also uses alarms grouping to identify and remove redundant repositionined conditions (Section 5.4.3). Thus, these techniques help each other when used to reduce the number of alarms generated. Furthermore we observe that, if implication is handled during computation of the antconds and avconds (Equations 6 and 17), the presented repositioning technique subsumes the grouping of alarms, and therefore grouping of the original alarms can be skipped when repositioning is performed.

We believe that the backward inter-functional repositioning can provide more benefits in manual inspection of alarms. The expected gain is due to eliminating code traversals from the functions of their original reporting to the functions having repositioned alarms. We consider empirical evaluation of this gain as future work.

From the evaluation, we also see that while merging alarms originally reported in different functions is not frequent (column *#Mergings*), repositioning alarms across the function boundaries is quite common (column *#Repos.*). Our attempt to understand the reasons for stopping backward repositioning showed the following. (1) For around 6% of the repositioned alarms, the backward repositioning stopped at the entry of a function as the function was called from more than one place (column *#Func entry*); (2) For around 59% of the repositioned alarms, the backward repositioning stopped due to branching nodes: the repositioned warning appears only in one branch of the *if* statement (column *#Branches*); (3) For the other repositioned alarms (35%), the backward repositioning stopped due to definitions of a variable appearing in the alarm conditions (column *#Definitions*). We plan to improve the repositioning results of Case 2, by designing a strategy to identify branching conditions that are *irrelevant* to the alarm conditions, therefore they should not stop the repositioning process.

## 7 RELATED WORK

Several approaches to postprocessing of alarms have been proposed in the literature [14, 26]. The approaches are like grouping, ranking, pruning, automated false positives elimination, and even manual inspections of the alarms. Furthermore, there exists several techniques for each of the approaches. Thus, we limit the comparisons of our technique to the approaches/techniques that are closely related.

As discussed in Section 2, grouping of alarms based on similarity/correlations is the most related approach to postprocess the alarms. This approach [20, 24, 31] has helped to reduce the number of alarms significantly (34 to 60%). However, they fail to group alarms in certain cases (discussed in Section 2) and also report the alarms away from their causes. Gerhke et al. [13] have used repositioning approach similar to ours to overcome this limitation, however sometimes they end up repositioning more alarms than the alarms input for repositioning (Section 2). Also, their approach does not perform sinking of alarms when it helps to reduce overall alarms count. Furthermore, they do not maintain traceability link(s) between a repositioned alarm and the corresponding original alarm(s). In the absence of these links, reviewer needs to perform additional code traversals but in forward direction to locate the original alarms corresponding to a repositioned alarm, when (1) an error is found at repositioned alarm, and (2) correction is needed at the original alarm program point. Our technique has been designed to overcome these limitations.

Cousot et al. [8] have proposed usage of necessary preconditions which are hoisted to the method entry, corresponding to the inevitable checks within a method. The conditions hoisting is used in the context of providing the preconditions required by the Design by Contract [22]. On similar lines, Das et al. [10] have proposed *angelic verification* technique for verification of open programs. This technique is intended to prune the alarms generated during verification of open programs with unconstrained environment. The alarms repositioning technique is applicable to programs with both constrained and unconstrained environments.

Muske and Khedker [25] have proposed cause points analysis to handle alarms effectively during the manual inspections. In their approach, instead of alarms, ranked causes to the alarms are reported and user inputs are sought in several iterations to resolve the alarms. This approach does not reduce the number of alarms without user intervention.

We observe that the alarms repositioning can be applied in conjunction with other alarms postprocessing techniques to complement each other, and also, we believe that the combinations will provide more benefits as compared to the benefits obtained by applying them individually. Benefits of such combinations should be subject of further studies.

## 8 CONCLUSION

We have proposed a novel alarms postprocessing technique intended mainly to reduce the alarms count. The technique is also designed to report alarms as close as possible to their cause points.

We have evaluated the technique using a large set of alarms generated on twenty open source and industry applications. The technique reduces the number of alarms up to 20%, with median reduction of 7.25% on top of the state-of-the-art grouping techniques. These grouping techniques fail to merge and reduce those alarms as they report alarms in their original form and at their original locations. On the contrary, our repositioning approach identifies suitable locations for reporting of the alarms, targeting not only reducing the alarms count but also reporting them closer to their cause points. Furthermore, we observe that the repositioning technique can replace the grouping as a postprocessing technique.

We believe that the repositioning technique, being orthogonal to many of the existing approaches to postprocess alarms, can be applied in conjunction with those approaches.

## REFERENCES

[1] [n. d.]. Polyspace Code Prover. http://in.mathworks.com/products/polyspace-code-prover/. [Online: accessed 30-Jan-2017].
[2] Frances E. Allen. 1970. Control Flow Analysis. In *Symposium on Compiler Optimization*. ACM, New York, NY, USA, 1–19. https://doi.org/10.1145/800028.808479
[3] Nathaniel Ayewah and William Pugh. 2010. The Google FindBugs Fixit. In *International Symposium on Software Testing and Analysis*. ACM, New York, NY, USA, 241–252. https://doi.org/10.1145/1831708.1831738
[4] Nathaniel Ayewah, William Pugh, J. David Morgenthaler, John Penix, and YuQian Zhou. 2007. Evaluating Static Analysis Defect Warnings on Production Software. In *Workshop on Program Analysis for Software Tools and Engineering*. ACM, New York, NY, USA, 1–8. https://doi.org/10.1145/1251535.1251536
[5] Moritz Beller, Radjino Bholanath, Shane McIntosh, and Andy Zaidman. 2016. Analyzing the State of Static Analysis: A Large-Scale Evaluation in Open Source Software. In *International Conference on Software Analysis, Evolution, and Reengineering*, Vol. 1. 470–481. https://doi.org/10.1109/SANER.2016.105

[6] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. 2010. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Commun. ACM* 53, 2 (2010), 66–75. https://doi.org/10.1145/1646353.1646374

[7] Maria Christakis and Christian Bird. 2016. What Developers Want and Need from Program Analysis: An Empirical Study. In *International Conference on Automated Software Engineering*. ACM, New York, NY, USA, 332–343. https://doi.org/10.1145/2970276.2970347

[8] Patrick Cousot, Radhia Cousot, Manuel Fähndrich, and Francesco Logozzo. 2013. *Automatic Inference of Necessary Preconditions*. Springer Berlin Heidelberg, Berlin, Heidelberg, 128–148.

[9] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2012. Frama-c. In *International Conference on Software Engineering and Formal Methods*. Springer, 233–247.

[10] Ankush Das, Shuvendu K. Lahiri, Akash Lal, and Yi Li. 2015. *Angelic Verification: Precise Verification Modulo Unknowns*. Springer International Publishing, Cham, 324–342.

[11] Vinicius Rafael Lobo de Mendonca, Cassio Leonardo Rodrigues, Fabrízzio Alphonsus A de M. N. Soares, and Auri Marcelo Rizzo Vincenzi. 2013. Static analysis techniques and tools: A systematic mapping study. In *International Conference on Software Engineering Advances*.

[12] Isil Dillig, Thomas Dillig, and Alex Aiken. 2012. Automated Error Diagnosis Using Abductive Inference. In *Conference on Programming Language Design and Implementation*. ACM, New York, NY, USA, 181–192. https://doi.org/10.1145/2254064.2254087

[13] Marcel Gehrke. 2014. *Bidirectional Predicate Propagation in Frama-C and its Application to Warning Removal*. Master's thesis. Hamburg University of Technology.

[14] Sarah Heckman and Laurie Williams. 2011. A Systematic Literature Review of Actionable Alert Identification Techniques for Automated Static Code Analysis. *Inf. Softw. Technol.* 53, 4 (2011), 363–387. https://doi.org/10.1016/j.infsof.2010.12.007

[15] David Hovemeyer and William Pugh. 2004. Finding Bugs is Easy. *SIGPLAN Not.* 39, 12 (2004), 92–106. https://doi.org/10.1145/1052883.1052895

[16] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why Don't Software Developers Use Static Analysis Tools to Find Bugs?. In *International Conference on Software Engineering*. IEEE Press, Piscataway, NJ, USA, 672–681.

[17] Uday Khedker, Amitabha Sanyal, and Bageshri Sathe. 2009. *Data flow analysis: theory and practice*. CRC Press.

[18] Yit Phang Khoo, Jeffrey S. Foster, Michael Hicks, and Vibha Sazawal. 2008. Path Projection for User-centered Static Analysis Tools. In *Workshop on Program Analysis for Software Tools and Engineering*. ACM, New York, NY, USA, 57–63. https://doi.org/10.1145/1512475.1512488

[19] Lucas Layman, Laurie Williams, and Robert St. Amant. 2007. Toward Reducing Fault Fix Time: Understanding Developer Behavior for the Design of Automated Fault Detection Tools. In *International Symposium on Empirical Software Engineering and Measurement*. 176–185. https://doi.org/10.1109/ESEM.2007.11

[20] Woosuk Lee, Wonchan Lee, and Kwangkeun Yi. 2012. Sound Non-statistical Clustering of Static Analysis Alarms. In *International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer-Verlag, Berlin, Heidelberg, 299–314. https://doi.org/10.1007/978-3-642-27940-9_20

[21] Ravi Mangal, Xin Zhang, Aditya V. Nori, and Mayur Naik. 2015. A User-guided Approach to Program Analysis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. ACM, New York, NY, USA, 462–473.

[22] Bertrand Meyer. 2002. *Design by contract*. Prentice Hall.

[23] Tukaram Muske. 2014. Improving Review of Clustered-Code Analysis Warnings. In *International Conference on Software Maintenance and Evolution*. IEEE Computer Society, Washington, DC, USA, 569–572. https://doi.org/10.1109/ICSME.2014.97

[24] Tukaram Muske, Ankit Baid, and Tushar Sanas. 2013. Review efforts reduction by partitioning of static analysis warnings. In *International Working Conference on Source Code Analysis and Manipulation*. 106–115. https://doi.org/10.1109/SCAM.2013.6648191

[25] Tukaram Muske and Uday P. Khedker. 2016. Cause Points Analysis for Effective Handling of Alarms. In *International Symposium on Software Reliability Engineering*. 173–184. https://doi.org/10.1109/ISSRE.2016.45

[26] Tukaram Muske and Alexander Serebrenik. 2016. Survey of approaches for handling static analysis alarms. In *International Working Conference on Source Code Analysis and Manipulation*. 157–166.

[27] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. 1999. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[28] Caitlin Sadowski, Jeffrey van Gogh, Ciera Jaspan, Emma Söderberg, and Collin Winter. 2015. Tricorder: Building a Program Analysis Ecosystem. In *International Conference on Software Engineering*. IEEE Press, Piscataway, NJ, USA, 598–608.

[29] YN Srikant and Priti Shankar. 2007. *The compiler design handbook: optimizations and machine code generation*. CRC Press.

[30] Arnaud Venet. 2008. A Practical Approach to Formal Software Verification by Static Analysis. *Ada Lett.* XXVIII, 1 (2008), 92–95. https://doi.org/10.1145/1387830.1387836

[31] Dalin Zhang, Dahai Jin, Yunzhan Gong, and Hailong Zhang. 2013. Diagnosis-Oriented Alarm Correlations. In *Asia-Pacific Software Engineering Conference*, Vol. 1. 172–179. https://doi.org/10.1109/APSEC.2013.33

[32] Jiang Zheng, Laurie Williams, Nachiappan Nagappan, Will Snipes, John P. Hudepohl, and Mladen A. Vouk. 2006. On the Value of Static Analysis for Fault Detection in Software. *IEEE Trans. Softw. Eng.* 32, 4 (2006), 240–253. https://doi.org/10.1109/TSE.2006.38