Improving Review of Clustered-code Analysis Warnings

Tukaram Muske
Tata Research Development and Design Center
54-B, Hadapsar I.E., Pune, India
Email: t.muske@tcs.com

Abstract—While software verification using static analysis tools has proved its significance in early defect detection, the tools often fail to analyze many real world systems. Clustering, breaking a system into multiple clusters, is commonly used technique to scale these tools to large systems. Due to the imprecise nature of static analysis and the conservative approach taken for the inter-cluster communication (data sharing), a large number of analysis warnings are generated. All these reported warnings need to be reviewed manually to check if they represent a defect, and it is a time consuming process. We observe many of the reported warnings are common to multiple clusters and reviewing each of them individually incurs redundancy. We present an approach to improve the manual review process by eliminating the redundancy. This is achieved by grouping the inter-cluster common warnings such that review of a grouped warning under given constraint guarantees same review judgment for the other cluster warnings in the same group. Empirical results obtained with the presented approach indicate that - a) on an average, 45% of total warnings are common to multiple clusters, and b) with the proposed grouping technique, the manual efforts required to review these common warnings are reduced by 60%.

Keywords—Static Analysis, Clustering, Manual Review of Warnings, False Positives.

1. Motivation

Software systems are growing day by day in size and complexity, and many of the real world systems have several millions lines of code (MLOC). Static analysis tools are being increasingly used to assure code quality during development and maintenance of these systems [1]. However, the analysis tools fail to analyze a large software system as a whole [2]. This is because, the whole-system analysis demands more memory and time resources than available, and in practice, satisfying this demand is not always feasible.

Clustering [3] is one of the techniques attempted so far to scale the static analysis tools to very large software systems. The scalability is achieved by splitting the system code into smaller code clusters. A large system usually constitutes many functionalities (designated tasks) implemented independently of or in communication with other functionalities. The system code implementing a functionality is viewed as a cluster, and it is denoted by the function that is entry to the functionality code. Such a cluster being smaller and less complex than the original system is analyzable by static analysis tools to produce cluster-specific results. The clusters of a system can be user provided entry functions, or automatically identified (uncalled functions), or mixture of the both.

Static analysis tools are also known to have a high rate of false positives ([4], [5], [6]) and it varies as per the precision and completeness of the tool. Further, the conservative approach taken for the inter-cluster communication through data-sharing leads to the increased number of warnings. Each of the reported warnings needs to be reviewed manually to ensure it is not a defect. Such a review is found to take considerable time as it requires understanding program behavior at the warning point, and doing it for numerous warnings is a time consuming and tedious process [5]. Many times the involved efforts in reviewing the warnings have been found to be a reason for skipping the review process [7], and sometimes the reason to avoid usage of the static analysis tools [8]. To the best our knowledge nothing has been done to improve the reviewing of warnings produced during the clustered-code analysis.

This paper presents an approach to improve reviewing of cluster-specific warnings. It is based on our observation that many of generated warnings are common to two or more clusters since most of the code is reused across multiple clusters. For example, \texttt{w1} warning (divide by zero) in Figure 1 is reported for both the tasks (\texttt{Task1} and \texttt{Task2}) as it corresponds to the same program point in the reused function \texttt{f2}. Similarly, \texttt{w2} (array index out bound) also gets reported twice being a common-point warning. Henceforth in the paper, such common-point warnings are referred to as common warnings. The example warnings \texttt{w1} and \texttt{w2} are generated, respectively, because static analysis tools cannot compute precise values for \texttt{p2} and the expressions denoted by \texttt{\*}. The cluster-wise reporting of such common warnings is essential as their behavior can vary depending on the context of their associated clusters. For instance, \texttt{w2} may represent a defect (out of bound array access) in \texttt{Task1} while it being a safe array access for \texttt{Task2}, and vice versa.

Since the analysis results are cluster-specific, the output warnings are reviewed cluster-wise. The common warnings too are validated cluster-wise, because reviewing a common warning in the context of multiple clusters at the same time - a) requires switching between multiple clusters and it becomes tedious, b) information required while reviewing may become too large to manually analyze, and c) if any review assisting tools is used, it may not scale on the multiple clusters. We observe the cluster-wise reviewing leads to multiple reviews of a common warning and doing this incurs redundancy. For example, \texttt{w1} gets reviewed twice, once for each cluster separately, and the second review is completely redundant as it gets invalidated at \texttt{\*} context irrespective of the cluster. Further, it is intuitive that reviewing \texttt{w2} separately for both...
the tasks incurs partial redundancy.

This paper presents an approach to group common warnings and report a constraint for each group to eliminate the review redundancy. The grouping is such that review judgment of a grouped warning under given constraint is applicable to reviews of all other warnings. This way, only one warning from a group needs to be reviewed if first reviewed warning is judged as safe or unsafe under the given constraint. When a warning cannot be judged under the given constraint, the groups-reporting allows to log and reuse some part of review information while reviewing other warnings of the same group. Our experimental results demonstrated that around 45% of total warnings are common warnings, and with the proposed grouping technique the manual efforts involved in their reviewing are reduced by more than half.

The grouping approach is described in detail in Section II, and the experimental results are discussed in Section III. Section IV and V, respectively, presents the related work and conclusion.

II. PROPOSED APPROACH

This section describes our approach to improve reviewing of warnings through their systematic grouping and reporting. We start by defining few terms.

A. Definitions

**Review scope function.** A warning validation starts at the lowest code scope and later the scope is incremented until decision about the warning is made. For example, validating w2 in Task1 starts at the scope of f2, and on observing its insufficiency in decision-making the scope is increased to f1 and later to Task1. We refer such a visited function as a review-scope (or simply scope) function.

**Scope functions chain.** It is an ordered list of scope functions traversed during a warning inspection. For instance - a) validating w2 in Task2 has scope chain denoted by \( f2 \rightarrow f1 \rightarrow Task2 \) since the decision-making requires Task2 scope, b) inspecting w1 of any cluster has chain \( f2 \rightarrow f1 \) since the warning gets invalidated at the f2 scope itself. We refer the function at the end of a chain as top of the chain. For instance, f1 is the top of \( f2 \rightarrow f1 \) chain.

**Overlapped scope chain.** It can be observed that the scope chains traversed while validating a common warning against their associated clusters have initial portion overlapped. A reversed and joint call-graph of the clusters in relevance to a common warning is suitable to compute the overlapped scope chains (OSC). For instance, Figure 2 presents a few examples of such call graphs. Figure 1 (a) corresponds to the code sample in Figure 1, and it indicates \( f2 \rightarrow f1 \) is the OSC while inspecting the w1 (or w2) against both the clusters. Further, a common warning can have multiple OSCs, and its example is shown in Figure 2(b) where both the chains \( f3 \rightarrow f1 \) and \( f3 \rightarrow f2 \) are OSCs.

**Must-OSCs.** We refer a maximum-length OSC as must-OSC if - a) all the entry functions of clusters associated with common warnings are strictly reachable from its top, and b) none of these entry functions is reachable from the warning function without going through the top. For example, \( f2 \rightarrow f1 \) in Figure 2(a), and both the OSCs \( f3 \rightarrow f1 \) and \( f3 \rightarrow f2 \) in Figure 2(b) are must-OSCs. Figure 2(c) has \( f2 \) as the only element of its must-OSC.

![Fig. 1: Clustered-code analysis warning examples](image)

![Fig. 2: Overlapped scope chain examples](image)

---

**TABLE I: Reporting of groups of common warnings**

<table>
<thead>
<tr>
<th>Group ID</th>
<th>Sub-group Level</th>
<th>Warning ID</th>
<th>Cluster</th>
<th>Other details</th>
<th>Sub-group tops (length)</th>
<th>Group tops (length)</th>
<th>Info to be logged</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>w1</td>
<td>Task1</td>
<td>-</td>
<td>f(2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>w3</td>
<td>Task1</td>
<td>-</td>
<td>f(2), f(2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>w5</td>
<td>Task3</td>
<td>-</td>
<td>f(2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>w5</td>
<td>Task2</td>
<td>-</td>
<td>f(2)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

B. Grouping of common warnings

In practice, every cluster spans over thousands of lines of code and contains several calls to hundreds of functions that are reused across multiple clusters. We observe that while inspecting common warnings, a reviewer is usually not aware of the must-OSCs. Thus, repetitive manual checking of the OSCs get performed leading to redundant efforts. We eliminate this redundancy through systematic grouping of common warnings. It includes -

1) differentiating the warnings that are unique and common to the clusters,
TABLE II: Experimental Results

<table>
<thead>
<tr>
<th>Application</th>
<th>Verification property</th>
<th>Total warnings</th>
<th>Grouping analysis time (seconds)</th>
<th>Grouped warnings</th>
<th>Groups formed</th>
<th>Groups with 5+ warnings</th>
<th>Groups with multiple top OSCs</th>
<th>Groups ALM</th>
<th>Sub-grouping analysis time (seconds)</th>
<th>Groups having sub-groups</th>
<th>Groups with 5+ sub-groups</th>
<th>Groups ALM</th>
<th>Sub-groups ALM</th>
</tr>
</thead>
<tbody>
<tr>
<td>App 1</td>
<td>ZD</td>
<td>825</td>
<td>1</td>
<td>494</td>
<td>117</td>
<td>33</td>
<td>2</td>
<td>2.23</td>
<td>75</td>
<td>53</td>
<td>53</td>
<td>327</td>
<td>3.11</td>
</tr>
<tr>
<td></td>
<td>AIOB</td>
<td>12946</td>
<td>22</td>
<td>5766</td>
<td>7199</td>
<td>229</td>
<td>92</td>
<td>2.61</td>
<td>75</td>
<td>469</td>
<td>1058</td>
<td>3.43</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OFUF</td>
<td>99762</td>
<td>67</td>
<td>52412</td>
<td>8630</td>
<td>2234</td>
<td>338</td>
<td>2.48</td>
<td>620</td>
<td>3473</td>
<td>40290</td>
<td>2.84</td>
<td></td>
</tr>
<tr>
<td>App 2</td>
<td>ZD</td>
<td>91</td>
<td>0.1</td>
<td>4</td>
<td>2</td>
<td>0</td>
<td>1.5</td>
<td>0</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>AIOB</td>
<td>72</td>
<td>0.5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OPUE</td>
<td>531</td>
<td>1.1</td>
<td>77</td>
<td>48</td>
<td>11</td>
<td>1</td>
<td>1.25</td>
<td>1</td>
<td>25</td>
<td>24</td>
<td>1.74</td>
<td></td>
</tr>
</tbody>
</table>

2) cluster-wise reporting of unique warnings, and grouping and reporting the common warnings together.
3) for each group, reporting the top and length (number of scope functions) of each must-OSC associated with the grouped warnings.

Table I presents reporting example for groups of w1 and w3 warnings from Figure 2. To get maximum benefit out of the grouping approach, the common warnings are further partitioned into sub-groups based on their length of must-OSC. Table I also presents an example of two sub-groups corresponding to the four common warnings (w5) in Figure 2(d). The warnings in clusters Task1 and Task2, and Task3 and Task4 are placed in different sub-groups since length of their must-OSC vary. The top corresponding to each sub-group is also reported.

C. Reviewing of grouped warnings

Reviewing of both the unique and common warnings is performed cluster-wise. During inspection of a common warning -

a) if the warning is judged as a false positive without crossing any of the tops reported for its group, all other grouped warnings are also false positives and they all are eliminated at once.

b) if not (a) and it is invalidated without crossing a top of its subgroup, then all the warnings of its sub-group are eliminated together.

c) if it is not the case in both (a) and (b), user can log the review information available at the top functions, and reuse it later when the rest common warnings are reviewed in the context of their associated clusters. For example, during review of w2 in Task1 (group 1 in Table I) a reviewer can log that the warning judgment depends on p. This logged information can be reused while reviewing w2 of Task2, and it helps to start the inspection from calls to f1 function. This way, logging of review information also further avoids some of the review redundancy.

III. EXPERIMENTAL RESULTS

We used TCS ECA [9] to implement our proposed grouping technique and check its impact on review efforts reduction. We selected below described applications for our experiments.

1) App 1: An infotainment system having 14 MLOC code size and consisting of 98 clusters running in parallel. These clusters varied from 4 KLOC to 700 KLOC, and they consisted of total 31288 functions of which 9327 were reused across multiple clusters.

2) App 2: An embedded system (10 KLOC size) distributed in 29 clusters. This system spanned over 83 functions of which 14 were reused.

These applications were verified for Zero Division (ZD), Array Index Out of Bound (AIOB), Illegal Dereference of a Pointer (IDP), and OverFlow-UnderFlow (OFUF) since these are commonly checked properties in software verification. The experimental results are presented in Table II. Columns Groups ALM and Sub-groups ALM in this table denotes average length of must-OSC associated with the formed groups. Listed below are a few observations made from these results.

1) Around 45% of total warnings are grouped being common warnings, and among them, only 18% are unique.
2) A high percentage of groups formed (25%) include 5 or more warnings.
3) Around 6% of the groups have multiple tops indicating that reviewing of these warnings require higher efforts than efforts in reviewing grouped warnings with single top.
4) The ALM of the sub-groups is always greater than ALM of the groups, because must-OSC of a group are contained in the must-OSC of its sub-groups.

TABLE III: Manual Review Results

<table>
<thead>
<tr>
<th>Application</th>
<th>Property</th>
<th>Warnings reviewed</th>
<th>Groups (warnings) eliminated at once</th>
<th>Sub-groups (warnings) eliminated at once</th>
<th>Log info reused groups (warnings)</th>
</tr>
</thead>
<tbody>
<tr>
<td>App 1</td>
<td>AIOB</td>
<td>90</td>
<td>65 (224)</td>
<td>4 (133)</td>
<td>5 (30)</td>
</tr>
<tr>
<td></td>
<td>OFUF</td>
<td>60</td>
<td>40 (244)</td>
<td>5 (68)</td>
<td>12 (23)</td>
</tr>
<tr>
<td>App 2</td>
<td>ZD</td>
<td>2</td>
<td>1 (2)</td>
<td>0</td>
<td>1 (2)</td>
</tr>
<tr>
<td></td>
<td>IDP</td>
<td>2.5</td>
<td>4 (9)</td>
<td>3 (7)</td>
<td>14 (38)</td>
</tr>
<tr>
<td></td>
<td>OFUF</td>
<td>35</td>
<td>9 (31)</td>
<td>2 (4)</td>
<td>1 (4)</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>180</td>
<td>119 (487)</td>
<td>12 (30)</td>
<td>33 (75)</td>
</tr>
</tbody>
</table>

In order to check impact of the presented grouping technique, we randomly selected 180 warnings belonging to different groups and reviewed them manually. Table III presents the results of this activity. The results indicate -

1) reviewing of 131 warnings collectively lead to elimination of 517 warnings,
2) around 66% (119 out of 180) groups got eliminated just by inspecting a single warning from each group,
3) sub-grouping helped to eliminate some of the warnings when the top level grouping failed to do so,
4) review information gathered while inspecting 33 warnings got reused during reviews of other 42 warnings.

To check impact of the grouping technique on efforts reduction, we reviewed 640 warnings belonging to the above 180 groups in two different settings - 1) individual reviewing (without grouping), and 2) groups-based reviewing. This experiment indicated that around 50%-70% of the review efforts are saved by group-based warnings elimination, and logging and reviewing of the review information. This saving is found to be varying as per skills of the reviewers, tools support used, previous knowledge of the code, etc. It is to note that the saving achieved is with respect to the common warnings only, as warnings unique to a cluster are not part of the grouping. Further details about this experiment are avoided due to the lack of space.

IV. RELATED WORK

A literature survey of actionable alert identification techniques (AAITs) [10] done by Heckman, summarizes 21 AAITs that are used in classifying and prioritizing the actionable alerts. These AAITs improve the review process by identifying the alarms that are most likely to be the true errors. In contrast, our presented grouping technique considers each input warning to be of the same priority and hence all of them are actionable. We identify a warning as redundant only if some other grouped warning is found to be a false positive after manual review. None of these AAITs is observed to be similar to our presented grouping technique.

Rival has proposed a framework for semi-automatic investigation [4] to help users in judging a given warning as true error or false warning. This framework reduces the burden of tracking the source of alarms in Astrée [11]. However, the investigation techniques are applicable only to the warnings belonging to a cluster. Muske et al. [5] and Lee et al. [12] have proposed grouping techniques such that usually only one warning from a group gets reviewed. These techniques use must reachability and liveness of warning expressions [5], and alarm dominance [12] to group the warnings. On the similar lines, Zhang et al. [6] have used alarm correlations to group the warnings and assist their inspection. While these grouping techniques have similar objective to that of our presented technique, they fail to group warnings belonging to multiple clusters. This is because, the grouping criteria used in these techniques are applicable only to warnings that are from same cluster. Further, while grouping the warnings, none of these techniques consider function chains traversed during a warning review.

V. CONCLUSION

In our experiments, we observed a high percentage of warnings, around 45%, as common warnings because software applications reuse a lot of code. Due to this lot of warnings get reported multiple times. We have demonstrated that although the analysis results are cluster-specific, all of them need not be reviewed separately. The redundancy incurred while reviewing them individually in the context of their associated tasks is eliminated by their grouping. The main contribution of our presented approach lies in identifying must-OSCs based grouping criterion, reporting tops of the identified must-OSCs for each group, and making them part of a warning review process.

The experiments revealed that around 66% of the total groups formed are eliminated just by reviewing a single warning from each of the group. This groups-based elimination, and the logging of review information when decision about a warning at a top function can not be made, collectively lead to 60% of manual efforts reduction. Without the grouping approach, the saved efforts un-necessarily would have been spent in inspecting the same code multiple times. Further, our experiments also demonstrated that sub-grouping is helpful to eliminate some of the warnings when the first level grouping fails to do so.

We have considered AIOB, ZD, OFU, and IDP properties in our experiments, however, the proposed grouping technique can be applied to the warnings generated for other properties such as un-initialized variables. Although we have used embedded domain application written in C for the experiments, we expect similar benefits on other domain/language applications as well due to common coding practices.

REFERENCES